Computer Assignment 3

CPE 261456 (Introduction to Computational Intelligence)

โดย

นายธนาคม หัสแดง

รหัสนักศึกษา 590610624

เสนอ

ผศ.ดร. ศันสนีย์ เอื้อพันธ์วิริยะกุล

คณะวิศวกรรมศาสตร์ มหาวิทยาลัยเชียงใหม่

ลักษณะการทำงานของระบบ

เริ่มจากการโหลด text ไฟล์โดยจะเก็บอยู่ในรูปแบบของ dictionary ซึ่งจะช่วยให้สามารถใช้งานข้อมูลได้ง่ายขึ้น หลังจากนั้นจะเริ่มเข้าสู่กระบวนการ train นั่นคือ ต้องกำหนด รูปแบบของ MLP ว่าจะให้มี Hidden layer และ จำนวน Layer ขนาดเท่าใด และ หลังจากนั้นจะต้องมากำหนด จำนวน Chromosome ของการ train โดยเมื่อกำหนด ส่วนนี้ครบแล้ว จะเข้าสู่ลูป เพื่อ ทำ cross validation โดยกำหนด เป็น test 10% และ train 90% และ เมื่อได้ data ที่จะนำไป train และ test เรียบร้อยแล้ว จึงเข้าสู่กระบวนการ train คือ ต้องกำหนด จำนวน Max Generation ต่อ 1 รอบการ train โดยเมื่อเข้าสู่กระบวนการแล้ว จะเริ่มจากการกำหนด ค่า weight ให้แต่ละ Edge ของ MLP โดยในนี้จะกำหนดแทนด้วยคำว่า gene โดยค่า gene นั้นจะน่ามาจากการ สุ่มค่า โดยกำหนดให้อยู่ในช่วง -1 ถึง 1 โดยเมื่อได้ ค่า gene ครบทุก edge แล้วก็จะนำไปเก็บไว้ที่ array ของ Chromosome นั้นเพื่อนำไปใช้เป็น weight ในการ train MLP ต่อไป โดยหลังจากได้ gene จนครบแล้ว จะเริ่มทำการ train โดย จะนำ dataset ที่เป็นข้อมูล train เข้ามา และ เริ่มต้นด้วยการหา fitness ของค่า Chromosome ใน generation ที่ 1 เสียก่อน โดยในการหา จะเป็นการนำ ค่า gene ของแต่ละ Chromosome ที่คำนวณไว้ก่อนหน้านี้ ไปเข้า MLP และ คำนวณและทำนายค่า โดยจะกำหนด ให้ออกมาเป็น 2 output Node โดยถ้า Node ไหนมีค่ามาก และ มีค่าของ Node นั้นตรงกับ Desire Output ก็จะเพิ่มค่า fitness ของ Chromosome นั้นทุก 1 ครั้งที่มีค่าตรง ทำแบบนี้ไปจนครบ ตามจำนวน Chromosome จะได้ ค่า fitness ทั้งหมด ของ ทุก Chromosome ใน generation ที่ 1 โดยหลังจากนั้น จะเข้าสู่การ ดังนี้

- **Selection**

  โดย การ Selection นั้น จะกำหนดให้สุ่มเลือกจาก Chromosome ใน Generation ก่อนหน้า โดยเลือกไว้จำนวน 90 % ของ จำนวนทั้งหมด

- **Crossover**

  โดย การ Crossover นั้น ก็จะใช้จำนวนของการ Cross เท่ากับ 90 % ของทั้งหมดเช่นกัน โดยหลักการจะเป็น การสุ่ม จาก Chromosome ที่ Select จากขั้นตอนก่อนหน้า โดยจะสุ่มออกมาจำนวน 2 Chromosome และ นำ gene ของทั้ง 2 มามใส่ใน gene ใหม่ที่ จะนำค่า gene ของตัวแรก และ ตัวที่ 2 อย่างละครึ่ง มารวมกัน

- **Mutate**

  โดย การ Mutate นั้น จะเป็นการสุ่มเลือกจาก Chromosome ที่ได้มาจาก การ Crossover โดยจะเลือกมาเพียง 40% ของจำนวนทั้งหมด โดยหลักการคือ สุ่มตำแหน่ง และ จำนวน โดย

