**Computer Intelligence**

**Kunnapat Thippayapalaphonkul (590612113)**

**1 December 2019**

**รายงานผลการทดลองการบ้าน Swarm Intelligence**

ในการบ้านรายงานนี้ นั้นได้ทำการเรียกใช้ Libraries 1). numpy 2). copy 3). random 4). Matplotlib 5). pandas โดยที่ numpy นั้นเอาไว้ใช้ในการคำนวณต่างๆ เช่น dot product, mean, random matrix เป็นต้น copy ใช้ทำการ copy List, random ใช้ในการเลือก Chromosome ในขั้นตอนต่างๆ matplotlib เพื่อ Plot Graph และ pandas ใช้เพื่ออ่านไฟล์ Data set เนื่องจากเป็น excel

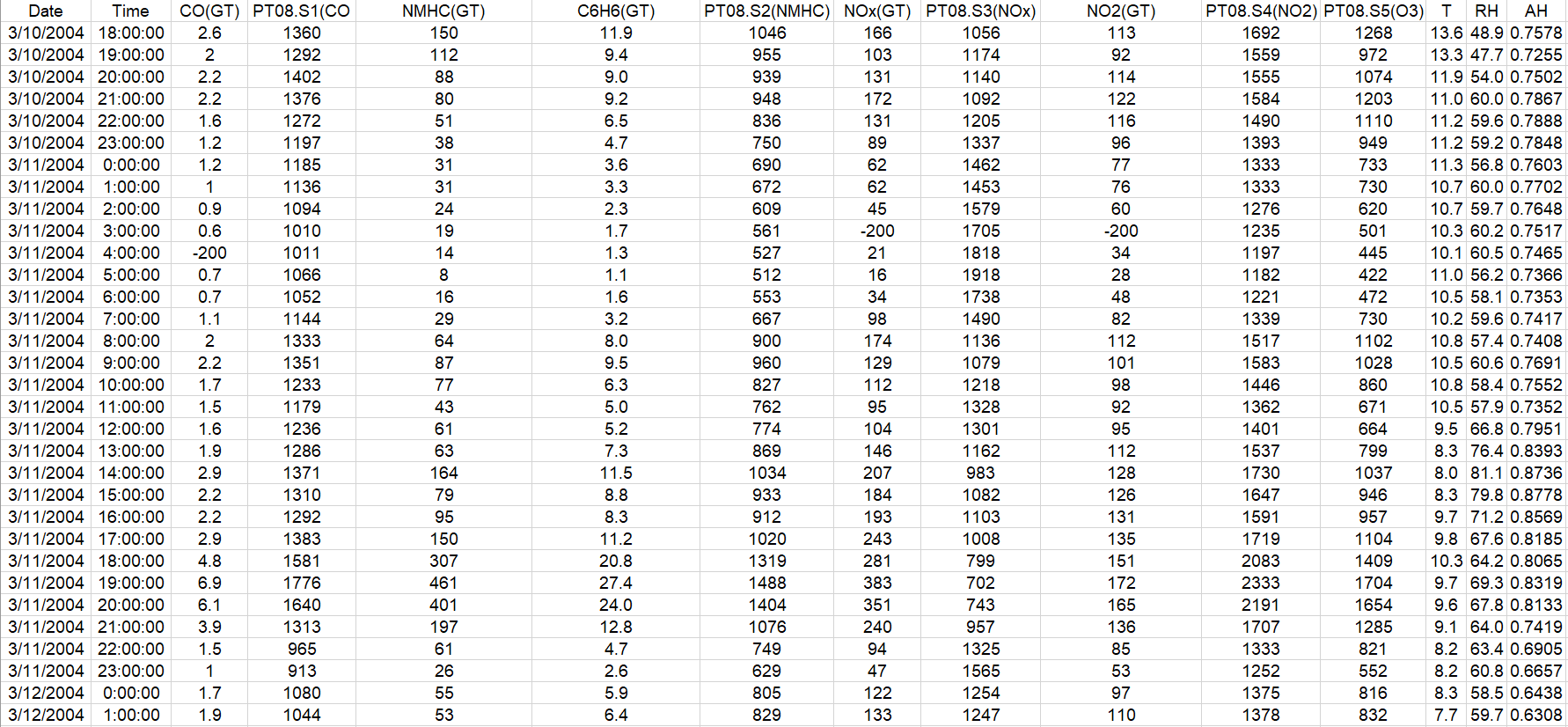
รายงานผลการทดลองทำกับ Data set [Air Quality Data set](https://archive.ics.uci.edu/ml/datasets/air+quality) โดยที่ Data set นี้มี 15 Attributes โดยที่มี 9 Features, Attributes ที่ 5 จะเปิดค่าเฉลย True hourly averaged Benzone concentration in microg/m^3 (reference analyzer) โดยที่ให้ทำการ Regression โดยที่การบ้านนี้นั้นได้ทำการใช้ Neural Network ในการคำนวณหา Error และจากขั้นตอน Backpropagation เปลี่ยนไปใช้ Swarm Intelligence แทนในการบ้านนี้ โดยมีขั้นตอนดังนี้

