simplenn.py

```
import copy
import random
import json

def sigmoid(n):
    """
    Given a number, n, return the sigmoid() of that number
    """
    e = 2.7182818284
    n = float(n)
    try:
        return 1.0/(1.0 + e**(-n))
    except OverflowError:
        if n > 0:
            return 0.99999
        else:
            return -0.99999

def sigmoid_derivative(n):
    """
    Given a number, n, return the gradient (derivative) of the sigmoid curve at position n
    """
    n = float(n)
    try:
        return sigmoid(n)*(1.0-sigmoid(n))
    except OverflowError:
        return 0

# this file continues...
```

```
class Matrix:
    def __init__(self, rows, cols):
        """ Create a 2d array rows x cols populated with zeros """
        self.rows = rows
        self.cols = cols
        self.data = []
        for row in range(rows):
            row_data = []
            for _ in range(cols):
                row_data.append(0)
            self.data.append(row_data)

    def randomise(self):
        """ Fill the 2d array with random floating numbers between 0 and 1 """
        for row in range(len(self.data)):
            for col in range(len(self.data[row])):
                self.data[row][col] = random.random()

    @staticmethod
    def subtract(a, b):
        """
        Where a and b are 2d matrices,
        On each cell, calculate a minus b
        Return a new 2d array with the result
        """
        result = Matrix(a.rows, a.cols)
        for row in range(a.rows):
            for col in range(a.cols):
                result.data[row][col] = a.data[row][col] - b.data[row][col]
        return result

    def add(self, n):
        """
        If n is a Matrix,
            iterate through the array,
            add the relevant cell position to the value of the matching cell in this array
        If n is a number,
            add that number to every cell in the array
        """
        if isinstance(n, Matrix):
            for row in range(len(self.data)):
                for col in range(len(self.data[row])):
                    self.data[row][col] += n.data[row][col]
        elif isinstance(n, float):
            for row in range(len(self.data)):
                for col in range(len(self.data[row])):
                    self.data[row][col] += n
        else:
            raise AssertionError("Invalid types for addition")

# this file continues...
```

```
    @staticmethod
    def transpose(matrix):
        """
        Returns a transposed (rotated) matrix
        Example, this...
            [ [ 1  2  3]
              [11 12 13]
              [21 22 23]
              [31 32 33] ]
        becomes this...
            [ [ 1 11 21 31]
              [ 2 12 22 32]
              [ 3 13 23 33] ]
        """
        new_matrix = Matrix(matrix.cols, matrix.rows)
        for row in range(len(matrix.data)):
            for col in range(len(matrix.data[row])):
                new_matrix.data[col][row] = matrix.data[row][col]
        return new_matrix

    @staticmethod
    def dot_product(a, b):
        """
        Return a dot product of the two 2D arrays, a and b
        All the cells in a row of `a` are multiplied against all the cells in a column of `b`
        and summed together. The resulting number is placed in a cell where the respective
        row of `a` and column of `b` would intersect.
        Example if a = [ [ 0, 4,-2],
                         [-4,-3, 0] ]
               and b = [ [ 0, 1],
                         [ 1,-1],
                         [ 2, 3] ]
        Then result  = [ [ ( 0*0 +  4*1 + -2*2), ( 0*1 +  4*-1 + -2*3) ],
                         [ (-4*0 + -3*1 +  0*2), (-4*1 + -3*-1 +  0*3) ] ]
                     = [ [ 0 , -10 ],
                         [ -3,  -1 ] ]
        """
        assert len(a.data[0]) == len(b.data), "Dimensions invalid! a.rows must equal b.columns"
        result = Matrix(a.rows, b.cols)
        for row in range(a.rows):
            for col in range(b.cols):
                # calculate value for cell result[row][col]
                sum = 0
                # for each cell in the row a[row], multiply it by the cell in column b[col]
                items = len(a.data[row])
                for i in range(items):
                    sum = sum + a.data[row][i] * b.data[i][col]
                result.data[row][col] = sum
        return result

# this file continues...
```

```
    def multiply(self, n):
        """ Multiply every cell by the value of n """
        if isinstance(n, Matrix): # Hadamard product
            for row in range(len(self.data)):
                for col in range(len(self.data[row])):
                    self.data[row][col] = self.data[row][col] * n.data[row][col]
        elif isinstance(n, float):
            for row in range(len(self.data)):
                for col in range(len(self.data[row])):
                    self.data[row][col] = self.data[row][col] * n
        else:
            raise AssertionError("Invalid type to multiply")

    def map(self, f):
        """ Apply a function to every element of the 2D array """
        for row in range(len(self.data)):
            for col in range(len(self.data[0])):
                self.data[row][col] = f(self.data[row][col])

    @staticmethod
    def from_array(a):
        assert isinstance(a, list), "a must be a list of lists"
        m = Matrix(len(a), 1)
        for row in range(len(a)):
            m.data[row][0] = a[row]
        return m

