**Sukrit Wongariyakavee**
**H00288398**

# Bio Inspired Computation
# Coursework 1 - BIC Techniques

## Part I.A: Evolutionary Algorithms

### Introduction

In this coursework I selected "Python" to be my programming language as I have zero background in computer science.

### Solution

**Data Cleaning**

I perform a data sanitization by removing duplicate from the dataset which resulted in 452 less instances. This left me with 528 cities in total (46.12% decrease). Here I generate .csv file of 10 cities, 20 cities, 100 cities until the full dataset in order to test with different size of data.

**Genetic Algorithm implementation**

1. I create a tsp cost function (Approximation of the great circle distance), distance measure function (simple $x$ and $y$ coordinate for faster computation time and for visualization purpose) and a mutation function that swap the sequence of the random solution.
2. I define all parameters required (population size, generation, tournament size) and create a random solution and put it in an array of population. Next, I compute each population fitness and put the result in a same order as the population.
3. I randomly select the random solution by the amount of the tournament size and put it in an array of tournament and calculate the fitness of each random solution in the tournament.
4. I mutate the best random solution in the tournament and replace the population with worst fitness with it.

**Setting**

On all combination of the number of population, tournament size, and no. of generation the normal mutation process couldn't achieve the good result

**Result(tsp_ga.py)**

1. I test it with small dataset of 20 cities.
2. The solution could not achieve the optimal solution. If the optimal solution is achieved, the solution only came from the random solution on step "3" not the mutation process.

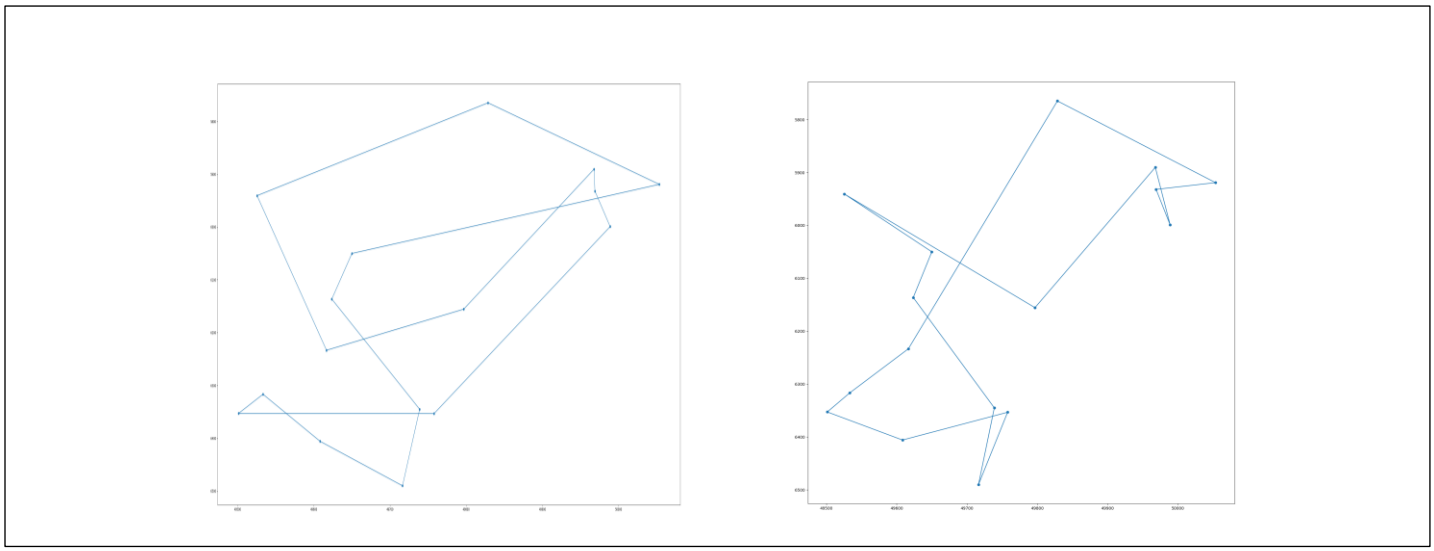


*Figure 1, 20 cities solution*

**Improvement (tsp_with_improvement.py)**

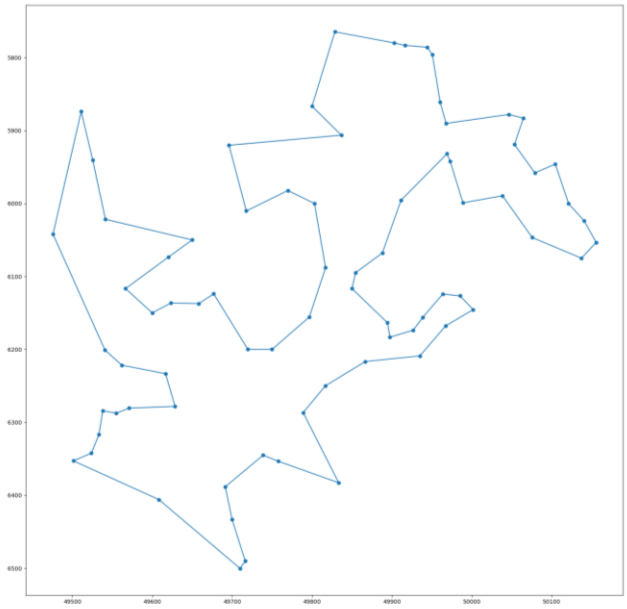
1. I try to improve the genetic algorithm by changing the mutation process to a 2-opt and check for better result

**Setting**

1. The only option to set the algorithm is the number of generation “n”
2. For small dataset only with 10-20 generation the optimal solution could easily achieve.
3. for large dataset number of generation must be more than 20 in order to get a good quality result.