* Preprocessing Data
* Cross Validation
* Neural Network
* Fitness
* Compare Pbest
* Compare Gbest
* Update Velocity
* Update Position

**ขั้นตอน**

**Preprocessing Data**

ในขั้นตอนนี้เช็ค Data ว่าสมบูรณ์รึเปล่าในแต่ละ Attributes จากนั้นใน Data จะมี 15 Attributes ใน Data set ชุดนี้โดยจะมี Data โดยที่แต่ละ Attributes โดยเริ่มนับจาก Column แรกเริ่มเป็น 0 และที่ใช้ในการ Train ก็คือ Attributes [3, 6, 8, 10, 11, 12, 13, 14] และ Attributes ที่เป็นคำตอบในการ Regression คือ Attribute ที่ 5 ซึ่งตัวอย่าง Data set จะเป็นดัง รูปที่ 1 จะได้เห็นว่า Column 0, 1, 2, 4, 7, 9 นั้นจะไม่ใช้ในการบ้านครั้งนี้ แต่จะใช้ Column ที่เหลือทั้งหมด



รูปที่ 1 ตัวอย่าง Data set

จาก รูปที่ 1 ใน Column ที่ 5 หรือ Column ที่ชื่อ C6H6 นั้นจะเป็นคำตอบของ Data set ชุดนี้ ซึ่งข้อมูลนั้นเป็นเป็นตัวเลขทั้งหมดเลย ดังนั้นจึงทำ Regression เพื่อที่จะได้ Predict Benzone Concentration และข้อมูลในทุก Column ที่สนใจนั้นเป็นตัวเลขทั้งหมดแต่ในแต่ละ Column นั้นมีระดับที่ต่างกัน จึงทำการ Normalize ในแต่ละ Column เพื่อที่จะให้ Data นั้นอยู่ในระดับเดียวกัน [0, 1] ในแต่ละ Column จะทำการหา Max และ Min เพื่อนำมาคิด x ในแต่ละ Column ดัง [ 1 ]

[ 1 ]

**Cross Validation**

ได้ทำการแบ่ง Data set เป็น 10 Fold โดยที่แต่ละ Fold นั้นแบ่งเป็น Training set 90 % และ Testing set 10 % ดัง รูปที่ 2 ทำเป็นจำนวน 10 Fold โดยที่ Data set ชุดนี้มีขนาดจำนวน 9,357 Sample แบ่งเป็นให้ Testing set ประมาณ 10 % เท่ากับ 935 Sample ที่เหลือแบ่งใส่ Training set จำนวน 8,422 Sample โดยที่ Training set จะไม่มีข้อมูลของ Testing set เลยเพื่อเช็คความถูกต้องของ Neural Network ที่ได้ทำการ Train ว่าไม่ได้ Overfitting กับ Training set จนเกินไป

