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)