หลังจากได้ ค่าแล้ว จะนำ ตำแหน่งที่ได้ ไปหา Mutate rate เพื่อดูว่าจะต้อง Mutate หรือไม่ โดยถ้าต้อง Mutate นั้นจะสุ่มเลขมาเพิ่มโดยอยู่ในช่วง -1 ถึง 1 และ นำค่าที่ได้มาใส่แทน gene เดิม

โดยเมื่อ ทำ Crossover เสร็จนั้น จะต้องทำการจัดเรียงตามค่า fitness เสียก่อนเพื่อให้สามารถเลือก การ Mutate จาก chromosome ที่มีค่า fitness ดี และท้ายที่สุดก็จะนำค่า chromosome ลำดับต้นๆที่มีค่า fitness ดีไปเข้าสู่ generation ต่อไป โดยจะวนทำจนครบตามจำนวน max generation โดยเมื่อจบกระบวนการแล้ว จะนำ Chromosome ที่มีค่า fitness ดีที่สุดใน generation ทั้งหมด มาทดสอบกับ test set ที่ได้แบ่งไว้ก่อนหน้านี้ และคำนวณหา ค่าความผิดพลาด และนำมาวิเคราะห์ต่อไป

ตัวอย่างโปรแกรม :

Structure คือ [30,15,15,2]

- 40 Max Generation
- 100 Chromosome
- Validation test 10 train 90

ตัวอย่าง Round ที่ 1

```
Generation : 1
------ best accurate : 87.71929824561404%
Generation : 2
------ best accurate : 91.61793372319688%
Generation : 3
------ best accurate : 91.81286549707602%
Generation : 4
------ best accurate : 91.81286549707602%
Generation : 5
------ best accurate : 91.81286549707602%
Generation : 6
------ best accurate : 91.81286549707602%
Generation : 7
------ best accurate : 91.81286549707602%
Generation : 8
------ best accurate : 91.81286549707602%
Generation : 9
------ best accurate : 91.81286549707602%
Generation : 10
------ best accurate : 91.81286549707602%
Generation : 11
------ best accurate : 91.81286549707602%
Generation : 12
------ best accurate : 91.81286549707602%
Generation : 13
------ best accurate : 91.81286549707602%
Generation : 14
------ best accurate : 91.81286549707602%
Generation : 15
------ best accurate : 91.81286549707602%
Generation : 16
------ best accurate : 91.81286549707602%
Generation : 17
------ best accurate : 91.81286549707602%
Generation : 18
------ best accurate : 91.81286549707602%
Generation : 19
------ best accurate : 91.81286549707602%
Generation : 20
------ best accurate : 91.81286549707602%
```

```
Generation : 21
------ best accurate : 91.81286549707602%
Generation : 22
------ best accurate : 91.81286549707602%
Generation : 23
------ best accurate : 91.81286549707602%
Generation : 24
------ best accurate : 91.81286549707602%
Generation : 25
------ best accurate : 91.81286549707602%
Generation : 26
------ best accurate : 91.81286549707602%
Generation : 27
------ best accurate : 91.81286549707602%
Generation : 28
------ best accurate : 91.81286549707602%
Generation : 29
------ best accurate : 91.81286549707602%
Generation : 30
------ best accurate : 91.81286549707602%
Generation : 31
------ best accurate : 91.81286549707602%
Generation : 32
------ best accurate : 91.81286549707602%
Generation : 33
------ best accurate : 91.81286549707602%
Generation : 34
------ best accurate : 91.81286549707602%
Generation : 35
------ best accurate : 91.81286549707602%
Generation : 36
------ best accurate : 91.81286549707602%
Generation : 37
------ best accurate : 91.81286549707602%
Generation : 38
------ best accurate : 91.81286549707602%
Generation : 39
------ best accurate : 91.81286549707602%
Generation : 40
------ best accurate : 91.81286549707602%

Testing accurate: 87.5%
```

การทดลองที่ 1 : การทดลองเปลี่ยนโครงสร้างของ Multilayer perceptron (Hidden layer , layer)

- 25 Generation
- 25 Chromosome

ผลการทดลองที่ 1 :