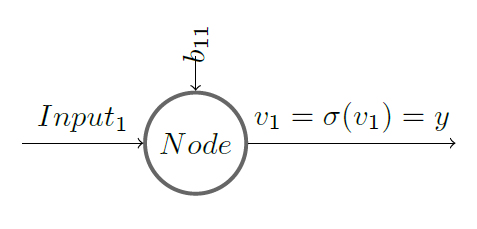
Training set 90 %

Testing set 10 %

รูปที่ 2 ตัวอย่างการทำ Cross Validation

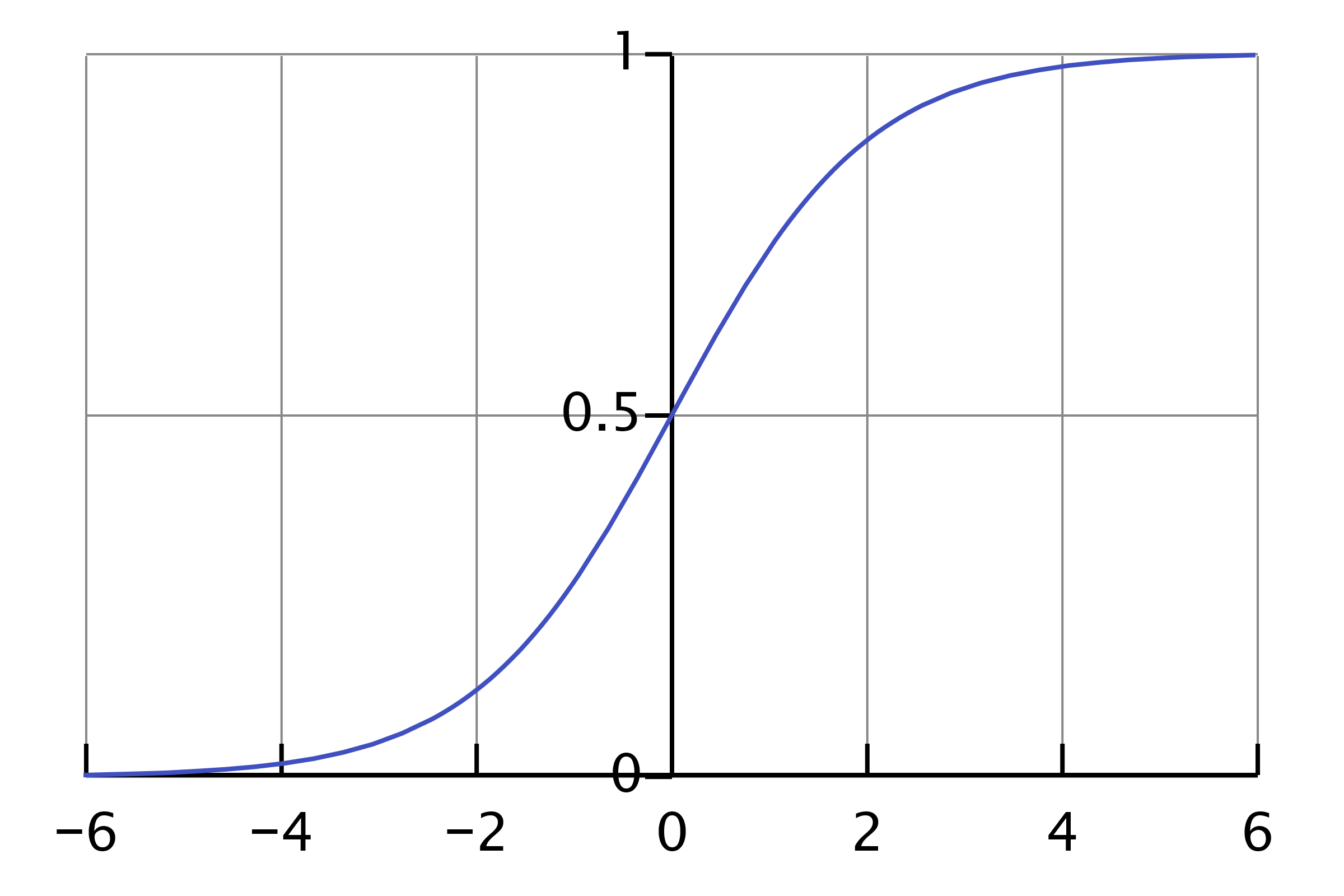
**Neural Network**

ในขั้นตอนนี้จะทำการสร้าง Neural Network โดยที่จะกำหนดค่าเริ่มต่างๆในกับ Network เพื่อที่จะ Train Network ให้ได้ผลตามที่เราต้องการ Neural Network จะมีการคำนวณดัง รูปที่ 3



รูปที่ 3 Node ของ Neural Network

หลังจากคำนวณแต่ละ Node เสร็จ จะทำการนำค่าที่คำนวณได้ไปทำการผ่าน Activation Function โดยที่ Activation Function ที่ผมได้เลือกใช้ในการบ้านนี้คือ Sigmoid Function โดยผลที่นำไปผ่าน Sigmoid Function จะได้ผลดัง รูปที่ 4



รูปที่ 4 ค่า Sigmoid Function

ผลลัพธ์ที่ได้จาก Sigmoid Function นั้นจะอยู่ระหว่าง [0, 1] ซึ่งทำการคำนวณแต่ละ Node ไปเรื่อย ๆ จนถึง Node สุดท้ายเพื่อที่จะหาค่า Error ของ Neural Network โดยการกำหนดค่าเริ่มต้นต่างๆให้กับ Neural Network มีดังนี้

* Input, Output ของ Neural Network สำหรับ Training set
* กำหนดจำนวน Node ในแต่ละ Layer
* จำนวน Population
* จำนวน Group\_population

