Definite Integral using ANN

Integration is one of the important and challenging problems for mathematician. Integration can be broadly divided into two types based on the limit provided or not:

1. Definite integration: This type of integration have limit bound i.e., upper and lower limit
2. Indefinite integration: This type of integration does not have limit bound.

Basically, definite integration gives area between their limit bound. For solving the definite integration, some methods are listed below:

1. The chaotic firefly algorithm (evolutionary computing)
2. Analytic methods

Analytic methods: In these methods, we get the approximate value for a given definite integration. There are numerous methods are listed below for calculation of definite integration:

1. The hyperbolic tan approximation
2. The midpoint approximation

1. The area approximation: in this method, we divide the curve into small pieces, and draw trapezoidal if adjacent points of vertical axis are not equal, else square as shown in fig. 1.

Vertical axis

Horizontal axis

Fig. 1. Area approximation

Definite Integration calculation using ANN method

ANN methods: we make the Artificial Neural Network (ANN) for solving the problem of definite integration. ANN is very power full techniques for solving the non-linear problems. For ANN, we use one input neurons, one output neurons and two hidden layer with seven neurons. Forward propagation is used to calculate the actual output of the network, while back propagation is used for the learning purpose. During back propagation, weight updates according to the error generated. Error is defined as difference of actual output and desired output. We use linear activation function during forward propagation, while their derivatives during back propagation.

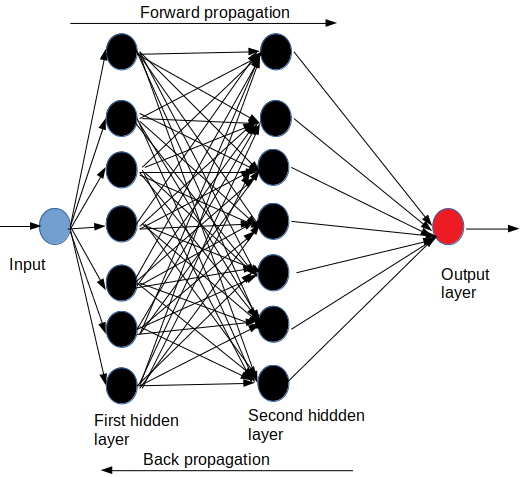


Fig. 2 Design method

Table 1: Comparison of definite integral value over limit [0, 1] using deferent methods

|  |  |  |  |
| --- | --- | --- | --- |
| Function | 2X | X | ex |
| Exact value | 1 | 0.5 | 1.72 |
| Trapezoidal | 1.0 | 0.5 | 1.72 |
| Midpoints | 1.00 | 0.5 | 1.67 |
| Tanh | 0.9966 | 0.499 | 1.71 |
| Chaotic firefly algorithm | 0.89 | 0.89 | 0.885 |
| ANN | 1.59 | 0.79 | 2.95 |

Accuracy of the ANN: We use datasets, i.e., only for function and where “a” is real number. The accuracy of the ANN is very low, but it will increase when we use larger datasets.

Steps for calculation the definite integration using ANN:

1. Create the desired function.
2. Generate the datasets for the generated function.
3. Doing normalization of the datasets.
4. Train neural network with generated datasets.
5. Predicting the result of definite integration.

Python code:

This part contains the source code for the artificial neural network and analytical methods those are explained above, i.e., midpoint, tanh and area method.

ANN methods have several parts those codes are below:

1. **ANN method**
2. Basic mathematical tools: my\_math.py

from equation import \*

import random

def mean(lst):

p = 0.0

for i in lst:

p += i

return p/len(lst)

def max(lst):

p = [-100000000000000000000000]

for i in lst:

if i>p[0]:

p[0] = i

return p[0]

def min(lst):

p = [1000000000000000000000000]

for i in lst:

if i<p[0]:

p[0] = i

return p[0]

def frange(start, stop=None, step=None):

# if stop and step argument is None set start=0.0 and step = 1.0

start = float(start)

if stop == None:

stop = start + 0.0

start = 0.0

if step == None:

step = 1.0

# print("start= ", start, "stop= ", stop, "step= ", step)

count = 0

while True:

temp = float(start + count \* step)

if step > 0 and temp >= stop:

break

elif step < 0 and temp <= stop:

break

yield temp

count += 1

def datasets():

'''

this function is used for the generating mathematical fucntion for definiteintegral

'''

x = equation('qua',2)

integrand = x.equation()

integral = x.eq\_int()

train\_x = []

train\_y = []

m = -10

n = 10

for i in frange(m, n,1):

# v\_integrand = integrand(i)

# p = (i-m)/(n-m)

p = sigmoid(i)