Input	Hidden 1	Hidden 2	Output	Error Average
30	1	-	2	17.85714285714286%
30	2	-	2	21.42857142857143%
30	3	-	2	12.5%
30	4	-	2	16.07142857142857%
30	5	-	2	14.285714285714292%
30	6	-	2	19.64285714285714%
30	7	-	2	14.285714285714292%
30	8	-	2	10.714285714285708%
30	9	-	2	10.714285714285708%
30	10	-	2	25.0%
30	15	-	2	19.64285714285714%
30	20	-	2	19.64285714285714%
30	25	-	2	25.0%
30	30	-	2	25.0%
30	1	1	2	23.214285714285708%
30	1	2	2	14.285714285714292%
30	1	3	2	14.285714285714292%
30	1	4	2	19.64285714285714%
30	1	5	2	14.285714285714292%
30	1	6	2	7.1428571428571 39%
30	1	7	2	10.714285714285708%
30	1	8	2	16.07142857142857%
30	1	9	2	25.0%

Input	Hidden 1	Hidden 2	Output	Error Average
30	1	10	2	23.214285714285708%
30	1	15	2	25.0%
30	2	1	2	16.071428571428 57%
30	2	2	2	23.214285714285708%
30	2	3	2	17.85714285714286%
30	2	4	2	28.57142857142857%
30	2	5	2	26.785714285714292%
30	2	6	2	7.142857142857139%
30	2	7	2	10.714285714285708%
30	2	8	2	12.5%
30	2	9	2	14.285714285714292%
30	2	10	2	25.0%
30	2	15	2	14.285714285714292%
30	3	1	2	14.285714285714292%
30	3	2	2	28.57142857142857%
30	3	3	2	25.0%
30	3	4	2	17.85714285714286%
30	3	5	2	14.285714285714292%
30	3	6	2	19.64285714285714%
30	3	7	2	10.714285714285708%
30	3	8	2	10.714285714285708%
30	3	9	2	12.5%
30	3	10	2	23.214285714285708%
30	3	15	2	19.64285714285714%
30	4	1	2	23.214285714285708%
30	4	2	2	16.071428571428 57%
30	4	3	2	16.071428571428 57%

Input	Hidden 1	Hidden 2	Output	Error Average
30	4	4	2	16.071428571428 57%
30	4	5	2	14.285714285714292%
30	4	6	2	19.64285714285714%
30	4	7	2	23.214285714285708%
30	4	8	2	17.85714285714286%
30	4	9	2	10.714285714285708%
30	4	10	2	12.5%
30	4	15	2	23.214285714285708%
30	5	1	2	12.5%
30	5	2	2	16.071428571428 57%
30	5	3	2	16.071428571428 57%
30	5	4	2	35.714285714285 71%
30	5	5	2	17.85714285714286%
30	5	6	2	21.428571428571 43%
30	5	7	2	30.357142857142 86%
30	5	8	2	16.071428571428 57%
30	5	9	2	12.5%
30	5	10	2	10.714285714285708%
30	5	15	2	25.0%
30	6	1	2	25.0%
30	6	2	2	19.64285714285714%
30	6	3	2	12.5%
30	6	4	2	23.214285714285708%
30	6	5	2	14.285714285714292%
30	6	6	2	10.714285714285708%
30	6	7	2	17.85714285714286%
30	6	8	2	23.214285714285708%

Input	Hidden 1	Hidden 2	Output	Error Average
30	6	9	2	17.85714285714286%
30	6	10	2	19.64285714285714%
30	6	15	2	19.64285714285714%
30	7	1	2	16.071428571428 57%
30	7	2	2	12.5%
30	7	3	2	10.714285714285708%
30	7	4	2	16.071428571428 57%
30	7	5	2	25.0%
30	7	6	2	28.571428571428 57%
30	7	7	2	14.285714285714292%
30	7	8	2	14.285714285714292%
30	7	9	2	7.142857142857139%
30	7	10	2	17.85714285714286%
30	7	15	2	16.071428571428 57%
30	8	1	2	12.5%
30	8	2	2	25.0%
30	8	3	2	17.85714285714286%
30	8	4	2	12.5%
30	8	5	2	16.071428571428 57%
30	8	6	2	10.714285714285708%
30	8	7	2	17.85714285714286%
30	8	8	2	10.714285714285708%
30	8	9	2	14.285714285714292%
30	8	10	2	14.285714285714292%
30	8	15	2	12.5%
30	9	1	2	14.285714285714292%
30	9	2	2	10.714285714285708%