# END OF CLASS Matrix
# this file continues...
```

```
class NeuralNetwork:
    def __init__(self, input_nodes, hidden_nodes, output_nodes, randomise_bias=False):
        self.input_nodes = input_nodes
        self.hidden_nodes = hidden_nodes
        self.output_nodes = output_nodes
        self.weights_i2h = Matrix(self.hidden_nodes, self.input_nodes)
        self.weights_h2o = Matrix(self.output_nodes, self.hidden_nodes)
        self.weights_i2h.randomise()
        self.weights_h2o.randomise()
        self.bias_h = Matrix(self.hidden_nodes, 1)
        self.bias_o = Matrix(self.output_nodes, 1)
        if randomise_bias:
            self.bias_h.randomise()
            self.bias_o.randomise()
        self.set_learning_rate()

    def set_activation_functions(self, activation, activation_derivative):
        self.activation_function = activation
        self.activation_function_derivative = activation_derivative

    def set_learning_rate(self, learning_rate = 0.1):
        self.learning_rate = learning_rate

    def serialise(self):
        export = {}
        export["input_nodes"] = self.input_nodes
        export["hidden_nodes"] = self.hidden_nodes
        export["output_nodes"] = self.output_nodes
        export["learning_rate"] = self.learning_rate
        export["i2h"] = self.weights_i2h.data
        export["h2o"] = self.weights_h2o.data
        export["bias_h"] = self.bias_h.data
        export["bias_o"] = self.bias_o.data
        return json.dumps(export, indent=3)

    @staticmethod
    def deserialise(data):
        ob = json.loads(data)
        nn = NeuralNetwork(ob["input_nodes"], ob["hidden_nodes"], ob["output_nodes"])
        nn.set_learning_rate(ob["learning_rate"])
        nn.weights_i2h.data = ob["i2h"]
        nn.weights_h2o.data = ob["h2o"]
        nn.bias_h.data = ob["bias_h"]
        nn.bias_o.data = ob["bias_o"]
        return nn

# this file continues...
```

```
    def predict(self, input_array):
        ### Setup
        inputs = Matrix.from_array(input_array)                # Convert to matrix

        ### Hidden layer nodes
        hidden = Matrix.dot_product(self.weights_i2h, inputs)  # Find raw values
        hidden.add(self.bias_h)                                # Add bias to nodes
        hidden.map(self.activation_function)                   # Apply activation function

        ### Output layer nodes
        outputs = Matrix.dot_product(self.weights_h2o, hidden) # Find raw values
        outputs.add(self.bias_o)                               # Apply bias
        outputs.map(self.activation_function)                  # Apply activation function
        return outputs

    def train(self, input_array, target_array):
        ### Setup
        inputs = Matrix.from_array(input_array)                    # Convert to matrix
        targets = Matrix.from_array(target_array)                  # Convert to matrix

        ### Hidden layer nodes
        hidden = Matrix.dot_product(self.weights_i2h, inputs)      # Find raw values
        hidden.add(self.bias_h)                                    # Add bias to nodes
        hidden.map(self.activation_function)                       # Apply activation function

        ### Output layer nodes
        outputs = Matrix.dot_product(self.weights_h2o, hidden)     # Find raw values
        outputs.add(self.bias_o)                                   # Add bias to nodes
        outputs.map(self.activation_function)                      # Apply activation function

        ### Adjustments for hidden->output
        output_errors = Matrix.subtract(targets, outputs)
        adjustments_h2o = copy.deepcopy(outputs)      # Make a copy of nodes
        adjustments_h2o.map(self.activation_function_derivative)   # Apply derivative function
        adjustments_h2o.multiply(output_errors)       # Moderate errors by derivatives
        adjustments_h2o.multiply(self.learning_rate) # Moderate by learning rate
        hidden_t = Matrix.transpose(hidden)           # Calculate delta for individual weights
        weight_h2o_deltas = Matrix.dot_product(adjustments_h2o, hidden_t)
        self.weights_h2o.add(weight_h2o_deltas)       # Adjust weights by their delta
        self.bias_o.add(adjustments_h2o)              # Adjust bias

        ### Adjustments for input->hidden
        weight_h2o_t = Matrix.transpose(self.weights_h2o)
        hidden_errors = Matrix.dot_product(weight_h2o_t, output_errors)
        adjustments_i2h = copy.deepcopy(hidden)       # Make a copy of nodes
        adjustments_i2h.map(self.activation_function_derivative)   # Apply derivative function
        adjustments_i2h.multiply(hidden_errors)       # Moderate hidden errors with derivatives
        adjustments_i2h.multiply(self.learning_rate) # Moderate again by learning rate
        inputs_t = Matrix.transpose(inputs)           # Calculate delta for individual weights
        weight_i2h_deltas = Matrix.dot_product(adjustments_i2h, inputs_t)
        self.weights_i2h.add(weight_i2h_deltas)       # Adjust weights by their delta
        self.bias_h.add(adjustments_i2h)              # Adjust bias

# END OF CLASS NeuralNetwork
# END OF FILE simplenn.py
```

demo.py

```
###
### DEMO TO TEST OUR NEURAL NETWORK
###

import random
import simplenn
import logging

### Seed the random number generator
random.seed()

### Define training data
# The network is a lot more reliable if we give it two output nodes.
# So, instead of just one target to indicate 1=TRUE, 0=FALSE, we will use two output nodes:
# --> one to indicate the network is predicting TRUE,
# --> the other to indicate the network is predicting FALSE
training_data_inputs = [ [0.0, 0.0], [0.0, 1.0], [1.0, 0.0], [1.0, 1.0] ]
training_data_target = [ [0.0, 1.0], [1.0, 0.0], [1.0, 0.0], [0.0, 1.0] ]

### Setup the network
nn = simplenn.NeuralNetwork(2,6,2)
nn.set_activation_functions(simplenn.sigmoid, simplenn.sigmoid_derivative)
nn.set_learning_rate(0.1)

### Train the network
for i in range(200000):
    r = random.randint(0, 3)
    if (i % 1000 == 0):
        print("Training run {}".format(i))
    nn.train(training_data_inputs[r], training_data_target[r])

### Make some predictions
result = nn.predict( [0.0, 0.0] )
print(result.data)
result = nn.predict( [0.0, 1.0] )
print(result.data)
result = nn.predict( [1.0, 0.0] )
print(result.data)
result = nn.predict( [1.0, 1.0] )
print(result.data)
```