**Result**

1. The algorithm could not achieve a global optimal solution. However, the result is good enough after I restructure everything
2. I test the algorithm with 100 cities



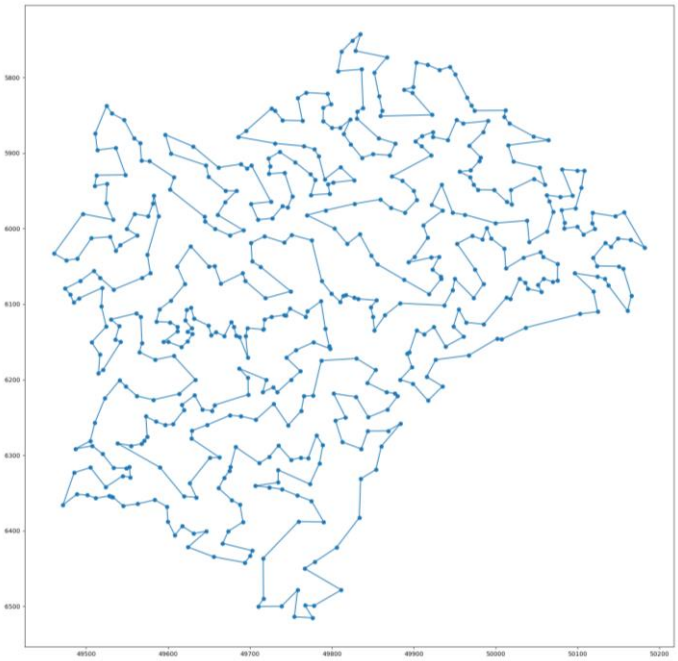
```
generation:  1  [f(s):  inf ]
generation:  2  [f(s):  4538.325650686199 ]
generation:  3  [f(s):  4360.6472651688855 ]
generation:  4  [f(s):  4360.6472651688855 ]
generation:  5  [f(s):  4360.6472651688855 ]
generation:  6  [f(s):  4360.6472651688855 ]
generation:  7  [f(s):  4194.4501380858055 ]
generation:  8  [f(s):  4194.4501380858055 ]
generation:  9  [f(s):  4194.4501380858055 ]
generation:  10  [f(s):  4194.4501380858055 ]

[2, 41, 6, 64, 38, 1, 5, 34, 47, 39, 31, 51, 72, 54, 48, 32, 17, 45, 67, 63, 18, 42, 25, 68, 14, 36, 15, 37, 53, 28, 23, 16, 55, 35, 73, 10, 58, 59, 11, 61, 66, 40, 62, 70, 4, 43, 49, 75, 57, 52, 50, 24, 3, 20, 21, 74, 44, 12, 30, 29, 71, 13, 19, 0, 22, 65, 33, 69, 8, 56, 60, 46, 27, 7, 26, 9]

time =  19.889024019241333
total_distance = 4194.4501380858055
```

*Figure 2, Result from tsp_with_improvement.py*

3. Testing with full dataset (I only run it with 1 generation as it around 15 mins per generation)



```
Shortest Route
generation:  1  [f(s):  inf ]
[519, 312, 356, 365, 114, 445, 69, 354, 33, 515, 520, 224, 418, 397, 421, 7, 128, 514, 472, 409, 26, 420, 457, 107, 145,
9, 92, 510, 526, 294, 144, 80, 367, 268, 317, 2, 430, 201, 181, 523, 250, 215, 64, 243, 385, 157, 38, 359, 500, 393, 170,
299, 171, 136, 402, 506, 369, 290, 448, 464, 450, 47, 216, 446, 139, 343, 465, 463, 1, 5, 108, 273, 383, 271, 6, 41, 221,
454, 476, 340, 381, 382, 76, 248, 167, 427, 34, 361, 438, 490, 337, 146, 502, 508, 357, 111, 39, 237, 414, 415, 225,
364, 492, 31, 389, 88, 79, 187, 276, 411, 351, 143, 380, 311, 376, 412, 205, 213, 56, 121, 234, 491, 306, 373, 106, 27,
46, 131, 60, 407, 236, 219, 172, 95, 209, 8, 335, 86, 184, 152, 473, 316, 423, 21, 521, 287, 220, 20, 228, 97, 518, 178,
94, 3, 281, 120, 159, 226, 222, 498, 127, 374, 155, 89, 259, 320, 255, 485, 345, 481, 217, 74, 348, 44, 344, 507, 218,
84, 126, 261, 425, 12, 435, 297, 210, 30, 477, 449, 251, 332, 384, 452, 232, 71, 497, 244, 13, 468, 284, 235, 300, 98,
151, 334, 517, 474, 57, 291, 456, 471, 487, 270, 447, 501, 24, 451, 366, 42, 400, 18, 315, 408, 458, 371, 328, 174, 327,
330, 285, 303, 522, 269, 15, 161, 85, 484, 214, 293, 142, 53, 394, 289, 479, 242, 134, 431, 432, 211, 326, 162, 48, 32,
230, 212, 154, 17, 278, 467, 298, 63, 511, 67, 204, 256, 494, 51, 387, 45, 72, 54, 180, 133, 310, 263, 274, 319, 466,
323, 28, 272, 505, 23, 396, 333, 488, 429, 16, 377, 322, 100, 183, 203, 486, 240, 55, 166, 113, 329, 470, 117, 513, 35,
360, 336, 207, 417, 73, 433, 442, 267, 401, 238, 37, 416, 233, 192, 460, 375, 444, 68, 140, 301, 109, 436, 241, 379,
101, 196, 258, 25, 200, 179, 229, 227, 50, 52, 110, 363, 441, 231, 350, 349, 208, 406, 341, 304, 252, 314, 264, 239,
527, 185, 499, 61, 102, 362, 11, 462, 190, 194, 119, 14, 36, 410, 403, 292, 295, 87, 483, 253, 10, 469, 440, 58, 59, 173,
104, 195, 480, 434, 82, 66, 40, 355, 62, 70, 391, 475, 512, 482, 254, 516, 489, 478, 124, 4, 125, 352, 164, 331, 392,
286, 153, 90, 347, 206, 280, 141, 49, 405, 388, 249, 325, 43, 188, 245, 277, 123, 182, 202, 422, 265, 175, 439, 115,
424, 509, 149, 309, 168, 158, 246, 455, 156, 495, 147, 399, 137, 148, 197, 318, 324, 419, 91, 191, 138, 93, 305, 116,
368, 83, 453, 193, 75, 338, 77, 370, 279, 395, 296, 358, 283, 99, 378, 372, 122, 186, 169, 307, 308, 443, 339, 177, 199,
260, 163, 129, 135, 275, 22, 176, 198, 112, 504, 413, 0, 426, 223, 404, 118, 65, 302, 105, 390, 78, 19, 150, 160, 262,
496, 81, 96, 437, 132, 103, 266, 524, 321, 189, 461, 313, 428, 525, 459, 398, 493, 165, 346, 247, 386, 353, 130, 282,
503, 257, 288, 29, 342]
time =  1124.889024019241333
total_distance = 11417.741154165311
```