**Fitness**

ขั้นตอนนี้หลังการคำนวณ Loss หรือ Error จาก Network โดยปกติเราจะอยากได้ค่า Error ของ Network ให้มีค่าน้อยที่สุดเท่าที่จะทำได้ โดยที่ในการบ้านนี้จะใช้การคำนวณ Error ทั้งหมดคือ Mean Absolute Error (MAE) และค่า Fitness จะเป็นค่าผลลัพธ์ของแต่ละ Group\_population ก็จะใช้ค่า MAE เช่นเดียวกับ Error เลยดัง [ 2 ]

[ 2 ]

**Compare Pbest**

ในขั้นตอนนี้นั้นจะทำการเปรียบเทียบเพื่อหาค่า Fitness ที่ดีที่สุดในตัวเองที่สามารถทำได้ในแต่ละ Iteration ซึ่งถ้าเจอค่าที่ดีที่สุดแล้วจะทำการอัพเดทค่าที่ดีที่สุดและตำแหน่งที่ดีที่สุด ให้กับตัว Pbest

**Compare Gbest**

ในขั้นตอนนี้นั้นจะเหมือนกับขั้นตอน Pbest เลยก็คือทำการเทียบค่า Fitness ของทุกคนเพื่อหาค่าของ Fitness ที่ดีที่สุดในสังคมและในทุก Iteration เพื่อที่จะได้ค่า Fitness ที่ดีที่สุดและตำแหน่งที่ดีที่สุดด้วย

**Update Velocity**

ในขั้นตอนนี้นั้นจะทำการอัพเดทค่าความเร็วของแต่ละกลุ่มและภายในกลุ่มคือแต่ละคนด้วย โดยที่ใช้สมการ Update ความเร็วดัง [ 3 ] ซึ่งความเร็วที่ได้ Update ในแต่ iteration นั้นขึ้นกับ และ เพื่อกำหนดความเร็วในแต่ละ กลุ่มว่าควรเป็นเท่าไหร่จากความเร็วในรอบที่แล้ว

[ 3 ]

**Update Position**

ในขั้นตอนนี้นั้นจะทำการอัพเดทตำแหน่งของแต่ละกลุ่มโดยที่นำ ความเร็วที่คำนวณมาจาก [ 3 ] มาอัพเดทตำแหน่งของแต่ละกลุ่มดัง [ 4 ]

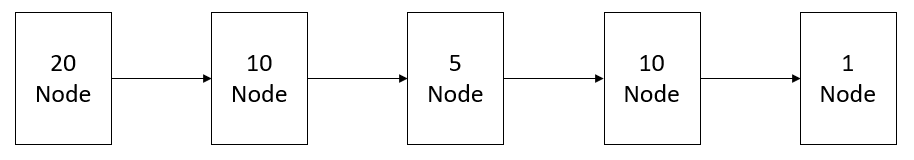
[ 4 ]

**ผลการทดลอง**

การทดลองครั้งนี้นั้นจะทดลองโดยมี 50 Group Population และมี 100 Iteration

**การทดลองครั้งที่ 1**

ซึ่งจะทำการทดลองโดยสร้าง Neural Network ที่มี Layer และ Node ดัง รูปที่ 5



รูปที่ 5 จำนวน Layer และ Node ในแต่ละ Layer

ตาราง 1 ค่า Fitness ของแต่ละ Fold ของการทดลองครั้งที่ 1

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

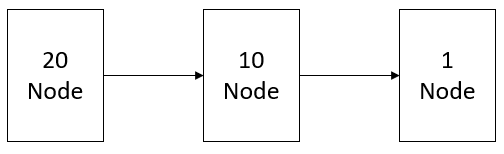
จาก ตาราง 1 จะเห็นได้ว่ามีหลายๆ Fold ที่ลู่เข้า แต่ในหลายๆ Fold ที่ไม่ได้ลู่เข้าคิดว่าเนื่องจากผมได้ทำการ Random 1 ครั้งต่อหนึ่ง Fold ซึ่งอาจจะ Random แล้วได้ค่าที่มากจนเกินไปทำให้ค่า MAE ที่คำนวณมาจาก Network นั้นไม่สามารถลู่เข้าได้ในแต่ละ Iteration และจาก ตาราง 2 ค่า Fitness ที่ได้จาก Test set ของแต่ละ Fold ที่ดีที่สุดคือ Fold 1

ตาราง 2 แสดงค่า Fitness ของ Test set แต่ละ Fold ของการทดลองครั้งที่ 1

|  |  |
| --- | --- |
| **Fold** | **Fitness** |
| Fold 1 | 0.0158610425710746 |
| Fold 2 | 0.02451764203349384 |
| Fold 3 | 0.03069327744343776 |
| Fold 4 | 0.035096333685947785 |
| Fold 5 | 0.03575401426818805 |
| Fold 6 | 0.07399093286914757 |
| Fold 7 | 0.032029285621598354 |
| Fold 8 | 0.036856660548262116 |
| Fold 9 | 0.05192566953728086 |
| Fold 10 | 0.10832223634059243 |

