LESSON 22 Neural Network Applications Last Update: June 3, 2021

In this **Artificial Neural Network** application we will make an app that prepares ingredients for a food items like hamburger, French fries and ham sandwich.

A chef in a restaurant needs to know what ingredients are used to cook a certain food Item. This is a real scenario. I once knew a lady who worked in a sandwich shop. She needed to know what ingredients went into each sandwich ordered. The way she did it was to have the sandwich menu right beside her! It would be handy to have a program where you select the food Item to prepare and then automatically lists all the ingredients and cooking instructions. A smart AI program is needed because there would be too many variations to store in a data base. Artificial Neural Networks comes to the rescue.

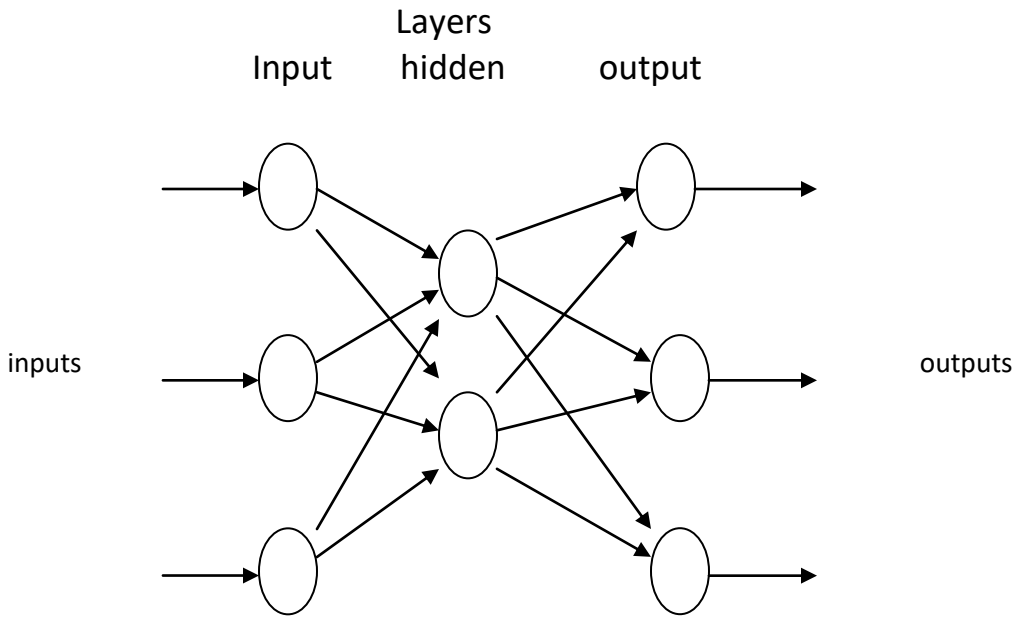
A Neural Network is software program that tries to mimic the operation of the human brain. The brain is made up of cells called neurons that store information and provide thought and visual processes. These communicate with each other by passing signals to each other.

Our Neural Network will contain software neurons. We will have input neuron to provide inputs, hidden neurons to provide processing and output neurons to provide output. The neurons are grouped together an array known as layers.

Neural Network Structure

A neural network is a group of neurons contained in a layer. A neural network may have many layers. Each layer is connected to each other. Each layer can have different number of neurons. The input layer connects to the hidden layer and the hidden layer connects to the output layers. A neural network may have many hidden layers. Usually if there are many inputs there may be additional hidden layers to provide more processing power.

Neural Network Model:



We can now apply our cooking application using a Neural Network.

Our inputs would be food items and our outputs would be the ingredients needed. For this application we will keep it simple but we could easily expand to add variations like serving sizes and cooking instructions.

We will have the following food items as input:

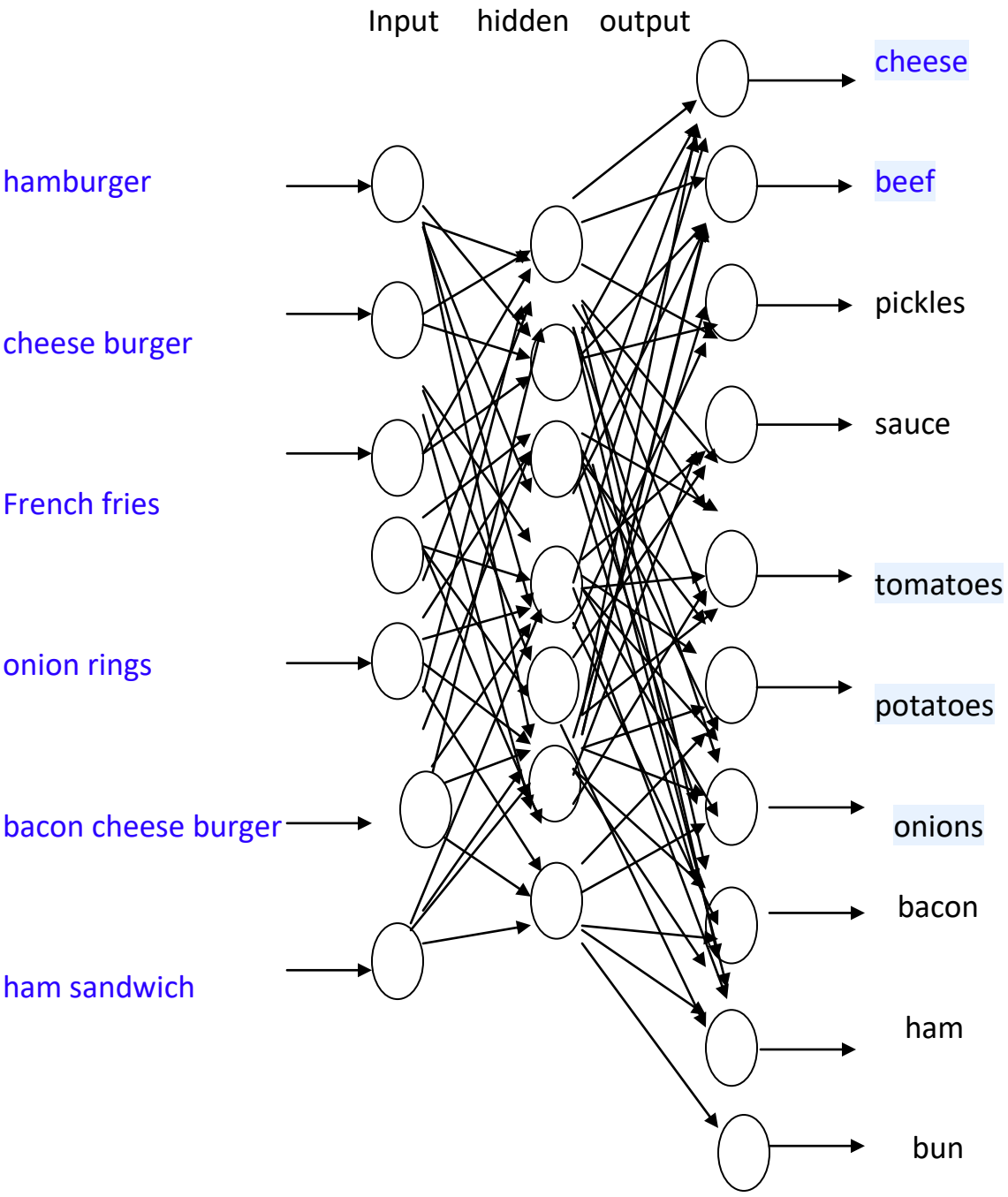
Hamburger, cheese burger, French fries, onion rings, bacon cheese burger and ham sandwich

Just to let you know bacon cheese burger is my favorite food.

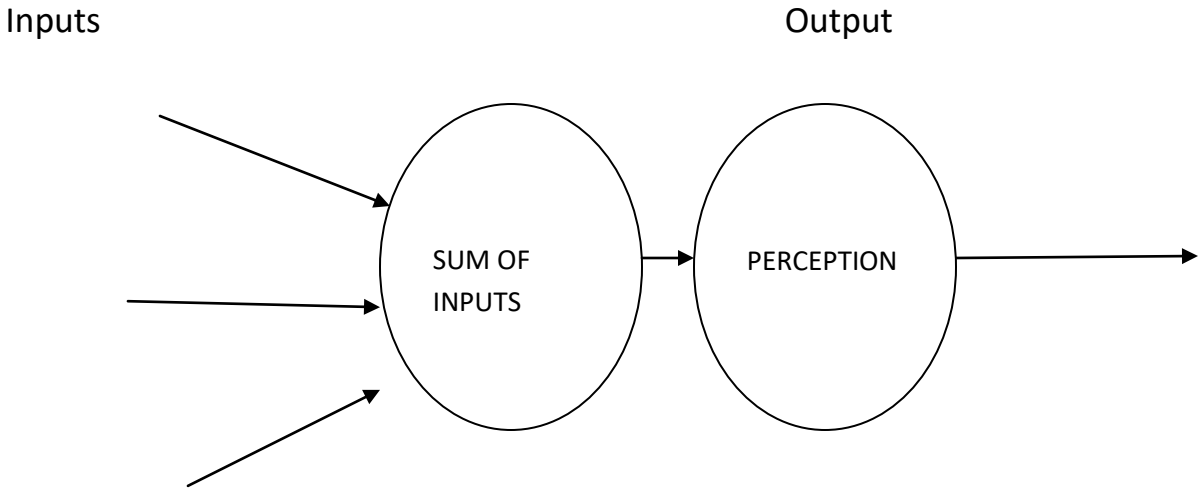
We will have the following outputs as calculated ingredients:

cheese, beef, pickles, sauce, tomatoes, potatoes, onions, bacon, ham, bun

Note: Some same ingredients are used in many different food items. We can now model our neural network with our input food items and our output ingredients;



Each neuron has a perception. The purpose of perceptions is to take the sum of the inputs and then determine if the sum of the inputs is above a certain threshold. Just like how the brain fires.



The perception fires for an activation level. The activation level is a mathematical function.

Neural Network Model

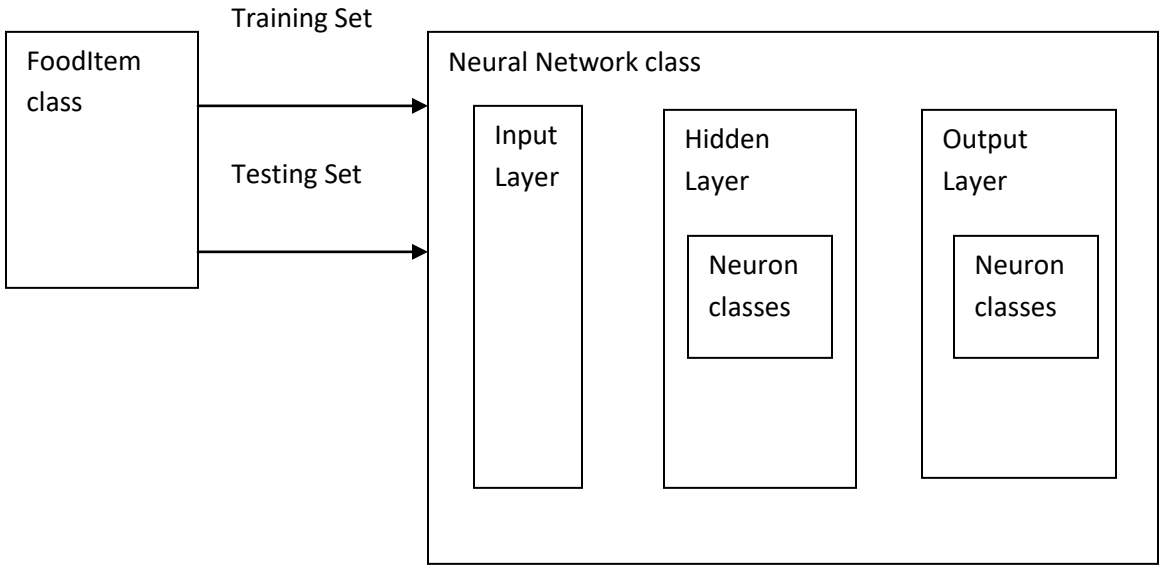
We are using object oriented programming for our neural network. Classes make it easier to program but maybe make it more difficult to understand the program flow. Object Oriented Programming provides an elegant solution to the Neural Network problem.

We have the following classes:

Class	**Purpose**
Neuron	stores inputs, weights, and output
FoodItem	store food item and ingredients
NeuralNetwork	stores all layers, trains neurons

The NeuralNetwork class will contain the input, hidden and output layers. The hidden and output layer will contain the neurons. The FoodItem class will contain the testing and training data to be applied the neural network outputs and neural network outputs.

Class Block Diagram:



Training and Testing a Neural Network

We train a neural network with known test data, and observing the output results to the expected results. An error occurs if the actual output does not match the expected output. The error is used to adjust the neural network so the outputs will match the expected output. Usually a sample subset of data is used to train a neural network.