Input	Hidden 1	Hidden 2	Output	Error Average
30	9	3	2	30.357142857142 86%
30	9	4	2	12.5%
30	9	5	2	7.142857142857139%
30	9	6	2	10.714285714285708%
30	9	7	2	23.214285714285708%
30	9	8	2	12.5%
30	9	9	2	21.42857142857143%
30	9	10	2	19.64285714285714%
30	9	15	2	17.85714285714286%
30	10	1	2	17.85714285714286%
30	10	2	2	23.214285714285708%
30	10	3	2	12.5%
30	10	4	2	14.285714285714292%
30	10	5	2	12.5%
30	10	6	2	16.07142857142857%
30	10	7	2	12.5%
30	10	8	2	21.42857142857143%
30	10	9	2	8.92857142857143%
30	10	10	2	17.85714285714286%
30	10	15	2	26.785714285714292%

วิเคราะห์ผลการทดลองที่ 1 :

จะเห็นว่าในการเปลี่ยนแปลงจำนวนและขนาดนของ Hidden layer และ Hidden Node นั้นแทบไม่มีความแตกต่าง หรือ มีค่า Error ต่างกันเพียงเล็กน้อยไม่มีนัยสำคัญอะไร แต่จะมีอยู่ในบางช่วงของ ส่วนที่มี 2 Hidden layer และ มี Hidden Node จำนวนกลางๆ ช่วง 7 – 9 นั้น จะมีค่า Error เฉลี่ยที่ต่ำกว่าช่วงอื่น และในกรณีที่มี 1 Hidden layer ก็เช่นกันในช่วงที่มี Hidden Node จำนวน 8 – 9 จะมีค่า Error ที่น้อยกว่าช่วงอื่น จึงสรุป ได้ว่า จำเป็นต้องประบให้ค่า Hidden layer และ จำนวน Hidden Node มีค่าไม่มากหรือน้อยจนเกินไป จะทำให้ได้ผลดีมากที่สุด โดยในการทดลองจะเห็นว่าผลลัพธ์ที่ดีที่สุด อยู่ที่ Neural 30 – 9 – 5 – 2 ซึ่งให้ผลลัพธ์คือ Error ที่มีค่าเพียง 7.142857142857139% เท่านั้น

การทดลองที่ 2 : ทดลองเพิ่ม - ลด จำนวน Chromosome และจำนวน generation

- Structure [30,8,2]

ผลการทดลองที่ 2 :

Chromosome-Count	Max-Generation	Error Average
10	25	35.71428571428571%
25	25	12.50000000000000%
50	25	13.214285714285708%
100	25	14.285714285714292%
200	25	21.42857142857143%
25	10	33.92857142857143%
25	25	7.142857142857139%
25	50	14.285714285714292%
25	100	19.64285714285714%
25	200	27.852857111243%

วิเคราะห์ผลการทดลองที่ 2 :

จะเห็นว่าการที่เพิ่มจำนวน Chromosome หรือว่า จำนวน Max Generation จะมีผลเสียต่อการ train ในกรณีที่มีจำนวนรอบ มากเกิน ซึ่งอาจทำให้เกิดการ Overfit ของการ train ได้ และถึงแม้ว่าจะใช้จำนวนน้อยๆ ก็อาจมีผลเสียได้เช่นกัน ซึ่งอาจจะเกิดจากการที่มีการ train น้อยเกินไป จึงทำให้ไม่เกิดการเปลี่ยนแปลงของ gene ใน Chromosome มากพอที่จะเปลี่ยนแปลงผลลัพธ์ ดังนั้นจึงควรกำหนด ค่า Chromosome และ Max Generation ใหมีค่ากลางๆ ไม่มากหรือไม่น้อยจบเกินไป เพื่อป้องกันปัญหาข้างต้น แ