*Figure 3, Full dataset on tsp_with_improvement*

**Summary**

*Table 1, result from varying no. of generation (Test with 100 cities)*

| No. of generation | Total distance | Time (Second) |
|---|---|---|
| 1 | 4400.47 | 2.22 |
| 3 | 4317.70 | 10.06 |
| 5 | 4247.86 | 12.44 |
| 10 | 4201.39 | 21.72 |
| 20 | 4176.88 | 50.71 |
| 50 | 4173.33 | 129.78 |

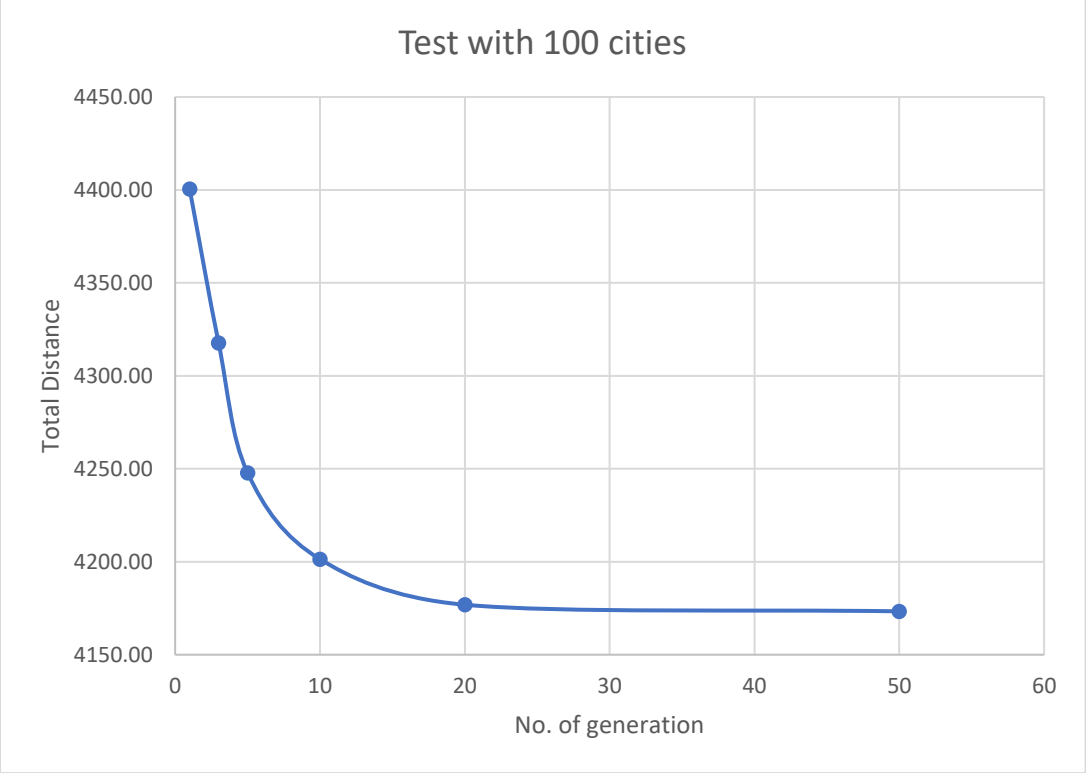



*Figure 4, No. of generation vs Total distance*

# Part I.B: Artificial Neural Network

## Introduction

In this part I also selected "Python" to be my programming language. I selected $y = x^3$ and the sigmoid function $\frac{1}{1+e^{(-x)}}$ as my activation function.

## Solution(ANN.py)

**Dataset**

I create an array of ten $X$ value by random the number in range of $[-20, 20]$ and compute the value of $y = x^3$ from $X$.

**Weight**

I create two arrays of weight, $W1$ for the hidden layer and $W2$ for weight at the output layer

**ANN implementation**

1. Define parameter: learning rate, no. of neuron in the hidden layer
2. Initialize the dataset and the weight
3. Define sigmoid function, the derivative of the sigmoid function
4. Create a forward propagation in a loop of 10 iteration
5. Passing the dataset to the input layer *($layer\_0$)*
6. Assign the hidden layer $(layer\_1)$ to be a sigmoid function a product of $layer\_0$ and $W1$ matrix multiplication
7. Assign the output layer $(layer\_2)$ to be a sigmoid function a product of $layer\_1$ and $W2$ matrix multiplication
8. Compute the error from the output layer and the $Delta$ value of the second layer
9. Use the output layer $Delta$ to compute the hidden layer $Delta$ as well as the error from the hidden layer
10. Update the weight $W1$ and $W2$
11. Repeat for 10 iterations

**Result**

| input -> X | y | after training -> Predict |
|---|---|---|
| [ -3.23221942] | [ -33.76777982] | [ 1. ] |
| [ 7.40878002] | [ 406.66809305] | [ 0.00208087] |
| [-11.82191001] | [-1652.20125392] | [ 1. ] |
| [ 15.12469746] | [ 3459.87244703] | [ 0.00204454] |
| [-18.90449627] | [-6756.08848642] | [ 1. ] |
| [ 6.81870041] | [ 317.033261 ] | [ 0.00210236] |
| [ -3.30780791] | [ -36.19268828] | [ 1. ] |
| [ 2.34759314] | [ 12.93804014] | [ 0.00507351] |
| [-14.38452246] | [-2976.36607459] | [ 1. ] |
| [-12.07594044] | [-1761.01431675] | [ 1. ] |

*Figure 5, input(X), y, prediction*

```
Training with Learning rate: 0.01
Error after 1 iterations:1741.20054263
Error after 2 iterations:1741.21471719
Error after 3 iterations:1741.25870472
Error after 4 iterations:1741.81165118
Error after 5 iterations:1741.81211917
Error after 6 iterations:1741.81244061
Error after 7 iterations:1741.8126759
Error after 8 iterations:1741.81285595
Error after 9 iterations:1741.81299838
Error after 10 iterations:1741.81311397
```