**การทดลองครั้งที่ 2**

การทดลองครั้งที่ 2 จะทำการลดจำนวน Layer ลงมาดัง รูปที่ 6



รูปที่ 6 จำนวน Layer และ Node ในแต่ละ Layer

ตาราง 3 ค่า Fitness ของแต่ละ Fold ของการทดลองครั้งที่ 2

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

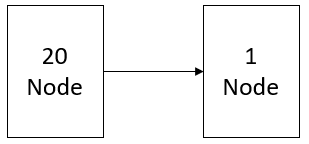
จาก ตาราง 3 จะเห็นว่าหลายๆ Fold นั้นไม่ได้ลู่เข้าเพราะการ Random ขึ้นมานั้นอาจจะเยอะเกินไปทำให้แต่ละ Group\_Population นั้นไม่สามารถที่จะลู่เข้าได้ แต่ก็จะมีหลายๆ Fold ที่ลู่เข้าและจาก ตาราง 4 ค่า Fitness ของ Test set ที่น้อยที่สุดจากทั้ง 10 Fold นั้นคือ Fold 1 แต่จาก ตาราง 3 Fold ที่ 1 นั้นไม่ได้ลู่เข้าแต่ทุกๆ Fold นั้นได้ค่า Fitness ที่น้อยทั้งหมด

ตาราง 4 แสดงค่า Fitness ของ Test set แต่ละ Fold ของการทดลองครั้งที่ 2

|  |  |
| --- | --- |
| **Fold** | **Fitness** |
| Fold 1 | 0.010234164561734326 |
| Fold 2 | 0.0392490543667484 |
| Fold 3 | 0.06558196379984695 |
| Fold 4 | 0.026689723109169575 |
| Fold 5 | 0.03490345632715396 |
| Fold 6 | 0.023004671560783153 |
| Fold 7 | 0.021112299037273317 |
| Fold 8 | 0.0479491088793699 |
| Fold 9 | 0.027763087890868184 |
| Fold 10 | 0.02774983859562331 |

**การทดลองครั้งที่ 3**

การทดลองครั้งที่ 3 จะทำการลดจำนวน Layer ลงมาดัง รูปที่ 7



รูปที่ 7 จำนวน Layer และ Node ในแต่ละ Layer

ตาราง 5 ค่า Fitness ของแต่ละ Fold ของการทดลองครั้งที่ 3

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

จาก ตาราง 5 นั้นจะเห็นได้ว่าเกือบทุก Fold นั้นไม่ได้ลู่เข้าเลยแต่ค่าในแต่ละ Iteration นั้นกระโดดไปมาอาจจะเพราะการ Random แต่ละ Fold นั้นสูงทั้งหมดแต่นั้นอาจจะไม่ใช่เหตุผลเดียวโดยจาก Network ในการทดลองครั้งนี้เป็นไปตาม รูปที่ 7 จะเห็นได้ว่าไม่มี Hidden Layer เลยและจำนวน Node นั้นก็ยังน้อยอีกด้วยทำให้ค่าในแต่ละ Fold นั้นไม่ได้ลู่เข้าแต่กลับกันจาก

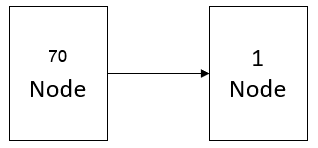
ตาราง *6* จะเห็นได้ว่าค่า Fitness กับ Test set แต่ละ Fold นั้นก็ยังได้ต่ำอยู่และ Fold ที่มีค่า Fitness ต่ำที่สุดคือ Fold 2

ตาราง 6 แสดงค่า Fitness ของ Test set แต่ละ Fold ของการทดลองครั้งที่ 3

|  |  |
| --- | --- |
| **Fold** | **Fitness** |
| Fold 1 | 0.034780554758632384 |
| Fold 2 | 0.01162929276929691 |
| Fold 3 | 0.02030348051973223 |
| Fold 4 | 0.08624744167831479 |
| Fold 5 | 0.04403095896623724 |
| Fold 6 | 0.10940734642801157 |
| Fold 7 | 0.13114373633746823 |
| Fold 8 | 0.05597359275446113 |
| Fold 9 | 0.01946293575171072 |
| Fold 10 | 0.03727316575478198 |

