Lab 6: Optimization using Simulated Annealing

1 Objectives

The main objective for this lab is to create a numerical optimization example using simulated annealing and then to develop your own custom version of the optimizer for more general purpose applications.

1. Learn how simulated annealing optimization works using an example
2. Create a simple simulated annealing function to optimize a number
3. Create a simulated annealing function to optimize a function

At the end of this laboratory session you should have the basic framework for simulated annealing optimization and you could explore how to potentially apply this to optimization of neural network parameters.

2 Getting Started

As we have seen in the lecture on optimization, the algorithm for simulated is as shown in figure 1. The key aspect of the algorithm is that we need to have specific functions to create a new (random) starting point, a *neighbour* version for comparison based on this starting individual, a *cost* function, an acceptance probability function, and finally the overall annealing loop.

The good news as far as this is concerned is that if we keep the functions generic, then all we need to do is to modify the cost function and creation of new values to optimize and the main loop function can remain unchanged.

In the lecture we saw a simple example where a 10x10 matrix of energy demand figures were used as the target for optimization of a 10x10 matrix of energy supply values, and this was based on the python code below:

Listing 1: sa.py

```
1 import numpy as np
2 import math
3 import random
4 import matplotlib.pyplot as plt
```

```
5
6  def acceptance_probability(old_cost, new_cost, T):
7      a = math.exp((old_cost-new_cost)/T)
8      return a
9
10 def anneal(solution,target,alpha,iterations,var):
11      old_cost = cost(solution, target)
12      cost_values = list()
13      cost_values.append(old_cost)
14     T = 1.0
15      T_min = 0.000001
16      while T > T_min:
17          i = 1
18          while i <= iterations:
19              print("Iteration : " + str(i) + "Cost : " + str(old_cost))
20              new_solution = neighbour(solution,var)
21              new_cost = cost(new_solution,target)
22              ap = acceptance_probability(old_cost, new_cost, T)
23              if ap > random.random():
24                  solution = new_solution
25                  old_cost = new_cost
26              i += 1
27              cost_values.append(old_cost)
28          T = T*alpha
29      return solution, old_cost, cost_values
30
31 def cost(supply,demand):
32     delta = np.subtract(demand,supply)
33     delta2 = np.square(delta)
34     ave = np.average(delta2)
35     dcost = math.sqrt(ave)
36     return dcost
37
38 def neighbour(solution,d):
39     delta = np.random.random((10,10))
40     scale = np.full((10,10),2*d)
41
42     offset = np.full((10,10),1.0-d)
43
44     var = np.multiply(delta,scale)
45     m = np.add(var,offset)
46
47     new_solution = np.multiply(solution,m)
48
49     return new_solution
50
51
52
53 supply = np.full((10,10),0.5)
54 demand = np.random.random((10,10))
```

```
55
56  print("Supply")
57  print(supply)
58
59  print("Demand")
60  print(demand)
61
62  rms = cost(supply,demand)
63  print("RMS = " + str(rms))
64
65  alpha=0.9
66  iterations = 200
67  var = 0.01
68
69  final_solution, cost, cost_values = anneal(supply,demand,alpha,iterations,var)
70
71  print(demand)
72  print(final_solution)
73  print(cost)
74
75  plt.subplot(232)
76  plt.title("Error Function in Simulated Annealing")
77  plt.plot(cost_values)
78  plt.grid(True)
79
80  plt.subplot(234)
81  plt.title("Initial Supply Matrix")
82  plt.imshow(supply, cmap='hot')
83
84  plt.subplot(235)
85  plt.title("Optimized Supply Matrix")
86  plt.imshow(final_solution, cmap='hot')
87
88  plt.subplot(236)
89  plt.title("Demand Matrix")
90  plt.imshow(demand, cmap='hot')
91
92  plt.show()
```

Now, in this example we have quite a bit of plotting of data, but we can examine the code section by section to understand what is going on.

2.1 Simulated Annealing Function

The heart of the simulated annealing algorithm is implemented in the anneal function. the current solution is passed to the function, along with the target value (this would be a constraint perhaps or minima/maxima for example), alpha (reduction in temperature per iteration), iterations (number of iterations at each temperature) and var (the variation of the random number used to

generate a new neighbour).

The temperature is defined inside this routine and you could legitimately move this outside and make it a parameter.

It would also be good practice to consider making a simulated annealing class and have all the relevant functions defined in the class.