*Figure 6, error value recorded*

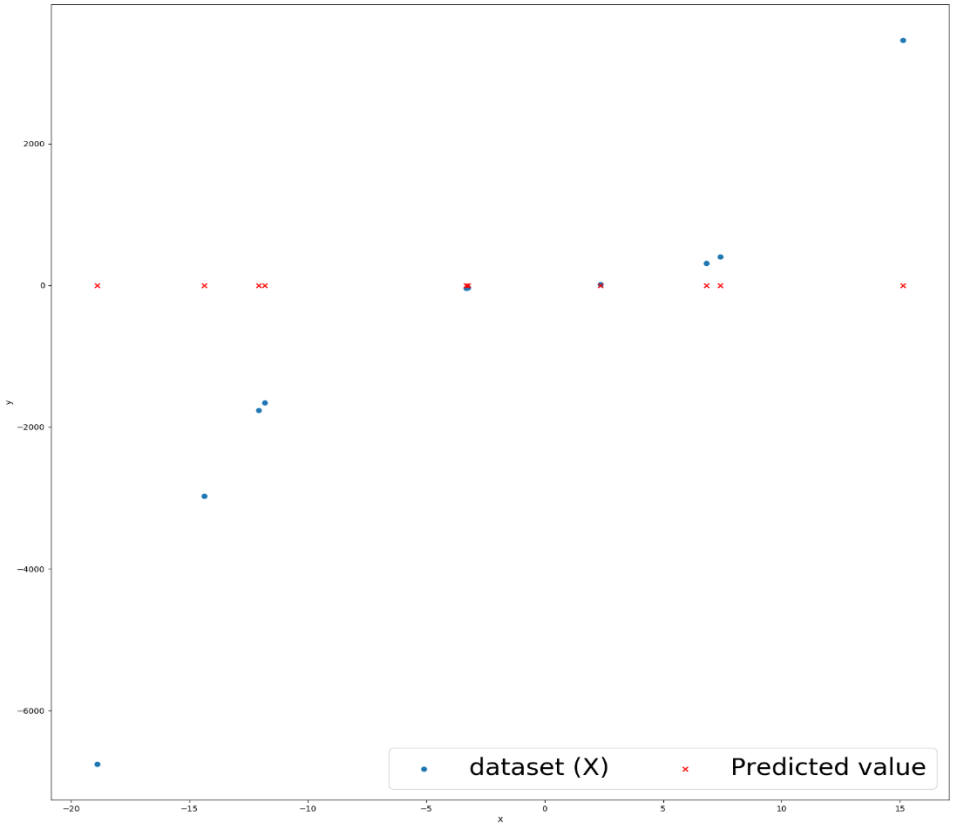



*Figure 7, Comparison of the result from the prediction with the actual value*

# Part II

## Introduction

In this part, we will explore more options to solve the traveling salesman problem and other type of traveling salesman problem by surveying the most recent literature. This report will cover, other methods to optimize the normal traveling salesman problem, traveling salesman problem with an arbitrary neighbourhood, asymmetric traveling salesman problem (ATSP).

## Other combinatorial optimization problem (Other type of TSP)

### Traveling Salesman with an arbitrary neighbourhood

From figure 8, TSP with arbitrary neighbourhoods could be illustrated as TSP that consists of scattered neighbourhoods with random shapes and sizes. Instead of cities, each neighbourhood must be visited only once, start and end in the exact same neighbourhood.

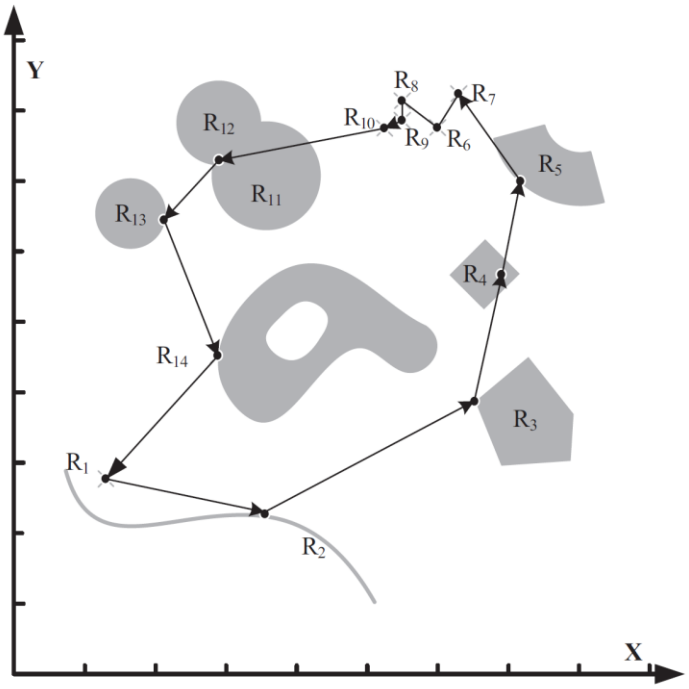



*Figure 8, (Yang et al., 2018) An illustration of TSP with arbitrary neighbourhoods.*

For this type of TSP Yang et al. (2018) proposed the "Hybrid algorithm based on LDI-WPSO and GA". This hybrid algorithm is inspired by how the flocks of birds and schools of fish behave. From figure 9, the hybrid algorithm consists of two main loop where LDI-WPSO take care of the continuous optimization and GA take care of the discrete optimization. In the hybrid algorithm, the LDI-WPSO is used to create the path-point on the neighbourhood area and improve it after each iteration, while GA will be use to find the optimal path for the TSP. This could confirm a good solution with short computation time.

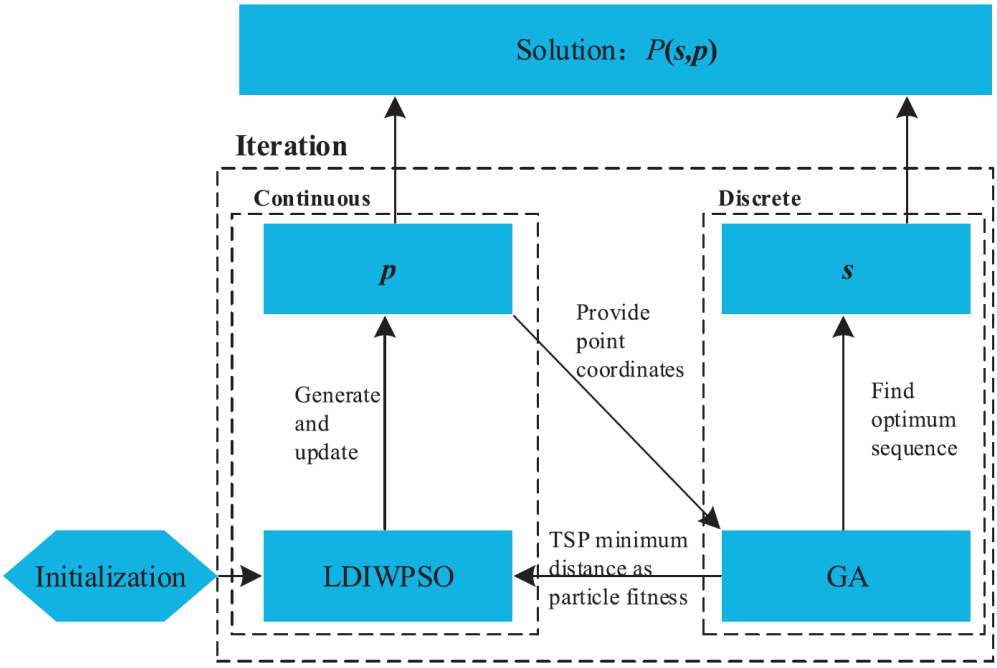



*Figure 9, (Yang et al., 2018) Outline of hybrid algorithm for TSP with arbitrary neighbourhoods.*