**การทดลองครั้งที่ 4**

จากการทดลองครั้งที่ 3 จะทำการลดจำนวน Layer ลงมาดัง รูปที่ 8



รูปที่ 8 จำนวน Layer และ Node ในแต่ละ Layer

ตาราง 7 ค่า Fitness ของแต่ละ Fold ของการทดลองครั้งที่ 4

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

จากการทดลองที่ 3 นั้นได้ทำการเพิ่มจำนวน Node ในการทดลองที่ 4 นี้ดัง รูปที่ 8 เพื่อดูว่าถ้าเพิ่มจำนวน Node แล้วแต่ละ Fold นั้นจะลู่เข้าหรือเปล่า โดยที่ผลการทดลองดัง ตาราง 7 ซึ่งจะเห็นได้ว่าไม่ได้มีผลมากนักและจาก ตาราง 8 จะเห็นได้ว่าแต่ค่า Fitness ของแต่ละ Fold นั้นสูงที่สุดในการทดลองทั้ง 4 ครั้ง ค่า Fitness ที่น้อยที่สุดคือ Fold 7

ตาราง 8 แสดงค่า Fitness ของ Test set แต่ละ Fold ของการทดลองครั้งที่ 4

|  |  |
| --- | --- |
| **Fold** | **Fitness** |
| Fold 1 | 0.13734909974084558 |
| Fold 2 | 0.08268511117110684 |
| Fold 3 | 0.054716382039849576 |
| Fold 4 | 0.1244549756095977 |
| Fold 5 | 0.10803180740706839 |
| Fold 6 | 0.08731068734815368 |
| Fold 7 | 0.04267949753843406 |
| Fold 8 | 0.046782312134706905 |
| Fold 9 | 0.18069372544965698 |
| Fold 10 | 0.04582749685045916 |

**สรุปผลการทดลอง**

จากผลการทดลองทั้ง 4 ครั้งนั้นจะเห็นได้ว่าจากกราฟของแต่ละ Fold ทั้ง 10 Fold นั้นพอลดจำนวน Hidden Layer ลงนั้นจะเห็นได้ว่าค่า Fitness นั้นคิดมาจาก [ 2 ] คือค่า MAE นั้นของแต่ละ Fold Training set นั้นไม่ได้แย่ไปกว่าเดิมเลยเพราะเนื่องจากได้ทำการ Random ซึ่งสองค่านี้อาจจะได้สูงเลยทำให้หลายๆ Fold นั้นไม่สามารถที่จะลู่เข้าได้แต่จะเห็นได้ว่าการทดลองที่ 3, 4 นั้นถ้า Fitness นั้นแทบจะไม่ลู่เข้าเลยเนื่องจากทำการลด Hidden Layer ไปจึงทำให้ค่าในแต่ละ Fold ไม่ลู่เข้า แต่จากการทดลองทั้ง 4 ครั้งดัง ตาราง 9 นั้นค่า Fitness ถึงจะลด Hidden Layer ลงแต่ค่า Fitness ที่ดีที่สุดของแต่ละ Fold การทดลองนั้นแทบจะไม่แตกต่างกันเลยดังนั้นคิดว่าปัจจัยที่มีผลต่อการอัพเดทตำแหน่งที่สำคัญของแต่ละ Group\_Population นั้นคือ เพราะจากการทดลองทั้ง 4 ครั้งนั้นได้ทำการ Random ครั้งเดียวต่อ Fold ทำให้ถ้าเกิดว่า Random ได้ค่ามากจะทำให้ไม่สามารถลู่เข้าได้

ตาราง 9 สรุปของการทดลอง Fold ไหนที่มีค่า Fitness มากที่สุด

|  |  |  |
| --- | --- | --- |
| **การทดลอง** | **Fold** | **Fitness** |
| 1 | 1 | 0.0158610425710746 |
| 2 | 1 | 0.010234164561734326 |
| 3 | 2 | 0.01162929276929691 |
| 4 | 7 | 0.04267949753843406 |

**ภาคผนวก**

import numpy as np

import copy

import random

import matplotlib

import matplotlib.pyplot as plt

import pandas as pd

#### Attributes | 3, 6, 8, 10, 11, 12, 13, 14 to input

#### Attributes | 5 to output

def main():

    input\_x, true\_x = getData()

    input\_x = transposeList(input\_x)

    x\_train, y\_train, x\_test, y\_test= createFold(10, input\_x, true\_x)

    for i in range(len(x\_train)): #### Fold

        print("Fold {}" .format(i+1))

        neural = NeuralNetwork(x\_train[i], y\_train[i])

        neural.addLayer(70)