# print (i,p)

v\_integral = integral(i)

# q = (v\_integral-0)/(integral(n)-0)

q = sigmoid(v\_integral)

train\_x.append([p])

train\_y.append([q])

return train\_x,train\_y

def fact(n):

if n==0:

return 1

return n\*fact(n-1)

def exp(x):

y = 0

for i in range(10):

t = (x\*\*i)/fact(i)

y += t

return y

def sin(x):

result = 0

for i in rnage(1,100):

t = ((-1)\*\*i+1)\*(x\*\*(2\*i-1))/fact(2\*i-1)

result += t

return result

def cos(x):

result = 1

for i in range(1,100):

t = ((-1\*\*i)\*(x\*\*2\*i))/fact(2\*i)

result += t

return result

def mid\_point(a,b):

return (a+b)/2

def linear(x):

return 1\*x

def sigmoid(x):

return 1/(1+exp(-x))

def derivatives\_sigmoid(x):

return x\*(1-x)

def tanh(x):

return (exp(2\*x)-1)/(exp(2\*x)+1)

def rand(a, b):

return (b-a)\*random.random() + a

def makeMatrix(I, J, fill=0.0):

m = []

for i in range(I):

m.append([fill]\*J)

return m

1. Equation formation: equation.py

from my\_math import \*

class equation:

def \_\_init\_\_(self,type,a,b=0,c=0):

self.type = type

self.a = a

self.b = b

self.c = c

def equation(self):

if self.type == 'lin':

y = lambda x: self.a\*x+self.b

elif self.type == 't\_sin':

y = lambda x:self.a\*sin(x)

elif self.type == 'qua':

y = lambda x: self.a\*(x\*\*2)+self.b\*x+self.c

elif self.type == 'exp':

y = lambda x: exp(self.a\*x)

elif self.type == 'tri':

y = lambda x: self.a\*(x\*\*3)

elif self.type == 'for':

y = lambda x: self.a\*(x\*\*4)

elif self.type == 'fiv':

y = lambda x: self.a\*(x\*\*5)

elif self.type == 'six':

y = lambda x: self.a\*(x\*\*6)

elif self.type == 'sev':

y = lambda x: self.a\*(x\*\*7)

elif self.type == 'eig':

y = lambda x: self.a\*(x\*\*8)

elif self.type == 'nin':

y = lambda x: self.a\*(x\*\*9)

elif self.type == 'ten':

y = lambda x: self.a\*(x\*\*10)

elif self.type == 'ele':

y = lambda x: self.a\*(x\*\*11)

elif self.type == 'twe':

y = lambda x: self.a\*(x\*\*12)

elif self.type == 'thir':

y = lambda x: self.a\*(x\*\*13)

elif self.type == 'fourt':

y = lambda x: self.a\*(x\*\*14)

elif self.type == 'fift':

y = lambda x: self.a\*(x\*\*15)

elif self.type == 'sixt':

y = lambda x: self.a\*(x\*\*16)

elif self.type == 'sevet':

y = lambda x: self.a\*(x\*\*17)

elif self.type == 'eight':

y = lambda x: self.a\*(x\*\*18)

elif self.type == 'nint':

y = lambda x: self.a\*(x\*\*19)

elif self.type == 'twen':

y = lambda x: self.a\*(x\*\*20)

elif self.type == 'tweno':

y = lambda x: self.a\*(x\*\*21)

elif self.type == 'twent':

y = lambda x: self.a\*(x\*\*22)

elif self.type == 'twenth':

y = lambda x: self.a\*(x\*\*23)

elif self.type == 'twenf':

y = lambda x: self.a\*(x\*\*24)

elif self.type == 'twenfi':

y = lambda x: self.a\*(x\*\*25)

elif self.type == 'twensi':

y = lambda x: self.a\*(x\*\*26)

elif self.type == 'twense':

y = lambda x: self.a\*(x\*\*27)

elif self.type == 'twenei':

y = lambda x: self.a\*(x\*\*28)

return y

def eq\_int(self):

if self.type == 'lin':

y = lambda x: (self.a\*(x\*\*2))/2+self.b\*x

elif self.type == 't\_sin':

y = lambda x:self.a\*cos(x)

elif self.type == 'qua':

y = lambda x: self.a\*(x\*\*3)/3+self.b\*(x\*\*2)/2+self.c\*x

elif self.type == 'exp':

y = lambda x: self.a\*exp(self.a\*x)

elif self.type == 'tri':

y = lambda x: self.a\*(x\*\*4)/4

elif self.type == 'for':

