Creating a Neuron:

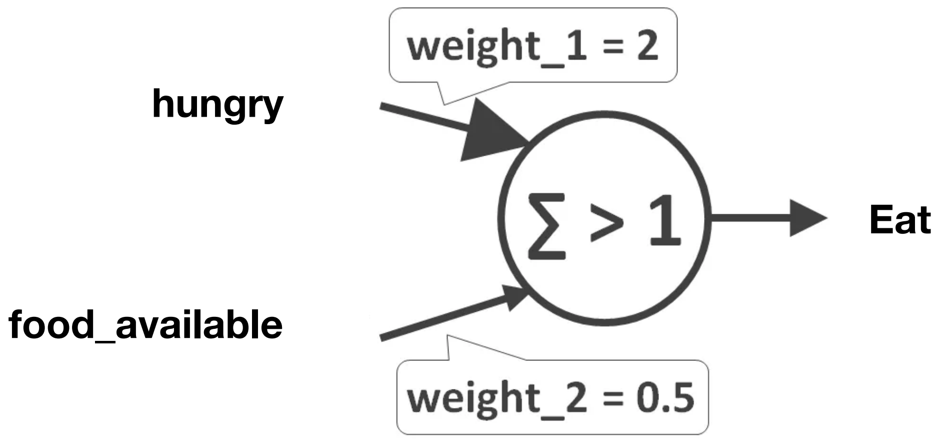
Input		Output
hungry	food_available	Eat
1	1	1
1	0	0
0	1	0
0	0	0

```python
def predict(hungry, food_available):
    storm = 0
    if hungry == 1 and food_available == 1:
        Eat = 1
    return Eat

print(predict(1, 1))
print(predict(1, 0))
print(predict(0, 1))
print(predict(0, 0))
```

In practice, we begin with a random prediction and gradually refine it through iterations until we achieve a "good enough" result. To accomplish this, we simulate the behavior of neurons in the brain. Neurons function like complex processors that take multiple inputs, perform computations, and produce an output.

As we have two inputs here, we need to keep two weights for them. So, we can create a neuron



class with the necessary weights and the predict() function.

```python
class Neuron:
    def __init__(self):
        self.w1 = 2
        self.w2 = 0.5

    def predict(self, hungry, food_available):
        Eat = 0
        if (hungry*self.w1 + food_available*self.w2) > 1:
            Eat = 1
        return Eat

neuron = Neuron()
print(neuron.predict(1, 1))
print(neuron.predict(1, 0))
print(neuron.predict(0, 1))
print(neuron.predict(0, 0))
```

Where do we get the values of the weights? Well, that is machine learning

For now, let's set w1=2 and w2=0.5. If you run the code, you will get the following output-

```
1
1
0
0
```

We can see that there is only one mistake here- the 2nd prediction should be 0. Let's set both the weights to 3. So, w1=3 and w2=3. Now, we get this-

```
1
1
1
0
```

We have two mistakes now- 2nd and 3rd prediction. Lets set w1=0.6 and w2=0.8-

```
1
0
0
0
```

We have all the correct answers now. You can play with different other weight values and check that give correct results.

By adjusting the weights, we can adapt our prediction function to fit any input-output pattern. To achieve this, we need a `learn()` function that takes the inputs and output, then applies a process to update the weights accordingly.

Let's initialize the weights with some random values
First, we make a prediction using the random weights we initially have. Next, we calculate the error by subtracting our prediction from the actual result. Finally, we update the weights based on the error and the corresponding inputs. It's like penalizing the weights according to their contribution to

the error. Alternatively, you can think of it as distributing the error among the weights, depending on how much each one influenced the prediction.

```
import random

class Neuron:
    def __init__(self):
        self.w1 = random.random()
        self.w2 = random.random()

    def predict(self, hungry, food_available):
        Eat = 0
        if (hungry*self.w1 + food_available*self.w2) > 1:
            Eat = 1
        return Eat


    def learn(self, hungry, food_available, Eat):
        error = self.predict(hungry, food_available) - Eat
        self.w1 -= error * hungry / 100
        self.w2 -= error * food_available / 100
```

We do not want to change the weights too much at a time; we wish to take small steps towards the solution (big steps often cause divergence). So, we divide the error by 100 while updating the weights.
Now, we can add training and testing code-

```
import random

class Neuron: [...]


neuron = Neuron()

while True:
    # testing
    if (neuron.predict(1, 1) == 1 and
            neuron.predict(1, 0) == 0 and
            neuron.predict(0, 1) == 0 and
            neuron.predict(0, 0) == 0):
        break

    # training
    neuron.learn(1, 1, 1)
    neuron.learn(1, 0, 0)
    neuron.learn(0, 1, 0)
    neuron.learn(0, 0, 0)


# output
print(neuron.predict(1, 1))
print(neuron.predict(1, 0))
print(neuron.predict(0, 1))
print(neuron.predict(0, 0))
```