        # neural.addLayer(10)

        # neural.addLayer(5)

        # neural.addLayer(10)

        neural.addLayer(1)

        neural.fit(50, 100)

        neural.plotFitness(i+1)

        neural.evaluate(x\_test[i], y\_test[i])

        print("---------------")

def getData(): ### return list

    path = 'AirQualityUCI.xlsx'

    listdata = []

    df = pd.read\_excel(path)

    x = []

    cou = 0

    for atr in df.columns:

        if cou == 3 or cou == 6 or cou == 8 or cou >= 10 :

            x.append(normalizeList(df[atr]))

        elif cou == 5 :

            y = normalizeList(df[atr])

        cou += 1

    return x, y

def transposeList(datalist):

    return list(map(list, zip(\*datalist)))

def normalizeList(x):

    # x = transposeList(x)

    nor\_x = []

    minx = min(x)

    maxx = max(x)

    for i in range(len(x)):

        nor = (x[i] - minx)/(maxx - minx)

        nor\_x.append(nor)

    return nor\_x

class NeuralNetwork:

    def \_\_init\_\_(self, inpu, out):

        # np.random.seed(1)

        self.Fullinput = inpu

        self.FullTrueOutput = out

        self.input = np.asarray(copy.deepcopy(inpu))

        self.weight = []

        self.bias = []

        self.predict = 0

        self.lr = 0

        self.E = 0

        self.countLayer = 0

        self.Node = []

        self.output = []

        self.TrueOutput= None

        self.cou = 0

        self.deltaweight = []

        self.err = []

        self.deltabias = []

        self.gradient = [] # back to front

        self.momentum = 0

        self.Loss = []

        self.minimumLoss = 10000000

        self.rememberweightT\_1 = []

        self.countsample = 0

        self.rememberweightT\_2 = []

        self.npopulation = 0

        self.group\_pop = []

        self.pbest = []

        self.gbest = [[100000, 0]]

        self.velocity = []

    def addLayer(self, node):

        self.countLayer+=1

        self.Node.append(node)

    def sigmoid(self, v):

        return 1/(1+np.exp(-v))

    def diffsigmoid(self, y):

        return y\*(1-y)

    def createweight(self):

        for i in range(len(self.Node)):

            if ( i == 0 ):

                self.weight.append(2\*np.random.rand(len(self.input), self.Node[i]) - 1)

                self.bias.append(2\*np.random.rand(self.Node[i]) - 1)

                # self.deltaweight.append(np.ones((len(self.input), self.Node[i])))

                # self.deltabias.append(np.ones(self.Node[i]))

                # self.rememberweightT\_1.append(np.zeros((len(self.input), self.Node[i])))

                # self.rememberweightT\_2.append(np.zeros((len(self.input), self.Node[i])))

            else:

                self.weight.append(2\*np.random.rand(self.Node[i-1], self.Node[i]) - 1)

                self.bias.append(2\*np.random.rand(self.Node[i]) - 1)

                # self.deltaweight.append(np.ones((self.Node[i-1], self.Node[i])))

                # self.deltabias.append(np.ones(self.Node[i]))

                # self.rememberweightT\_1.append(np.zeros((len(self.input), self.Node[i])))

                # self.rememberweightT\_2.append(np.zeros((len(self.input), self.Node[i])))

    def FeedForward(self, chromosome): # each sample, each generation

        self.output = []

        self.output.append(self.input.T)

        out = np.array(self.output[0])

        for i in range(len(self.Node)): # feed in each layer

            v = np.dot(copy.deepcopy(chromosome[i].T), copy.deepcopy(self.output[i]))

            out = self.sigmoid(v)

            self.output.append(out)

        return out

    def fit(self, npopulatation, iteration):

        self.npopulation = npopulatation

        self.fitnessplot = []

        for k in range(iteration) :

            fitne = []

            for i in range(npopulatation): # individual

                if k == 0 :

                    temp = self.population()

                    temp = np.asarray(copy.deepcopy(temp))

                    out = self.FeedForward(temp)

                else :

                    out = self.FeedForward(copy.deepcopy(self.group\_pop[i][1]))

                self.Loss = []

                for j in range(len(self.Fullinput)): # sample

                    err = out[0][j] - copy.deepcopy(self.FullTrueOutput[j])

                    self.Loss.append(err)

                mae = np.asarray(copy.deepcopy(self.Loss))

                fitness = self.fitness(mae)

                fitne.append(fitness)

                #### compare find pbest and gbest

                if k == 0 :

                    self.group\_pop.append([fitness, temp])

                    self.velocity.append(copy.deepcopy(self.population()))

                    self.pbest.append(copy.deepcopy(self.group\_pop[i]))