y = lambda x: self.a\*(x\*\*5)/5

elif self.type == 'fiv':

y = lambda x: self.a\*(x\*\*6)/6

elif self.type == 'six':

y = lambda x: self.a\*(x\*\*7)/7

elif self.type == 'sev':

y = lambda x: self.a\*(x\*\*8)/8

elif self.type == 'eig':

y = lambda x: self.a\*(x\*\*9)/9

elif self.type == 'nin':

y = lambda x: self.a\*(x\*\*10)/10

elif self.type == 'ten':

y = lambda x: self.a\*(x\*\*11)/11

elif self.type == 'ele':

y = lambda x: self.a\*(x\*\*12)/12

elif self.type == 'twe':

y = lambda x: self.a\*(x\*\*13)/13

elif self.type == 'thir':

y = lambda x: self.a\*(x\*\*14)/14

elif self.type == 'fourt':

y = lambda x: self.a\*(x\*\*15)/15

elif self.type == 'fift':

y = lambda x: self.a\*(x\*\*16)/16

elif self.type == 'sixt':

y = lambda x: self.a\*(x\*\*17)/17

elif self.type == 'sevet':

y = lambda x: self.a\*(x\*\*18)/18

elif self.type == 'eight':

y = lambda x: self.a\*(x\*\*19)/19

elif self.type == 'nint':

y = lambda x: self.a\*(x\*\*20)/20

elif self.type == 'twen':

y = lambda x: self.a\*(x\*\*21)/21

elif self.type == 'tweno':

y = lambda x: self.a\*(x\*\*22)/22

elif self.type == 'twent':

y = lambda x: self.a\*(x\*\*23)/23

elif self.type == 'twenth':

y = lambda x: self.a\*(x\*\*24)/24

elif self.type == 'twenf':

y = lambda x: self.a\*(x\*\*25)/25

elif self.type == 'twenfi':

y = lambda x: self.a\*(x\*\*26)/26

elif self.type == 'twensi':

y = lambda x: self.a\*(x\*\*27)/27

elif self.type == 'twense':

y = lambda x: self.a\*(x\*\*28)/28

elif self.type == 'twenei':

y = lambda x: self.a\*(x\*\*29)/29

return y

1. ANN architecture: ag\_ann.py

class NN:

def \_\_init\_\_(self, ni, nh, nh1 ,no):

# number of input, hidden\_layer one, hidden layer two and output nodes

self.ni = ni + 1 # +1 for bias node

self.nh = nh

self.nh1 = nh1

self.no = no

# activations for nodes

self.ai = [1.0]\*self.ni

self.ah = [1.0]\*self.nh

self.ah1 =[1.0]\*self.nh1

self.ao = [1.0]\*self.no

# create weights

self.wi = makeMatrix(self.ni, self.nh)

self.wi1 = makeMatrix(self.nh,self.nh1)

self.wo = makeMatrix(self.nh1, self.no)

# set them to random vaules

for i in range(self.ni):

for j in range(self.nh):

self.wi[i][j] = rand(0.02, 0.020)

for i in range(self.nh):

for j in range(self.nh1):

self.wi1[i][j] = rand(0.002,0.0020)

for j in range(self.nh1):

for k in range(self.no):

self.wo[j][k] = rand(-0.2, 0.2)

# last change in weights for momentum

self.ci = makeMatrix(self.ni, self.nh)

self.ci1 = makeMatrix(self.nh,self.nh1)

self.co = makeMatrix(self.nh1, self.no)

def update(self, inputs):

if len(inputs) != self.ni-1:

raise ValueError('wrong number of inputs')

# input activations

for i in range(self.ni-1):

# print (i)

# print (inputs[i])

#self.ai[i] = sigmoid(inputs[i])

self.ai[i] = inputs[i]

# hidden activations

for j in range(self.nh):

sum = 0.0

for i in range(self.ni):

sum = sum + self.ai[i] \* self.wi[i][j]

self.ah[j] = sigmoid(sum)

#hidden1 activations

for j in range(self.nh1):

sum = 0.0

for i in range(self.nh):

sum = sum + self.ah[i] \* self.wi1[i][j]

self.ah1[j] = sigmoid(sum)

# output activations

for k in range(self.no):

sum = 0.0

for j in range(self.nh1):

sum = sum + self.ah1[j] \* self.wo[j][k]

self.ao[k] = sigmoid(sum)

return self.ao[:]

def pred(self, inputs):

if len(inputs) != self.ni-1:

raise ValueError('wrong number of inputs')

# input activations

for i in range(self.ni-1):