## Asymmetric traveling salesman problem (ATSP)

Being one of the most well-known combinatorial optimization, the ATSP could be stated as a complete directed graph $G = (V, A)$, where $V$is the vertex set and $A$ is the arc set. The arc set have a nonnegative cost, find the minimum cost circuit in G that pass each vertex only once.

The algorithm proposed by Nagata and Soler (2012) is the edge assembly crossover (EAX) operator. The EAX generate child solution by combining the arc from two parents and adding few short arc to it.

### EAX illustration

1. Given a pair of solution to be set as parent $P_A$ and $P_B$, Let $G_{AB}$ be the directed graph from both parent where $G_{AB} = (V, E_A \cup E_B \setminus E_A \cap E_B)$, where $E_A$ and $E_B$ is a set of arc that are in $P_A$ and $P_B$.
2. Separate all arc in $G_{AB}$ into AB-cycle in the way that the arcs from $E_A$ and $E_B$ are connected in the opposite direction.
3. Create E-set using defined rule set
4. Generate initial solution from $P_A$ by removing the arcs of $E_A$ and adding $E_B$ in the E-set.
5. Connect subtours from the solution in step 4 and create a valid tour

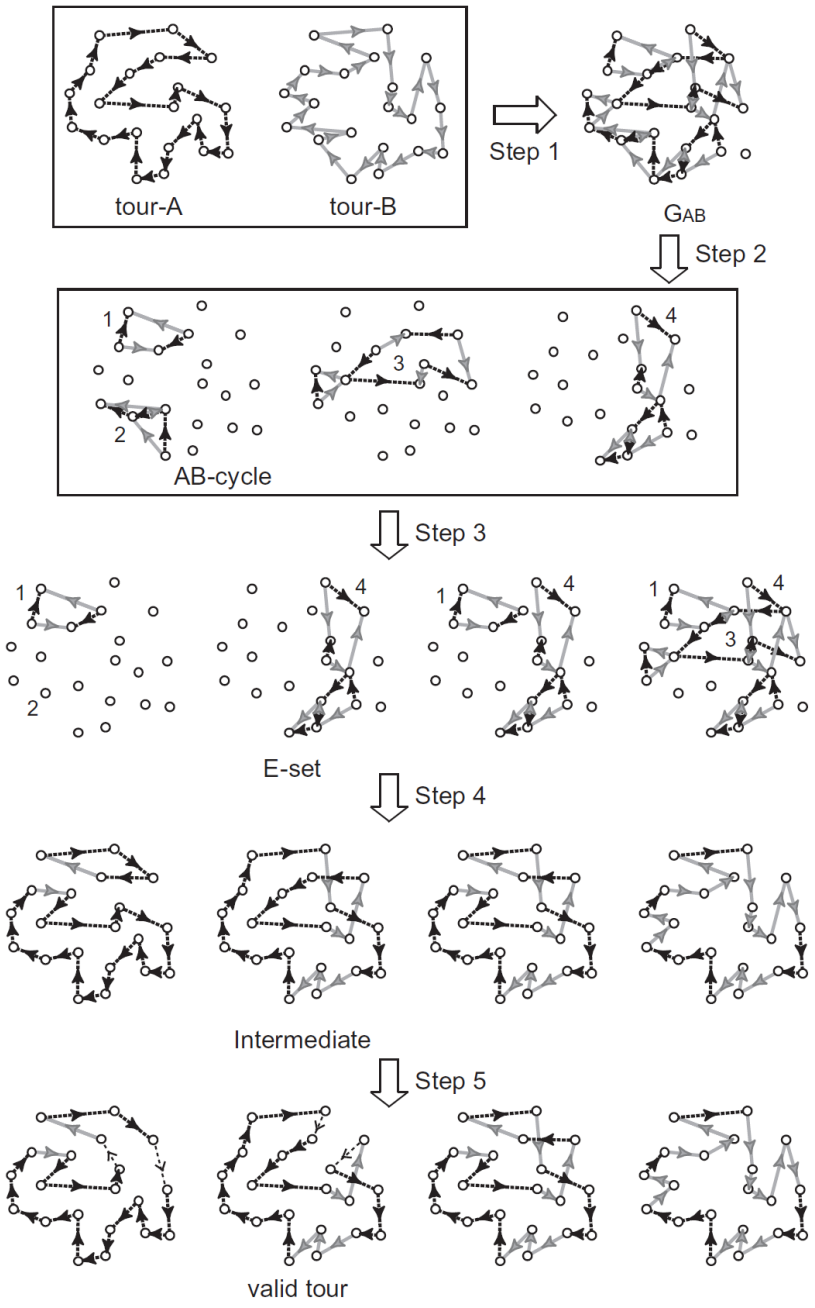



*Figure 10, (Nagata and Soler, 2012), Illustration of the EAX for the ATSP.*

## Optimization of symmetric TSP (Traditional TSP)

### Multi-offspring genetic algorithm

In the typical GA for TSP, the mutation or crossover restrict the number of child solution produce from 2 offspring to be equal of less than the number of parent solution. However, it is possible to overcome those restriction to produce a better child solution for the problem.

**Steps**

1. Create the initial population of size $n$ and sort it in ascending order according to their fitness.
2. Select select q-elitist members with best ranking among $n$ parents.
3. Using crossover operator to generate $2n$ number of children solution.
4. Create a new population consisting of $2n$ children solution and the q-elitist in 2nd step.
5. Sort the members of new population in ascending order according to their fitness.
6. Select q-elitist members in the new population.
7. Using mutation operator to mutate the $2n$ children that generated by the crossover.
8. Generate the new population using the $2n(p_m)$ mutated children, unmutated children $2n(1-p_m)$, and the q-elitist chosen in step 6th.
9. Sort the members of the new population in step 8 in ascending order according their fitness.
10. If the stopping criterion is met, output the best solution and terminate. If not continue step 11th
11. Select $n$ members with best fitness within the last population and go to step 2nd.