Code :

```
import numpy as np
import random
import pdb
from functools import cmp_to_key

def load_txt(path):
    f=open(path, "r")
    if(path[-3:] == 'txt'):
        contents =f.readlines()

        dataset = []
        dic = {}

        for i in range(len(contents)):
            x = contents[i].split(",")
            for j in range(len(x)):
                if j == 0 :
                    dic.update({'ID_number': x[j]})
                elif j == 1 :
                    dic.update({'Diagnosis': x[j]})
                else :
                    if(j != len(x)-1):
                        dic.update({'features_'+str(j-1): x[j]})
                    else:
                        dic.update({'features_'+str(j-1): x[j][:-1]})
            dataset.append(dic.copy())

    return dataset

train = load_txt('./wdbc.data.txt')

class Chomosome:
    def __init__(self,genes):
        self.gene = genes
        self.fitness = 0

def partition(nums, low, high):

    pivot = nums[(low + high) // 2].fitness
    i = low - 1
    j = high + 1

    while True:
        i += 1
        while nums[i].fitness < pivot:
            i += 1
        j -= 1
        while nums[j].fitness > pivot:
            j -= 1
        if i >= j:
            return j

        nums[i], nums[j] = nums[j], nums[i]

def quick_sort(nums):
    # Create a helper function that will be called recursively
    def _quick_sort(items, low, high):
        if low < high:
```

```
            # This is the index after the pivot, where our lists are split
            split_index = partition(items, low, high)
            _quick_sort(items, low, split_index)
            _quick_sort(items, split_index + 1, high)

    _quick_sort(nums, 0, len(nums) - 1)

class GA :

    def __init__(self):
        self.layer = []
        self.initChomosome = []
        self.chomosomelist = []
        self.trainingset = []
        self.weightmin = -1
        self.weightmax = 1

    def getchomosome(self,layer,amount):
        self.layer = layer
        chromolen = 0

        for i in range(1,len(layer)):
            chromolen = chromolen +  (layer[i-1]*layer[i])

        for i in range(amount):
            gene = np.zeros(chromolen)
            for j in range(len(gene)):
                gene[j] = self.weightmin + (self.weightmax - self.weightmin)*random.random()
            self.initChomosome.append(Chomosome(gene))

    def trainchomosome(self,trainingset,maxgeneration):
        self.trainingset = trainingset
        self.chomosomelist = self.initChomosome.copy()
#         pdb.set_trace()
        self.computefitnessinlist(self.chomosomelist)
        for i in range(maxgeneration):
                selected = self.randomSelect(int(len(self.chomosomelist)*0.9))
                crossed = self.crossover(selected,int(len(self.chomosomelist)*0.9))
                pool = self.chomosomelist.copy()
#                 print(len(pool))
                pool.extend(crossed)
         #                 print(len(pool))
                        mutateAmount = int(len(self.chomosomelist)*0.4)
                        mutated = self.mutate(pool,mutateAmount)
                        quick_sort(pool)
                        pool = pool[::-1]
         #                 pdb.set_trace()
                        self.chomosomelist = pool[0:len(self.chomosomelist) - mutateAmount]
                        self.chomosomelist.extend(mutated)
                        print("Generation : "+str(i+1)+" best accurate :
    "+str(self.chomosomelist[0].fitness*100/len(trainingset))+"%")
                        quick_sort(self.chomosomelist)
                        self.chomosomelist = self.chomosomelist[::-1]


             def testchomosome(self,testset):
                 mlp = MLP(self.layer)
                 bestChomosome = self.chomosomelist[0]
                 mlp.initweight(bestChomosome)
                 bestChomosome.fitness = 0
                 for i in testset:
```

```
                    if(mlp.forward(i)):
                        bestChomosome.fitness += 1
                print("Testing accurate: "+str(bestChomosome.fitness*100/len(testset))+"%")
                return 100-(bestChomosome.fitness*100/len(testset))

            def computefitness(self,chomosome):
                mlp = MLP(self.layer)
                mlp.initweight(chomosome)
                chomosome.fitness = 0
                for idx,i in enumerate(self.trainingset) :
                    if mlp.forward(i) :
                        chomosome.fitness += 1
        #         print(chomosome.fitness)

            def computefitnessinlist(self,chomosomelist):
                j = 0
        #         pdb.set_trace()
                for i in chomosomelist:
        #             print(str(j) + " : ",end='')
                    j = j+1
                    self.computefitness(i)

            def randomSelect(self,amount):