# print (i)

# print (inputs[i])

#self.ai[i] = sigmoid(inputs[i])

self.ai[i] = inputs[i]

# hidden activations

for j in range(self.nh):

sum = 0.0

for i in range(self.ni):

sum = sum + self.ai[i] \* self.wi[i][j]

self.ah[j] = sigmoid(sum)

#hidden1 activations

for j in range(self.nh1):

sum = 0.0

for i in range(self.nh):

sum = sum + self.ah[i] \* self.wi1[i][j]

self.ah1[j] = sigmoid(sum)

# output activations

for k in range(self.no):

sum = 0.0

for j in range(self.nh1):

sum = sum + self.ah1[j] \* self.wo[j][k]

self.ao[k] = sigmoid(sum)

return self.ao[:]

def backPropagate(self, targets, N, M):

if len(targets) != self.no:

raise ValueError('wrong number of target values')

# calculate error terms for output

output\_deltas = [0.0] \* self.no

for k in range(self.no):

# print (targets[k])

error = targets[k]-self.ao[k]

output\_deltas[k] = dsigmoid(self.ao[k]) \* error

# print (self.ah1)

# print (self.ah)

# calculate error terms for hidden1

hidden\_deltas1 = [0.0] \* self.nh1

for j in range(self.nh1):

error = 0.0

for k in range(self.no):

error = error + output\_deltas[k]\*self.wo[j][k]

hidden\_deltas1[j] = dsigmoid(self.ah1[j]) \* error

# calculate error terms for hidden

hidden\_deltas = [0.0] \* self.nh

for j in range(self.nh):

error = 0.0

for k in range(self.nh1):

error = error + hidden\_deltas1[k]\*self.wi1[j][k]

hidden\_deltas[j] = dsigmoid(self.ah[j]) \* error

# update output weights

for j in range(self.nh1):

for k in range(self.no):

change = output\_deltas[k]\*self.ah1[j]

self.wo[j][k] = self.wo[j][k] + N\*change #+ M\*self.co[j][k]

self.co[j][k] = change

#print N\*change, M\*self.co[j][k]

# update hidden1 layer weight

for j in range(self.nh):

for k in range(self.nh1):

change = hidden\_deltas1[k]\*self.ah[j]

self.wi1[j][k] = self.wi1[j][k] + N\*change #+ M\*self.ci1[j][k]

self.ci1[j][k] = change

# update input weights

for i in range(self.ni):

for j in range(self.nh):

change = hidden\_deltas[j]\*self.ai[i]

self.wi[i][j] = self.wi[i][j] + N\*change + M\*self.ci[i][j]

self.ci[i][j] = change

# calculate error

error = 0.0

for k in range(len(targets)):

self.ao[k]

# error = 0.99

error += 0.5\*(targets[k]-self.ao[k])\*\*2

return error,self.wi,self.wi1,self.wo

def test(self, inputs):

#print (self.wo)

#print (self.wi1)

#print (self.wi)

# for i in range(len(inputs)):

g = []

for p in inputs:

g.append(self.update(p))

# print(p, '->', self.update(p))

return g

def weights(self):

print('Input weights:')

for i in range(self.ni):

# print(i)

print(self.wi[i])

print('=================================')

for i in range(self.nh1):

print(self.wi1[i])

print('+++++++++++++++++++++++++++++++++')

print('Output weights:')

for j in range(self.nh):

print(self.wo[j])

def train(self, train\_x, train\_y, iterations=10000, N=0.1, M=1):

# N: learning rate

# M: momentum factor

# lenght of train\_x and train\_y are equal

#error = 0.0

# while (p):

for w in range(len(train\_y)-4):

r = random.randint(-2,2,)

for i in range(iterations):

error = 0.0

for p,q in zip ([train\_x],[train\_y]):

inputs = p[9]

# print (p[w])

# exit()

targets = q[9]

# print (targets[w])

# exit()

z = self.update(inputs)

error1,wi,wi1,wo = self.backPropagate(targets, N, M)

error += error1

'''if i % 1000 == 0:

#self.weights()

print('wwwwwwwwwwwwwwwwwwwiiiiiiiiiiiiiiiiiiiwwwwwwwwwwwwwwwwwwwwwiiiiiiiiiiiiiiiiiiiii')

print(wi)

print ('wwwwwwwwwwwwwwwwwwwiiiiiiiiiiiiiiiiiii11111111111111111wwwwwwwwwwwwwwwwwiiiiiii111')

print(wi1)

print('wwwwwwwwwwwwwwwwwwwwwwwooooooooooooooooooooooooowwwwwwwwwwwwwwwwwwwwwwwwwwwwoooooooo')

print(wo)

print('error {} after iteration {} {} {}, actula output is: {}'.format(error,i,p[w],q[w],z))'''