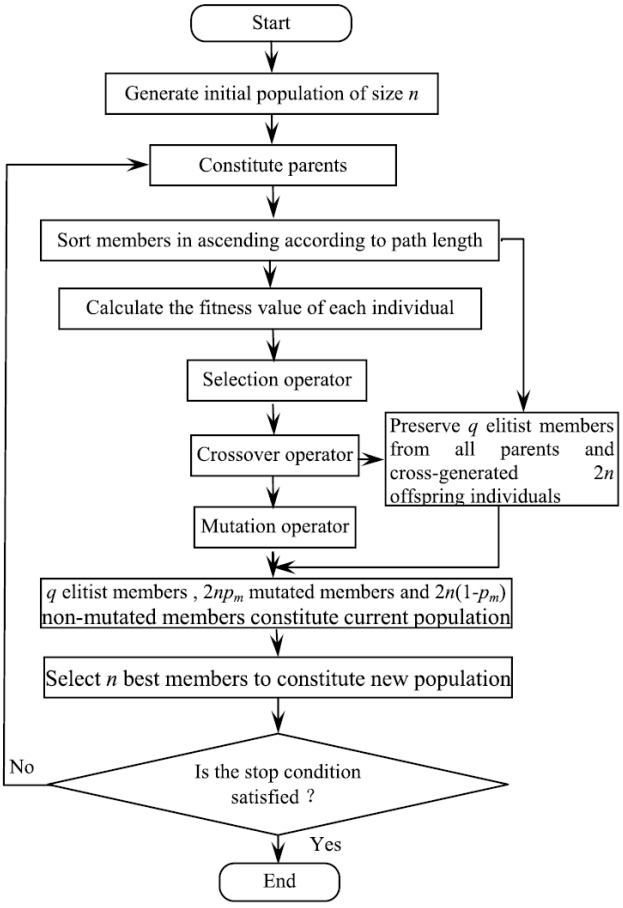



*Figure 11, (Wang et al., 2016) Flow diagram for Multiple offspring GA*

**Crossover in 3rd step**

Using the roulette wheel method to select crossover members by

$$P_i = \frac{F(X_i')}{\sum_{i=1}^{n} F(X_i')} \quad (1)$$

Let $PP_0 = 0$

$$PP_i = \sum_{i=1}^{n} P_i \, , i = 1, 2, \ldots . , n \quad (2)$$

The roulette wheel is rotated up to $n$ times, and a random number $\eta_k \in (0, 1)$ is generate at each rotation until

$PP_{i-1} \leq \eta_k \leq PP_i$ is satisfied.

When satisfied, the $i$ member is selected to apply crossover using the roulette wheel method above to pair the members together according to their fitness (highest fitness first)

**Create two children ($2n$) from the population of size $n$**

From 2 parents generate 4 children by performing 2-point crossover by:

Parent

$$A = 3\,4\,6\,8\,1\,2\,9\,7\,5$$

$$B = 2\,7\,1\,9\,5\,3\,6\,4\,8$$

1. Randomly generate the position of crossover and apply it to the parent. Keep the solution in the middle between 2-point generated randomly

$$A = 3\,4\,6\,|\,8\,1\,2\,9\,|\,7\,5$$

$$B = 2\,7\,1\,|\,9\,5\,3\,6\,|\,4\,8$$

2. Reposition the right end of the solution and put it in front of the left end of the solution.

$$A = 7\,5\,3\,4\,6\,|\,8\,1\,2\,9$$

$$B = 4\,8\,2\,7\,1\,|\,9\,5\,3\,6$$

3. Delete the elements in $B$ that contain a same elements in the middle part of $A$. Repeat he process by exchanging both $A$ and $B$

$$A = 7\,4\,8\,1\,2$$

$$B = 4\,7\,5\,3\,6$$

4. Let $k$ be the number of elements from the 3rd step, move the first $k$ elements of the first member obtained in step 3.

$$C1 = 8\,1\,2\,|\,9\,5\,3\,6\,|\,7\,4$$

$$C2 = 5\,3\,6\,|\,8\,1\,2\,9\,|\,4\,7$$

5. From step 2, exchange section 1 and 2

$$AA = 8\,1\,2\,9\,3\,4\,6\,7\,5$$

$$BB = 9\,5\,3\,6\,2\,7\,1\,4\,8$$

6. Find section 3 element in $AA$ and delete the one in $BB$

$$C3 = 1\,2\,9\,|\,3\,6\,7\,5\,|\,4\,8$$

$$C4 = 9\,3\,6\,|\,2\,1\,4\,8\,|\,7\,5$$

From this method, crossover will give us 4 children from 2 parents which could significantly increase in the number of population on each iteration. With more number of children there are more chance that we could create a good quality population which will greatly reduce computation time and less iteration. As a result, better solution could be achieve efficiently.

## References


1. Zhao Yang, Ming-Qing Xiao, Ya-Wei Ge, De-Long Feng, Lei Zhang, Hai-Fang Song, Xi-Lang Tang. (2018). A double-loop hybrid algorithm for the traveling salesman problem with arbitrary neighbourhoods. *In European Journal of Operational Research*. 265 (Issue 1), p65-80.

2. Yuichi Nagata, David Soler. (2012). A new genetic algorithm for the asymmetric traveling salesman problem. *In Expert Systems with Applications*. 39 (Issue 10), p8947-8953.

3. Chiung Moon, Jongsoo Kim, Gyunghyun Choi, Yoonho Seo. (2002). An efficient genetic algorithm for the traveling salesman problem with precedence constraints. *In European Journal of Operational Research*. 140 (Issue 3), p606-617.

4. László T. Kóczy, Péter Földesi, Boldizsár Tüű-Szabó. (2017). *Enhanced discrete bacterial memetic evolutionary algorithm - An efficacious metaheuristic for the traveling salesman optimization In Information Sciences, 2017, , ISSN 0020-0255.* Available: http://www.sciencedirect.com/science/article/pii/S0020025517309866. Last accessed 30th Oct 2017.

5. Jiquan Wang, Okan K. Ersoy, Mengying He, Fulin Wang. (2016). Multi-offspring genetic algorithm and its application to the traveling salesman problem. *In Applied Soft Computing*. 43, p415-423.