Listing 2: anneal function

```
def anneal(solution,target,alpha,iterations,var):
    old_cost = cost(solution, target)
    cost_values = list()
    cost_values.append(old_cost)
    T = 1.0
    T_min = 0.000001
    #alpha = 0.9
    #iterations = 100
    # var = 0.01
    while T > T_min:
        i = 1
        while i <= iterations:
            print("Iteration : " + str(i) + "Cost : " + str(old_cost))
            new_solution = neighbour(solution,var)
            new_cost = cost(new_solution,target)
            ap = acceptance_probability(old_cost, new_cost, T)
            if ap > random.random():
                solution = new_solution
                old_cost = new_cost
            i += 1
            cost_values.append(old_cost)
        T = T*alpha
    return solution, old_cost, cost_values
```

The cost function is defined as the difference in the value using the current solution and the desired target. In this example it it simply the RMS difference of all the values, and we are looking to minimize the cost as a result. In the annealing function we create a list called cost_values and the rationale for this is so we have a complete record of all the cost values which can be plotted at the conclusion of the optimization.

2.2 Acceptance Probability function

The acceptance probability function is the implementation of the exponential calculation of the probability that we will accept the new value, and this is useful as it can be applied to any problem, as long as we can reduce the cost to a single value. We could therefore use this in ANY simulated annealing implementation.

Listing 3: acceptance probability

```
def acceptance_probability(old_cost, new_cost, T):
    a = math.exp((old_cost-new_cost)/T)
    return a
```

2.3 Neighbour function

The neighbour function in this case is a new 10x10 matrix of values that is a random variation of the current solution. This function is specific to the current example and will need a new routine to be made to suit the problem under investigation.

Listing 4: new neighbour

```
def neighbour(solution,d):
    delta = np.random.random((10,10))
    scale = np.full((10,10),2*d)

    offset = np.full((10,10),1.0-d)

    var = np.multiply(delta,scale)
    m = np.add(var,offset)

    new_solution = np.multiply(solution,m)

    return new_solution
```

2.4 Cost function

The cost function in this example is a calculation of the RMS error between the energy supply and demand matrices. This has specific names for the supply and demand (but easy to change to target and current or similar to make it general) and it is based on np matrices, therefore these would need to be the format in which the data sets were defined (but obviously this could be done for most situations).

The cost function used will depend on the application and how you wished to calculate it (for example, Euclidean distance might be used)

Listing 5: cost function

```
def cost(supply,demand):
    delta = np.subtract(demand,supply)
    delta2 = np.square(delta)
    ave = np.average(delta2)
    dcost = math.sqrt(ave)
    return dcost
```

3 Lab Exercise

1. Modify the code (or create your own from scratch) to create a simple optimization where the function of the optimizer is to simply find a number specified in the shortest number of iterations

2. How do the number of iterations, variability and alpha parameters affect how quickly a solution is found?

3. Modify the code (or create your own from scratch) to create an optimization of a linear array of 10 numbers and find the values as quickly as possible

4. Create an optimization to optimize a straight line that gives the best fit of a random array of 100 points in XY space between 0 and 100 in both axes.

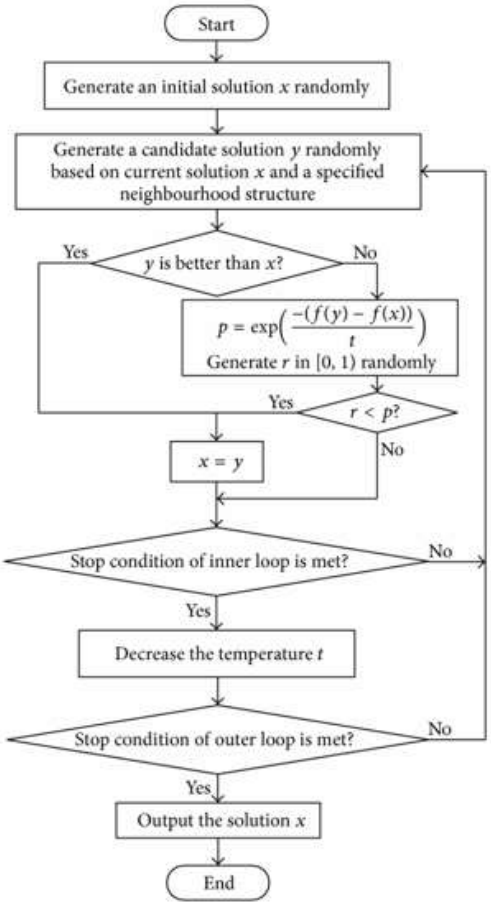

Figure 1: Algorithm for Simulated Annealing