'''if error < 0.001:

break

else:

continue

break

else:

continue

break'''

1. Main.py

def datasets(type,a,b):

'''

this function is used for the generating mathematical fucntion for definiteintegral

'''

x = equation(type,a,b)

integrand = x.equation()

print (" Your function is: {}\*x+{} ".format(a,b))

integral = x.eq\_int()

train\_x = []

train\_y = []

m = -10

n = 10

for i in frange(m, n,1):

# v\_integrand = integrand(i)

r = random.randint(-3,3)

p = (i-m)/(n-m)

#p = sigmoid(i)

# print (i,p)

v\_integral = integral(i+r)

#q = (v\_integral-0)/(integral(n)-0)

#q = sigmoid(v\_integral)

train\_x.append([i+r])

train\_y.append([v\_integral])

return train\_x,train\_y

def main():

# getting datasets

print ('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Hello fellow!!!\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

print (' Calculating integration of linear function only f(x) = a\*x+b')

print (' Using parameters: ')

print (' 1. type = lin for linear function')

print (' 2. a = slope of line')

print (' 3. b = constant')

print ("+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++")

print ("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

type = input(" Enter the type of function: ")

a = int(input(" Enter the slope: "))

b = int(input(" Enter the constant: "))

train\_x,train\_y = datasets(type,a,b)

print ("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print (" Provide me the upper and lower limit for integration")

a1 = int(input(" Upper limit is: "))

a2 = int(input(" Lower limit is: "))

#print ((train\_y[140:200]))

# exit()

##print ("+=================================================================================================")

print("++++++++++++++++++++++++++++++++++ wait wait wait !!!! ++++++++++++++++++++++++++++++++++++++++++++++++++++++++")

#print (train\_x[140:200])

# exit()

# create a network with one input, seven hidden neurons, two hidden layer ,and one output neurons

n = NN(1,7,7,1)

#n.weights()

# exit()

# train it with train\_x and train\_y

n.train(train\_x,train\_y,10000,0.0001,0.1)

# testing

# p1 = n.test(train\_x)

p2 = n.test([[a1],[a2]])

# print (p2[1])

#p3 = n.test([[(a2)]])

# n.test(train\_x[71:100])

print ("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Thanks for waiting\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

print (" Result is {}".format(p2[0][0]-p2[1][0]))

print ("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* one day, I will calculate the integration of all type.\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

if \_\_name\_\_ == '\_\_main\_\_':

main()

1. **Analytical Method:**
2. Approximation class

from my\_math import \*

class approximation():

def \_\_init\_\_(self,function,a,b,n=7):

'''

function = input function or we can say integrand

a = lower limit of the interval

b = upper limit of the interval

'''

self.function = function

self.a = a

self.b = b

self.n = n

def midpoint(self):

h = float(self.b-self.a)/self.n

result = 0

for i in range(self.n):

result += self.function((self.a + h/2.0) + i\*h)

result \*= h

return result

def trapezoidal(self):

h = float(self.b-self.a)/self.n

result = 0.5\*self.function(self.a) + 0.5\*self.function(self.b)

for i in range(1, self.n):

result += self.function(self.a + i\*h)

result \*= h

return result

def tanh(self):

h = float(self.b-self.a)/self.n

result = 0

for i in range(self.n):

result += tanh(h/2\*self.function(self.a+h/2.0+i\*h))

result \*= 2

return result

1. Integration calculation: cal.py

from equation import \*

from approximation\_layer import \*

def main():

fun = equation('lin',2)

fun1 = fun.equation()

p = approximation(fun1,1,0)

p2 = p.tanh()

p1 = p.midpoint()

print (p2)

print (p1)

if \_\_name\_\_='\_\_main\_\_':

main()

1. Using area methods:

from equation import \*

def area(lst):

l\_lst = len(lst)

area\_of\_curve = 0

for i in range(len(lst)-1):

if lst[i] == lst[i+1]:

print ('square')

area = 1\*(lst[i+1])

else:

area\_t = 0.5\*(lst[i+1]-lst[i])

area\_s = 1\*lst[i]

area = area\_t + area\_s

area\_of\_curve += area

return area\_of\_curve

p = []

for i in range(10):

p.append(i)

# data creation for linear equation

y = equation ('lin', 2,3)

#for j in range(1,100):

in\_data = []

area\_data = []

for i in frange(0,3,1):

# print(i)

y1 = y(i)

# print (y1)

in\_data.append(y1)

area\_for\_data = area(in\_data)