Each neuron gets a weight. By changing the weights of the neuron we can reduce the error. At setup the weights are set to random values. A response calculation is used to sum all the inputs with the weights of each neuron.

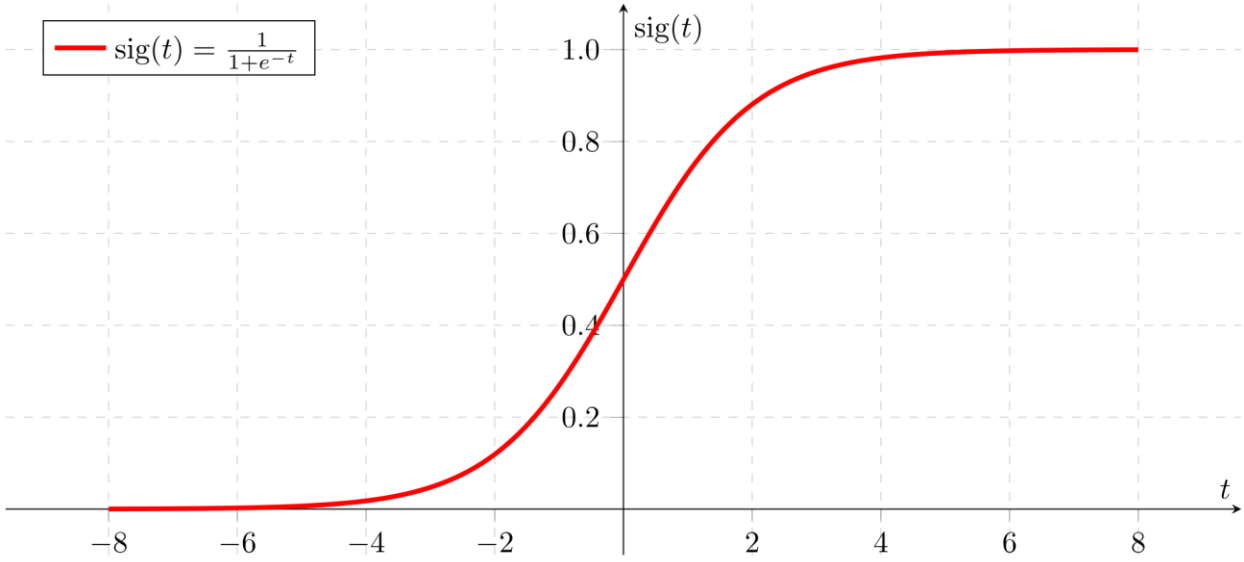
The response function is:

Input = sum of output layer * weight of neuron

The input is passed to an activation function called the Sigmoid function.

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

x is the input and y is the output, It is an exponential function that fires at higher activation levels.



We use the Neuron class to calculate the output for each input. Each Neuron class has an array of weights initialized to a random value when the neuron is first created. The length of the weights is equal to the number of inputs of the neuron.

Here are the input to output steps:

We first sum up the inputs and weights.

```
self.internal = 0.0;

# sum input and weights
for i in range(len(self.inputs)):
   self.internal += self.inputs[i].output * self.weights[i];
```

Each output perception has an activation level using the **sigmoid function**

```
# activate with sigmoid function
self.output = 2.0 / (1.0 + math.exp(-2.0 * self.internal)) - 1.0;
```

The error between actual value and expected value is used to adjust the weights of the neuron

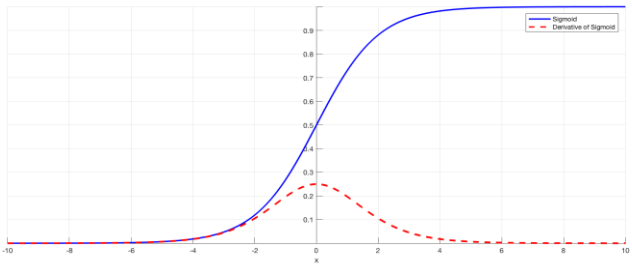
Error = expected – actual

The weights must be adjusted slowly using a **learning rate** like .01

We then differentiate our sigmoid output to get a small change in value

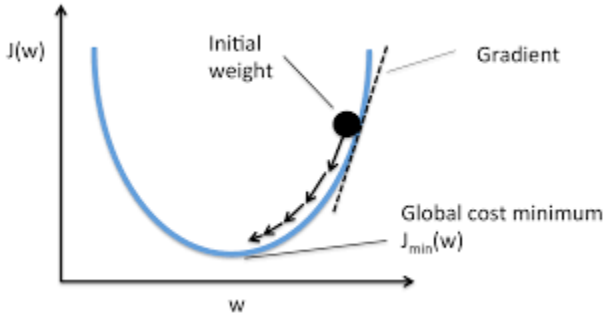
$$\text{Sigmoid}(x) = \frac{2}{1 + e\text{^}2x} - 1$$