6. Semya Elaoud, Jacques Teghem, Taïcir Loukil. (2010). Multiple crossover genetic algorithm for the multiobjective traveling salesman problem. *In Electronic Notes in Discrete Mathematics*. 36, p939-946.

# Annex

## Part I.A: Evolutionary Algorithms

Tsp_ga.py

```
import numpy as np
import pandas as pd
import random
import math
import matplotlib.pyplot as plt

list_of_cities = []

##########################    Parameters    ###################################
mut_prob = 0.4

# Number of generations to run for
generations = 1000

# Population size of 1 generation (RoutePop)
pop_size = 10

# Size of the tournament selection.
tournament_size = 2

# Dataset resize into 10 cities to ful list of 980 cities
######################
# File name
# test_10.csv  generation = 10, popsize = 2000, tour_size = 10
# test_20.csv
# test_50.csv
# test_100.csv
# test_200.csv
# test_450.csv
# test_980.csv
######################
csv_name = 'test_10.csv'

df = pd.read_csv(csv_name, header = 0)
df = df.set_index(df['city'])
df = df.drop('city', 1)
df_drop = df.drop_duplicates(subset = 'i', keep = 'first')

dataset = df_drop.as_matrix(columns = None)
################################################################################
def main():

#    print(df.head())
#    print(dataset)

    print("################################################################################")
    print("No. of Random Solution = %a" %pop_size, "\n")

    soln = []
    fitness = []

    for a in range(0, pop_size):
        order =  list(range(dataset.shape[0]))
        random.shuffle(order)
        soln.append(order)
        length = calc_length(dataset, order)
        fitness.append(length)

```

```
#         print(a, soln[a], "{f(s) = ", fitness[a], "}")

    print("#########################################################################")
    for x in range(0, generations):
#         print("#########################################################################")
#         print("Tournament\n")
        Tour_result = []
        Tour_fitness = []
        Tour_result = random_subset(soln, tournament_size)

        for a in range(0, len(Tour_result)):
            Tour_fit = calc_length(dataset, Tour_result[a])
            Tour_fitness.append(Tour_fit)

        mutate_soln = []
        high_f, high_index = h_fit(Tour_fitness)
#         print(Tour_result[high_index], "{f(s) = ", Tour_fitness[high_index], "}")

        mutate_soln, mutate_length = mutate(high_index, Tour_result, dataset)


        low_fit, low_index = l_fit(fitness)

        if fitness[low_index] > mutate_length:
            soln[low_index] = mutate_soln
            fitness[low_index] = mutate_length

        best_fit, best_index = h_fit(fitness)

#         print("\nBest_Solution")
        print("Gen",x+1,soln[best_index], "{f(s) = ", best_fit, "}")

    return soln[best_index], fitness[best_index]

###############################################################################
#GA Stuff
#1-point Crossover
def Crossover_1P(i, j, soln, fitness, dataset):
    """
    i: parent 1
    j: parent 2
            """
            child_soln = []
            x = random.randint(0,len(soln[i])-1)

            for m in range(0, x):
                child_soln.append(soln[i][m]) # set the values to eachother

            for n in range(x, 10):
                child_soln.append(soln[j][n]) # set the values to eachother

            print("Crossover position = %x" %x)
            child_length = calc_length(dataset, child_soln)
        #    print(child_soln, "{f(s) = ", child_length, "}")
            return child_soln, child_length

        ###############################################################################
        #Mutation
        def mutate(i, soln, dataset):
            '''''
            Route() --> Route()
            Swaps two random indexes in route_to_mut.route. Runs k_mut_prob*100 % of the time
            '''
        #    x = "no"
            mut_soln = soln[i]
```

```
            mut_1 = []
            mut_2 = []
            mut_pos1 = random.randint(0,len(soln[i])-1)
            mut_pos2 = random.randint(0,len(soln[i])-1)
        #    chance = random.random()
        #    set to always mutate
            chance = 1

            while mut_pos1 == mut_pos2:
                mut_pos2 = random.randint(0,len(soln[i])-1)


            # k_mut_prob %
            if chance > mut_prob:
                if mut_pos1 == mut_pos2:
        #            x = "no"

                    mut_length = calc_length(dataset, soln[i])
                    mut_soln = soln[i]
        #            print("mutate = ", x)
        #            print(mut_soln, "{f(s) = ", mut_length, "}")
                    return mut_soln, mut_length

            # Otherwise swap them:
                else:
        #            x = "yes"
                    mut_soln = soln[i]
                    mut_1 = soln[i][mut_pos2]
                    mut_2 = soln[i][mut_pos1]

                    mut_soln[mut_pos1] = mut_1
                    mut_soln[mut_pos2] = mut_2

            mut_length = calc_length(dataset, mut_soln)
            return mut_soln, mut_length

        ###############################################################################
        #fittest
        #highest
        def h_fit(fitness):
            h_f = min(fitness)
            index_hf = fitness.index(min(fitness))

            return h_f, index_hf

        #lowest
        def l_fit(fitness):
            l_f = max(fitness)
            index_lf = fitness.index(max(fitness))

            return l_f, index_lf

        ###############################################################################
        #Tournament
        def random_subset(soln, tournament_size):
            result = []
            N = 0

            for item in soln:
                N += 1
                if len(result) < tournament_size:
                    result.append(item)
                else:
                    s = int(random.random() * N)
                    if s < tournament_size:
```

```
                    result[s] = item

        return result

    ###############################################################################
    #Calculate length
    def calc_length(dataset, path):
        length = 0
        for i in list(range(len(path))):
            length += distance(dataset[path[i-1]], dataset[path[i]])
    #        length += tsp_cost(path[i-1], path[i], dataset)
        return length

    #Distance square
    def distance(c1, c2):
        t1 = c2[0] - c1[0]
        t2 = c2[1] - c1[1]

        return math.sqrt(t1**2 + t2**2)

    ###############################################################################
    def tsp_cost(i, j, dataset):
        pi = 3.1415926535897932384626

        lat_i = (pi*dataset[i][0])/180
        lat_j = (pi*dataset[j][0])/180
        long_i = (pi*dataset[i][1])/180
        long_j = (pi*dataset[j][1])/180

        q1 = math.cos(lat_j)*math.sin(long_i-long_j)
        q3 = math.sin((long_i-long_j)/2.0)
        q4 = math.cos((long_i-long_j)/2.0)
        q2 = math.sin(lat_i+lat_j)*q3*q3 - math.sin(lat_i-lat_j)*q4*q4
        q5 = math.cos(lat_i-lat_j)*q4*q4 - math.cos(lat_i+lat_j)*q3*q3

        return (6378388.0*math.atan2(math.sqrt((q1*q1)+(q2*q2)),q5)+1.0)

    ######################################################
    #run everything nicely

    list_plot = []
    b_soln, b_fitness = main()
    print("###############################################################################")
    print("\nBest Solution")
    print(b_soln, "{f(s) = ", b_fitness, "}")

    for a in range(0, len(b_soln)):
        list_plot.append(dataset[b_soln[a]])

    list_plot.append(list_plot[0])

    plt.gca().invert_yaxis()
    plt.scatter(*zip(*dataset))
    plt.plot(*zip(*list_plot))
    plt.show()
    print("###############################################################################")
```