                selected = []
                for i in range(amount):
                    selec = random.randrange(len(self.chomosomelist))
        #             print('select : '+str(i))
                    selected.append(self.chomosomelist[selec])
                return selected

            def crossover(self,selected,amount):
                crossed = []
                for i in range(amount):
                    i1 = random.randrange(len(selected))
                    i2 = random.randrange(len(selected))
                    daddy = selected[i1]
                    mommy = selected[i2]
                    chomosomelen = len(selected[0].gene)
                    halflen = int(chomosomelen/2)
                    gene = np.zeros(chomosomelen)
                    for j in range(halflen):
                        gene[j] = daddy.gene[j]
                    for j in range(halflen,chomosomelen):
                        gene[j] = mommy.gene[j]
                    crossed.append(Chomosome(gene))
                self.computefitnessinlist(crossed)
                return crossed

            def mutate(self,pool,amount):
                    mutated = []
                    index = random.sample(range(len(pool)), amount)
                    k = 0
                    for i in index:
                        gene = pool[i].gene.copy()
                        mutateRate = int(len(pool[i].gene)*0.9)
                        for j in range(mutateRate):
                            w = random.randrange(len(pool[i].gene))
                            gene[w] = self.weightmin + (self.weightmax - self.weightmin)*random
    .random()
                            if(gene[w] > self.weightmax):
                                gene[w] = self.weightmax
```

```
                            if(gene[w] < self.weightmin):
                                gene[w] = self.weightmin
                        mutated.append(Chomosome(gene))
                        self.computefitness(mutated[k])
                        k = k+1
                    return mutated

        class MLP:
            def __init__(self,structure_layer):
                self.maxNode = 0
                self.structure = structure_layer

                for i in structure_layer:
                    if self.maxNode < i :
                        self.maxNode = i

                self.weight = np.zeros((len(structure_layer)-1,self.maxNode,self.maxNode))
        #         print(self.weight)

            def initweight(self,chomosome):
                count = 0
        #         print(self.weight)
                for i in range(len(self.structure)-1):
                    for j in range(self.structure[i]):
                        for k in range(self.structure[i+1]):
                            self.weight[i][j][k] = chomosome.gene[count]
        #                     print(self.weight[i][j][k])
                            count = count+1
        #         print(self.weight)
        #         pdb.set_trace()

            def forward(self,data):
                y = np.zeros((len(self.structure),self.maxNode))
        #         print(list(data.values())[2:])
                y[0] = list(data.values())[2:]
                for i in range(1,len(self.structure)):
                    for j in range(self.structure[i]):
                        v = 0
                        for k in range(self.structure[i-1]):
                            v = v + y[i-1][k]*self.weight[i-1][k][j]
                        y[i][j] = np.tanh(v)
        #         print("1 :" +str(y[len(self.structure)-1]))
        #         print("1 :" +str(y[len(self.structure)-1][0])+" 2 :
    "+str(y[len(self.structure)-1][1]))
        #         print(data['Diagnosis'])
                if(data['Diagnosis'] == 'B'):
                    if(y[len(self.structure)-1][0] < y[len(self.structure)-1][1]):
                        return True
                    else:
                        return False
                else:
                    if(y[len(self.structure)-1][0] > y[len(self.structure)-1][1]):
                        return True
                    else:
                        return False

            def sigmoid(v):
                return 1/(1+np.exp(-v))

        genetic = GA()
        structure = [30,15,15,2]
```

```
        genetic.getchomosome(structure,100)
        # genetic.trainchomosome(train,5)
        eav = 0
        for c in range(10):
            print("------------------------- # Round "+str(c)+' ------------------------- ')
            i = int(c*len(train)*0.1)
            test = train[i:int(i+int(len(train)*0.1))]
            traingset = train.copy()
            del traingset[i:i+int(len(traingset)*0.1)]
            genetic.trainchomosome(traingset,50)
            eav += genetic.testchomosome(test)
        print("Error avg : "+str(eav/10)+'%')
```