$$\frac{d}{dx}\,\text{sigmoid}(x) = \frac{1}{1 + e^{-2x}} = \frac{e^{-x}}{(1 + e^{-x})^2} = \text{sig}(x)\,(1 - \text{sig}\,)x))$$



Gradient Descent

We use Gradient Descent to find the minimum error. You can also think of a gradient as the slope of a function. The higher the gradient, the steeper the slope and the faster a model can learn. But if the slope is zero, the model stops learning. In mathematical terms, a gradient is a partial derivative with respect to its inputs.



We find the minimum error by adjusting the weights by the derivative of the output function times error times learning rate

delta = (1.0 - output) * (1.0 + output) * error * learning rate

We then sum all the input to adjust the weights:

```
# calculate derivative of output
delta =(1.0 - self.output) * (1.0 + self.output) * self.error * self.learning_rate;

# update neuron error and weights
for i in range(len(self.inputs)):
   self.inputs[i].error += self.weights[i] * self.error;
   self.weights[i] += self.inputs[i].output * delta;
```

Here is our complete Neuron class that is use to calculate the outputs.

```
"""
Neuron class stores inputs, weights, and output
"""
class Neuron:

   # initialize neuron
   def __init__(self,inputs, learning_rate=.01):
      self.inputs=[None]*len(inputs)
      self.weights=[0]*len(inputs)
      self.internal=0.0
      self.output=0.0
      self.learning_rate = learning_rate
      self.error=0.0

      # store inputs and calculate random weights
      for i in range(len(self.inputs)):
         self.inputs[i] = inputs[i];
         self.weights[i] = (random.random()*2) - 1; # +/- 1

   # calculate neuron output
   def respond(self):
      self.internal = 0.0;

      # sum input and weights
      for i in range(len(self.inputs)):
         self.internal += self.inputs[i].output * self.weights[i];

      # activate with sigmoid function
      self.output = 2.0 / (1.0 + math.exp(-2.0 * self.internal)) - 1.0;
      self.error = 0.0;

   # calculate neuron error
   def setError(self,desired):
      self.error = desired - self.output;
```

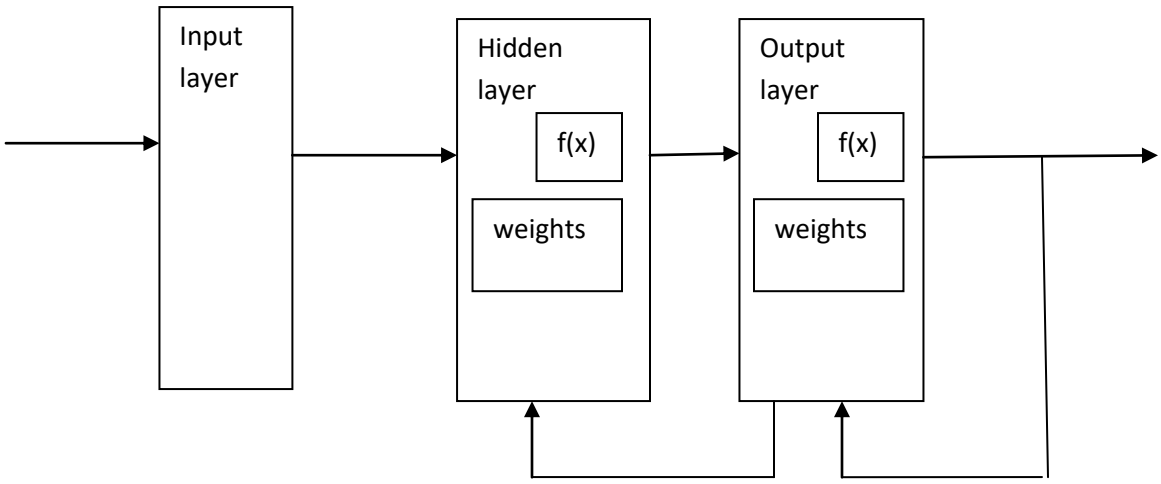
```
# train neuron
   def train(self):

      # calculate derivative of output
      delta =(1.0 - self.output) * (1.0 + self.output) * self.error * self.learning_rate;

      # update neuron error and weights
      for i in range(len(self.inputs)):
         self.inputs[i].error += self.weights[i] * self.error;
         self.weights[i] += self.inputs[i].output * delta;
```

Neural Network class

Our Neural network Class consists of layers each layer consists of neurons having input weights and output activation level.