Make functions for training and testing-

```
import random

class Neuron: [...]


data = [[1, 1, 1],
        [1, 0, 0],
        [0, 1, 0],
        [0, 0, 0]]

def runTraining(neuron):
    for row in data:
        neuron.learn(row[0], row[1], row[2])

def runTesting(neuron):
    return [neuron.predict(row[0], row[1]) for row in data]

neuron = Neuron()
while True:
    output = runTesting(neuron)
    print(output)
    if output == [row[2] for row in data]:
        break

    runTraining(neuron)
```

Each time you run the code, the number of steps required to reach a solution will vary due to the random initialization of the weights. In some cases, you might even reach the solution in just one step.

Now, let's experiment with different outputs. Change the second output to 1 and run the code multiple times to observe the behavior.

```
data = [[1, 1, 1],
        [1, 0, 1],
        [0, 1, 0],
        [0, 0, 0]]
```

Is it taking more steps to reach the solution? Why that might be the case? Take some time, think about it. You can print the weights after each training phase and analyze the gradual change.
Now, let's set the first output to 0 and run again-

```
data = [[1, 1, 0],
        [1, 0, 1],
        [0, 1, 0],
        [0, 0, 0]]
```

Is it taking fewer steps to reach the solution? Or, even more steps? This understanding is very useful to grasp the underlying mechanism of artificial neurons.
Now, let's set all the outputs to 1-

```
data = [[1, 1, 1],
        [1, 0, 1],
        [0, 1, 1],
        [0, 0, 1]]
```

What is happening? Did your computer crash? Or, not responding? Actually, it is stuck in the while loop. Because, we never get all correct output. The problem happens for the 4th row- [0, 0, 1]. And, it is easy to understand why- the input values are both 0 (zero). So, no matter what the weights are, zero multiplied by any value is zero. Thus, the condition in the predict() function `(hungry*self.w1 + food_available*self.w2) > 1` will never satisfy.

Think about this issue we are stuck at. What would you do to solve it.

Actually, that threshold- 1 (one) in the condition is the problem here. Instead of 1, you could put a -1 (negative 1) as the threshold- `(hungry*self.w1 + food_available*self.w2) > -1`. And it will solve the problem. But then, we will have issues with all 0 (zero) output-

```
data = [[1, 1, 0],
        [1, 0, 0],
        [0, 1, 0],
        [0, 0, 0]]
```

So, we need to adjust the threshold accordingly as well. We can set a random value to the threshold (t) and learn it the same way as we learned the weights. Here is the full code-

```
import random

class Neuron:
    def __init__(self):
        self.w1 = random.random()
        self.w2 = random.random()
        self.t = random.random()
    def predict(self, hungry, food_available):
        storm = 0
        if (hungry*self.w1 + food_available*self.w2) > self.t:
            Eat = 1
        return Eat

    def learn(self, hungry, food_available, Eat):
        error = self.predict(hungry, food_available) - Eat
        self.w1 -= error * hungry / 100
        self.w2 -= error * food_available / 100
        self.t += error / 100

data = [[1, 1, 1],
        [1, 0, 1],
        [0, 1, 1],
        [0, 0, 1]]

def runTraining(neuron):
    for row in data:
        neuron.learn(row[0], row[1], row[2])

def runTesting(neuron):
    return [neuron.predict(row[0], row[1]) for row in data]

neuron = Neuron()
while True:
    output = runTesting(neuron)
    print(output)
    if output == [row[2] for row in data]:
        break

    runTraining(neuron)
```

let's get familiar with some mouthful ML terms-

Bias: We simply call threshold t as bias in ML.

Parameters: w1, w2, and t (weights and bias) are called parameters — variables that we learn through training. The act of selecting parameters is called- Parameterization.

Model: After the training, we will have good values of w1, w2, and t (weights and bias) that produce correct outputs. This set of values of the parameters is called a model.

Learning Rate: While adjusting the parameters in the learn() function, we divide the values by 100. You can also think of it as multiplying by 0.01. We call this 0.01 as learning rate; often denoted by alpha (α). Learning rate is a very important hyper-parameter (values set by humans, not learned by training)

Gradient Descent: The algorithm we ran in the learn() function to adjust the parameters (weights and biases) is called gradient descent, or more specifically- Stochastic Gradient Descent (SGD).

Regression: The whole iterative process of predicting, calculating error, and then adjusting the parameters is called regression. The variant we used here is- Linear Regression.

Epochs: Number of loops that were required in regression for reaching the correct outputs.

The above code considered only two inputs. But you can easily upgrade it for any number of inputs simply by using arrays.