                # else :

                if fitness < self.pbest[i][0] : ### best of self

                    self.pbest[i][0] = copy.deepcopy(fitness)

                    self.pbest[i][1] = copy.deepcopy(self.group\_pop[i][1])

                if  fitness < self.gbest[0][0] : ### best of global

                    self.gbest[0][0] = copy.deepcopy(fitness)

                    self.gbest[0][1] = copy.deepcopy(self.group\_pop[i][1])

                # update velocity

                if k + i == 0 :

                    rho1, rho2 = self.rho()

                self.velocity[i] = copy.deepcopy(self.velocity[i]) + rho1\*(copy.deepcopy(self.pbest[i][1]) - copy.deepcopy(self.group\_pop[i][1])) + rho2\*(copy.deepcopy(self.gbest[0][1]) - copy.deepcopy(self.group\_pop[i][1]))

                # update position

                self.group\_pop[i][1] = copy.deepcopy(self.group\_pop[i][1]) + copy.deepcopy(self.velocity[i])

            self.fitnessplot.append(np.mean(np.asarray(copy.deepcopy(fitne))))

    def population(self): # return list

        weight = []

        for  i in range(len(self.Node)):

            if (i == 0):

                weight.append(np.random.uniform(-1, 1, (len(self.Fullinput[0]), self.Node[i])))

            else :

                weight.append(np.random.uniform(-1, 1, (self.Node[i-1], self.Node[i])))

        return weight

    def fitness(self, mae): #### Cal fitness from error | input Integer | how to fitness maximum

        return np.mean(np.abs(mae))

    def evaluate(self, x\_test, y\_test):

        x\_test = np.asarray(copy.deepcopy(x\_test))

        y\_test = np.asarray(copy.deepcopy(y\_test))

        self.input = x\_test

        fitne = []

        for i in range(len(self.group\_pop)):

            predict = self.FeedForward(self.group\_pop[i][1])

            Loss = []

            for j in range(len(y\_test)):

                err = predict[0][j] - y\_test[j]

                Loss.append(err)

            mse = np.asarray(copy.deepcopy(Loss))

            mse = np.power(mse, 2)/ 2

            mse = np.mean(mse) # scalar

            fitness = self.fitness(mse)

            fitne.append(fitness)

        fitne = np.asarray(copy.deepcopy(fitne))

        fitne = np.mean(fitne)

        print("Fitness = {}" .format(fitne))

    def plotFitness(self, fold):

        fitness = self.fitnessplot

        fig, ax = plt.subplots()

        ax.plot(range(1, len(fitness)+1), fitness)

        ax.set(xlabel='Iteration', ylabel='Fitness', title='Fold {}' .format(fold))

        fig.savefig("Fold {}.png" .format(fold))

    def rho(self):

        r1 = np.random.uniform(0, 1)

        r2 = np.random.uniform(0, 1)

        c1 = np.random.uniform(0, 2)

        c2 = np.random.uniform(0, 2)

        rho1 = r1\*c1

        rho2 =  r2\*c2

        return rho1, rho2

def createFold(fold, x\_data, y\_data):

    crossvalidation = int(len(x\_data)\*fold/100)

    # print(len(x\_data))

    x\_train\_testingset = []

    y\_true\_testingset = []

    x\_train\_trainingset = []

    y\_true\_trainingset = []

    for i in range(fold):

        if i == (fold-1) :

            x\_train\_testingset.append(x\_data[0+i\*crossvalidation:len(x\_data)])

            y\_true\_testingset.append(y\_data[0+i\*crossvalidation:len(y\_data)])

            x\_train\_trainingset.append(x\_data[0:0+i\*crossvalidation])

            y\_true\_trainingset.append(y\_data[0:0+i\*crossvalidation])

        else:

            x\_train\_testingset.append(x\_data[0+i\*crossvalidation:crossvalidation+i\*crossvalidation])

            y\_true\_testingset.append(y\_data[0+i\*crossvalidation:crossvalidation+i\*crossvalidation])

            x\_train\_trainingset1 = x\_data[0:i\*crossvalidation]

            x\_train\_trainingset2 = x\_data[crossvalidation\*(i+1):len(x\_data)]

            x\_train\_trainingset.append(x\_train\_trainingset1 + x\_train\_trainingset2)

            y\_true\_trainingset1 = y\_data[0:i\*crossvalidation]

            y\_true\_trainingset2 = y\_data[crossvalidation\*(i+1):len(x\_data)]

            y\_true\_trainingset.append(y\_true\_trainingset1 + y\_true\_trainingset2)

    return x\_train\_trainingset, y\_true\_trainingset, x\_train\_testingset, y\_true\_testingset

if \_\_name\_\_ == "\_\_main\_\_":

    main()