Feed Forward Propagation

The output of the input layer is just the input to the input layer that are stored in its output . The hidden layer output is calculated by summing the inputs and the weights of the hidden layer. The output layer is the sum on the hidden layer output times the weights of the output layer. This is known as **feed forward propagation.**

```
# apply inputs to input lawyer
      for i in range(len(self.input_layer)):
         self.input_layer[i].output = item.inputs[i];

      # activate hidden layer
      for i in range(len(self.hidden_layer)):
         self.hidden_layer[i].respond();

      # activate output layer
      for i in range(len(self.output_layer)):
         self.output_layer[i].respond();

      # store output responses
      for i in range(len(self.output_layer)):
          self.responses[i] = self.output_layer[i].output;
```

Back Propagation

Each neuron feds back values to its inputs. For updating the weights.

We first calculate the error so we can adjust the weights

```
# propogate output layer
      for i in range(len(self.output_layer)):
        self.output_layer[i].setError(outputs[i]);
        totalError += self.output_layer[i].error
        self.output_layer[i].train();
```

then we propogate back to the hidden layer.

```
      #propagate back to the hidden layer
      for i in range(len(self.hidden_layer)):
         self.hidden_layer[i].train();
```

Our NeuralNetwok class stores the inputs, hidden and output layers and learning rate. Our NeuralNetwok uses the Neuron class to do forward and backward propagation.

Here is our complete Neural Network class

```
"""
Neural Network class stores all layers
trains neurons
"""
class NeuralNetwork:

   # initialize network
   def __init__(self,inputs, hidden, outputs, learning_rate = .01):

      self.responses = [None]*outputs; # output responses
      self.input_layer = [None]*inputs
      self.hidden_layer = [None]*hidden
      self.output_layer = [None]*outputs
      self.learning_rate = learning_rate

      # make input layer
      for i in range(len(self.input_layer)):
         self.input_layer[i] = Neuron([]);

      # make hidden layer
      for i in range(len(self.hidden_layer)):
         self.hidden_layer[i] = Neuron(self.input_layer,learning_rate);

      # make output layer
      for i in range(len(self.output_layer)):
         self.output_layer[i] = Neuron(self.hidden_layer,learning_rate);

   # activate all layers
   def respond(self,item):

      # apply inputs to input lawyer
      for i in range(len(self.input_layer)):
         self.input_layer[i].output = item.inputs[i];

      # activate hidden layer
      for i in range(len(self.hidden_layer)):
         self.hidden_layer[i].respond();
```

```
    # activate output layer
    for i in range(len(self.output_layer)):
        self.output_layer[i].respond();

    # store output responses
    for i in range(len(self.output_layer)):
         self.responses[i] = self.output_layer[i].output;

# train all layers
# return totalError
def train(self,outputs):

    # Back Propagation
    # Each neuron feeds back values to its inputs, for updating the weights
    totalError = 0

    # propogate output layer
    for i in range(len(self.output_layer)):
      self.output_layer[i].setError(outputs[i]);
      totalError += self.output_layer[i].error
      self.output_layer[i].train();

    #propagate back to the hidden layer
    for i in range(len(self.hidden_layer)):
        self.hidden_layer[i].train();

    return totalError

# display layers
def display(self):
    #Draw the input layer
    print("Input Layer output:")
    for i in range(len(self.input_layer)):
        print(self.input_layer[i].output,end=" ")
    print()

    #Draw the hidden layer
    print("Hidden Layer output:")
    for i in range(len(self.hidden_layer)):
        print(self.hidden_layer[i].output,end=" ")
    print()
```

```
    #Draw the output layer
    print("Output Layer output:")
    for i in range(len(self.output_layer)):
       print(self.output_layer[i].output,end=" ")
    print()
```