Tsp_with_improvement.py

```
import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt

from time import time
from random import shuffle, randrange, randint
#number of generation
n = 10

def main():

    #Data preparation
    df = pd.read_csv('test_100.csv', header = 0)
    df = df.drop('city', 1)
    df_drop = df.drop_duplicates(subset = 'i', keep = 'first')
    #df -> array
    dataset = df_drop.as_matrix(columns = None)

    start = time()
    #Swap algorithm
    path, length = swap(dataset)
    print(path)

    tottime = time() - start
    print("time = ",tottime)
    print("total_distance", length)

    list_plot = []
    for x in range(0, len(path)):
        list_plot.append(dataset[path[x]])

    list_plot.append(list_plot[0])
    plt.gca().invert_yaxis()
    plt.scatter(*zip(*dataset))
    plt.plot(*zip(*list_plot))

    plt.show()

################################################################################
#Swap algorithm
def swap(dataset):
    best_order = []
    best_length = float('inf')

#    order =  list(range(dataset.shape[0]))
#    shuffle(order)

    for i in list(range(n)):
        order =  list(range(dataset.shape[0]))
#Create random solution
        shuffle(order)
        length = calc_length(dataset, order)
        print("generation: ", 1+i, " [f(s): ", best_length,"]")

        changed = True
        while changed:

            changed = False

            for a in range(0, dataset.shape[0]):

```

```
63.
64.                 for b in range(a+1, dataset.shape[0]):
65.
66.                     new_order = order[:a] + order[a:b][::-1] + order[b:]
67.                     new_length = calc_length(dataset, new_order)
68.
69.                     if new_length < length:
70.                         length = new_length
71.                         order = new_order
72.                         changed = True
73.
74.
75.         if length < best_length:
76.             best_length = length
77.             best_order = order
78.
79.     return best_order, best_length
80. ###############################################################################
81. #Calculate length
82. def calc_length(dataset, path):
83.     length = 0
84.     for i in list(range(0, len(path))):
85.         length += distance(dataset[path[i-1]], dataset[path[i]])
86. #        length += tsp_cost(path[i-1], path[i], dataset)
87.     return length
88.
89. #Distance square
90. def distance(c1, c2):
91.     t1 = c2[0] - c1[0]
92.     t2 = c2[1] - c1[1]
93.
94.     return math.sqrt(t1**2 + t2**2)
95. ###############################################################################
96. def tsp_cost(i, j, dataset):
97.     pi = 3.1415926535879323846264
98.
99.     lat_i = (pi*dataset[i][0])/180
100.          lat_j = (pi*dataset[j][0])/180
101.          long_i = (pi*dataset[i][1])/180
102.          long_j = (pi*dataset[j][1])/180
103.
104.          q1 = math.cos(lat_j)*math.sin(long_i-long_j)
105.          q3 = math.sin((long_i-long_j)/2.0)
106.          q4 = math.cos((long_i-long_j)/2.0)
107.          q2 = (math.sin(lat_i+lat_j)*q3*q3) - (math.sin(lat_i-lat_j)*q4*q4)
108.          q5 = (math.cos(lat_i-lat_j)*q4*q4) - (math.cos(lat_i+lat_j)*q3*q3)
109.
110.          return int(6378388.0*math.atan2(math.sqrt((q1*q1)+(q2*q2)),q5)+1.0)
111.      ###############################################################################
112.      main()
```

## Part I.B: Artificial Neural Network

ANN.py

```
import numpy as np
import matplotlib.pyplot as plt

#Learning rate
#L_r = [0.001,0.01,0.1,1,10,100,1000]
L_r = [0.01]
#Number of Hidden layer's neuron
hiddenSize = 5

# Sigmoid Function
def sigmoid(x):
    output = 1/(1+np.exp(-x))
    return output

# Derivative of the sigmoid function
def sigmoid_output_to_derivative(output):
    return output*(1-output)

X = 40 * np.random.random_sample((10, 1)) - 20
#print(X.shape)

y = (X**3)
#print(y.shape)

for n in L_r:
    print("\nTraining With Learning rate: " + str(n))
    np.random.seed(1)

    # randomly initialize our weights in range of [-5,5]
    W1 = 10*np.random.random((1,hiddenSize)) - 5
    W2 = 10*np.random.random((hiddenSize,1)) - 5

    for j in range(1000):

        # Feed forward through layers 0, 1, and 2
        layer_0 = X
        layer_1 = sigmoid(np.dot(layer_0,W1))
        layer_2 = sigmoid(np.dot(layer_1,W2))

        layer_2_error = layer_2 - y

        if (j% 1) == 0:
            print("Error after "+str(j+1)+" iterations:" + str(np.mean(np.abs(layer_2_error))))

        layer_2_delta = layer_2_error*sigmoid_output_to_derivative(layer_2)

        layer_1_error = layer_2_delta.dot(W2.T)
        layer_1_delta = layer_1_error * sigmoid_output_to_derivative(layer_1)
        # Update the weight
        W2 += L_r * (layer_1.T.dot(layer_2_delta))
        W1 += L_r * (layer_0.T.dot(layer_1_delta))

print("\ninput -> X")
print(X)

print("\ny")
print(y)

print("\nafter training -> Predict")
print(layer_2)
```

```
61.
62. plot_1 = plt.scatter(X, y)
63. plot_2 = plt.scatter(X, layer_2, marker='x', color='r')
64.
65. plt.legend((plot_1, plot_2),
66.            ('dataset (X)', 'Predicted value'),
67.            scatterpoints=1,
68.            loc='lower right',
69.            ncol=3,
70.            fontsize=15)
71.
72.
73. plt.xlabel('X')
74. plt.ylabel('y')
75. plt.show()
```