FoodItem class

We have a FoodItem class to store the inputs and outputs data items.

```
"""
FoodItem class to store food item and ingredients
"""
class FoodItem:

  # initialize food item
  def __init__(self):
   self.inputs=[];  # food item
   self.outputs=[];  # ingredients

  # load food items
  def foodItemLoad(self,foodItems):

    for i in range(len(foodItems)):

      if(foodItems[i] == 1):
        self.inputs.append(1.0);

      else:
        self.inputs.append(-1.0);

  # load ingredients
  def ingredientsLoad(self,ingredients):

    for i in range(len(ingredients)):
      if ingredients[i] == 1:
        self.outputs.append(1.0);
      else:
        self.outputs.append(-1.0);
```

Main Program

Our main program must setup and train the neural network

We will have 2 sets a training set and a testing set. The training set is usually a subset of the testing set. In our case they both could be the same since we want accurate results.

Before we can start training we need some training data.

Our main program must setup and train the neural network
We will have 2 sets a training set and a testing set.
The training set is usually a subset of the testing set.
In our case they both could be the same since we want accurate results.

Before we can start training we need some training data.
We make an array of expected inputs.

```
foodItemsNames = ["hamburger","cheese burger","french fries","onion rings",
        "bacon cheese burger","ham sandwhich","","","","",""];

# some food items
foodItems = [
        [1,0,0,0,0,0,0,0,0,0], # hamburger
        [0,1,0,0,0,0,0,0,0,0], # cheese burger
        [0,0,1,0,0,0,0,0,0,0], # french fries
        [0,0,0,1,0,0,0,0,0,0], # onion rings
        [0,0,0,0,1,0,0,0,0,0], # bacon cheese burger
        [0,0,0,0,0,1,0,0,0,0] # ham sandwhich
        ];

# ingredients for food  items
i ngredientsNames = ["cheese","beef","pickle","sauce","tomatoe","potatoes","onion","bacon","ham", "bun"];
```

Our main program is quite simple
It first sets ups the network then trains the network the tests the network.
We will have 10 input and 12 hidden nodes and 10 output nodes with a learning rate of .01. For training we will thrive for .1 accuracy.

```
# setup
print("restaurant app")
testing_set,training_set = loadData();
learning_rate = .01
accuracy = .1
neuralnet = NewuralNetwork(10, 12, 10,learning_rate);
print("training:")
train(accuracy);
print("testing")
test();
```

Training and Testing

For Training we give the inputs and use the actual output to train the neural network into an acceptable accuracy of lets say .1.

For testing we check the predicted output match the actual outputs and print the actual and predicted ingredients along with an accuracy score.

Like this:

Training

```
Total Error: -0.10014161259707999
total Error: -0.1001152255129153
total Error: -0.1000888558474019
total Error: -0.10006250357321644
total Error: -0.10003616866318632
total Error: -0.10000985109028171
total Error: -0.09998355082762822
testing
test # 1 :
food item: hamburger
Expected: beef pickle sauce tomatoe potatoes onion
Actual: beef pickle sauce tomatoe potatoes onion
score:  0.4275528941262542

test # 2 :
food item: cheese burger
Expected: cheese beef pickle sauce tomatoe potatoes onion
Actual: cheese beef pickle sauce tomatoe potatoes onion
score:  0.46479941805847896
```

test # 3 :
food item: french fries
Expected: pickle potatoes
Actual: pickle potatoes
score: 0.45944934997766623

test # 4 :
food item: onion rings
Expected: sauce onion
Actual: sauce onion
score: 0.3366653416154086

test # 5 :
food item: bacon cheese burger
Expected: cheese beef pickle sauce tomatoe bacon
Actual: cheese beef pickle sauce tomatoe bacon
score: 0.3971925718792292

test # 6 :
food item: ham sandwhich
Expected: onion ham bun
Actual: onion ham bun
score: 0.4294568442270051

6 out of 6

We get very accurate results.

Here is the complete main code:

```
"""
main
"""
# Our main program must setup and train the neural network
# We will have 2 sets a training set and a testing set.
# The training set is usually a subset of the testing set.
# In our case they both could be the same since we want accurate results.

# Before we can start training we need some training data.
# We make an array of  expected inputs.
foodItemsNames = ["hamburger","cheese burger","french fries","onion rings",
        "bacon cheese burger","ham sandwhich","","","","",""];
```

```
# some food items
foodItems = [
        [1,0,0,0,0,0,0,0,0,0], # hamburger
        [0,1,0,0,0,0,0,0,0,0], # cheese burger
        [0,0,1,0,0,0,0,0,0,0], # french fries
        [0,0,0,1,0,0,0,0,0,0], # onion rings
        [0,0,0,0,1,0,0,0,0,0], # bacon cheese burger
        [0,0,0,0,0,1,0,0,0,0] # ham sandwhich
        ];

# ingredients for food  items
ingredientsNames = ["cheese", "beef","pickle","sauce","tomatoe","potatoes","onion","bacon","ham", "bun"];

# make ingredients for foods
# cheese, beef,pickle,sauce,tomatoe,potatoes,onion,bacon,ham, bun
ingredients = [
        [0,1,1,1,1,1,1,0,0,0], # hamburger
        [1,1,1,1,1,1,1,0,0,0], # cheese burger
        [0,0,1,0,0,1,0,0,0,0], # french fries
        [0,0,0,1,0,0,1,0,0,0], # onion rings
        [1,1,1,1,1,0,0,1,0,0],  # bacon cheese burger
        [0,0,0,0,0,0,1,0,1,1] # ham sandwhich
        ]


# main first set ups, train and tests the network

# load food items and ingredients
def loadData():

   training_set = [None]*len(foodItems)
   testing_set = [None]*len(foodItems)

   # load training set
   for i in range(len(training_set)):

      training_set[i] = FoodItem();
      training_set[i].foodItemLoad(foodItems[i]);
      training_set[i].ingredientsLoad(ingredients[i]);

   # load testing set
   for i in range (len(testing_set)):
      testing_set[i] = FoodItem();
      testing_set[i].foodItemLoad(foodItems[i]);
      testing_set[i].ingredientsLoad(ingredients[i]);

   return training_set,testing_set
```

```
# train neural network
# train till accuracy met
def train(accuracy):

   iterations = 0

   while(True):
      totalError = 0
      for i in  range(len(training_set)):
         #print("training set input")
         #print(training_set[i].inputs)
         neuralnet.respond(training_set[i]);
         #print("before traiin")
         #neuralnet.display();
         error = neuralnet.train(training_set[i].outputs);
         #print("error: ",error)
         #print("after train")
         #neuralnet.display();
         #print("training set output")
         #print(training_set[i].outputs)
         totalError += error
      print("total Error:",totalError)
      if abs(totalError) < accuracy: break
      iterations+=1
   return iterations

# print results
def results(index, responses):

   print("food item:",foodItemsNames[index]);

   print("Expected:",end=" ");
   for i in range(len(ingredientsNames)):

      if testing_set[index].outputs[i]==1:
         print(ingredientsNames[i], end=" ");
   print();

   print("Actual:",end=" ");
   for i in range(len(ingredientsNames)):
      if responses[i] >= 0:
         print(ingredientsNames[i], end=" ");

   print()
   print("score: ",score(responses,testing_set[index].outputs))
   print()
```

```
# test
def test():

   totalRight = 0
   totalTest = 0

   for i in range(len(testing_set)):

      print("test #",(i+1),":")

      neuralnet.respond(testing_set[i]);
      #neuralnet.display();

      results(i,neuralnet.responses);

      if match(neuralnet.responses,testing_set[i].outputs):
         totalRight+=1;

      totalTest+=1;


   print(totalRight," out of ",totalTest) # "accuracy", score(neuralnet.responses,testing_set))


# We need to compare all outputs to the training set.

# return best Score
#return  true if match 50%
def match(expected, actual):

   count = 0;
   for i in range(len(expected)):
      if(abs(expected[i]-actual[i])>0):
         count+=1
   return count==len(expected)

# return sum of error
def score(expected, actual):

   sumerror = 0;

   for i in range(len(expected)):
      sumerror += abs(expected[i]-actual[i]);

   sumerror = math.sqrt(sumerror);
   return sumerror;
```

```
 # setup
print("restaurant app")
testing_set,training_set = loadData();
learning_rate = .01
accuracy = .1
neuralnet = NeuralNetwork(10, 12, 10,learning_rate);
print("training:")
train(accuracy);
print("testing")
test();
```

todo:

Type in or copy and paste all the above code and get it running.
Try different learning rate and accuracy to imp prove performance.
Try adding more food items with their needed ingredients

NEURAL NETWORK APPLICATIONS HOMEWORK

Reverse the app, give it the ingredients as inputs and check if it can guess the food item.

Call your py file neuralNetworkApplicationsHomework.py

END