Final Project

Proposed Theme: Simulated Annealing and Its Applications

Artificial Intelligence

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais

Author: *Rodrigo P. Coelho*
Date: *January 10th, 2017*

CONTENT

INTRODUCTION

Simulated Annealing (SA) is an optimization method which has been widely used since its introduction (1983 Kirkpatric, Gelatt, Vecchi) to solve many important problems in Engineering, Science and logistics. Examples include the Traveling Salesman Problem [1], layout and wiring in circuit boards [2], container optimization (or the classical Knapsack problem [3]), optimization of finance portfolios and protein chains, job shop scheduling and others [4]. It is particularly useful in problems where the objective function is non-linear with many local minima/maxima. It can be used to find the global minimum of continuous functions as well as combinatorial optimization of very large and complex systems.

Kirkpatric (1983) [2], who introduced the method as it is known today, further developed the Metropolis algorithm (1953), which was used to simulate the behavior of Annealing in physics, but had the insight to apply this methodology to solve other types of problems. He observed that materials, which contain 10^{23} atoms per cubic centimeter, tend to naturally find a low energy state when cooled slowly. He then developed the process by drawing an analogy of how these phenomena happened in nature and applying the same mechanical statistics principles to minimize combinatorial and other objective functions.

It can be observed that, the same characteristics that make the method very flexible, sometimes can make it difficult to analyze certain types of problems, especially in multidimensional state spaces where the characteristics of the utility function usually are not well known. In these situations, determining the starting temperature T_o and the cooling schedule, which governs the decay of the initial temperature, might have to be discovered empirically. Another parameter less analyzed in the literature is how to determine the exploration of the search space.

In this paper we will explore the components and parameters used in Simulated Annealing and discuss some features that were programed into a Python class (SA object). The full project entailed implementing the method to optimize functions of the type z = f(x,y), train 2 different neural networks, and solve the combinatorial TSP routing problem using an expandable data file as input (our sample data file contains the coordinates for 123 cities). However, in this paper we will limit ourselves in describing the developed test bench system and applying it to train a neural network to recognize hand written digits which was the most complex and challenging part of the project.

This study starts by analyzing and testing the different parameters available to us in a Simulated Annealing implementation and how they affect the method's capability of finding a global minimum and the program's execution time. Different types of objective functions, probability for acceptance functions, cooling schedules, and methods for exploring the search space were studied on the developed three dimensional test bench optimizer. The model was then expanded to work with n-dimensional problems, and applied to solve the practical problems mentioned in the previous paragraph. Finally, some of the difficulties encountered in working with the method will be described.

UNDERSTANDING COMPONENTS AND FUNCTION PARAMETERS

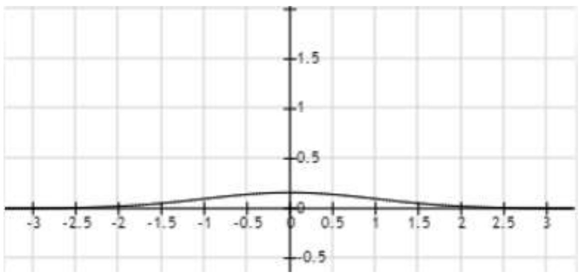
Simulated Annealing Components:

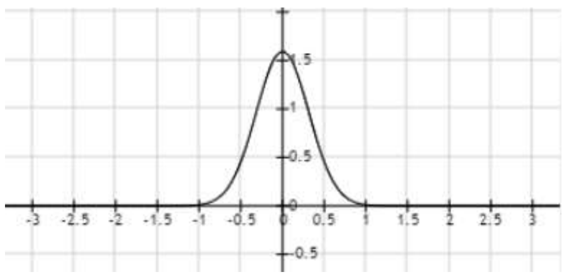
Ingber [5] breaks down the Simulated Annealing algorithm in three main components. Any attempt on improving the algorithm has to come from refining the performance of each of the components or how they work together. The three main components are:

1. *g(x)*: Probability density of state-space of D parameters $x = \{x^i ; i = 1, D\}$, at temperature T, or some criteria to choose the next point to explore in the search space. Ideally *g(x)* belongs to the Gaussian-Markovian systems and the original algorithm chooses for *g*:

$$g(x) = (2\pi T)^{-\frac{D}{2}} \exp[\frac{-(x - x_0)^2}{2T}]$$

This just means that it has a Gaussian distribution and for higher temperatures you can guess points further away as shown in the graphs below:


x_o=0, D=2,T=1


x_o=0, D=2,T=0.1

The temperature T, behaves as the standard deviation of this distribution. We can see that as the temperature decreases, there is a greater probability of guessing the next point closer to the current state.

The use of a Gaussian distribution is important as it greatly increases the rate at which global minima are found. In the developed test bench system this was implemented using the Python random.gauss() function:

```
xn[nd] = (random.gauss(xbest[nd],math.sqrt(T)))
```

In a 1000 simulations starting at [0,0] the global minimum of our test function was found 14 times when using a uniform distribution and 80 times using the Gaussian sampling shown above. The Gaussian approach took only 10 seconds more to execute all the iterations.

This rate can be improved even further when you introduce the concept of "Narrowing", which we will discuss later, to either distribution. In our tests, the best combination came out to be to use narrowing with the Gaussian distribution which had 114 successes out of 1000.

Many articles (for example [20]) just add a linear Δx for the guess of the next state, and certain behaviors can only be induced with this uniform distribution so it was also implemented as follows:

```
xn[nd] = xbest[nd] + random.uniform(-self.DELTAObjFunc * self.OBJfunctionRANGE,
            self.DELTAObjFunc * self.OBJfunctionRANGE)
```

However, as a general norm in this paper the Gaussian distribution was used because it showed a better rate finding the global minimum and because it is the way Simulated Annealing was originally proposed.

2. *h(ΔE)*: Probability for acceptance of new cost-function given the previous value. Here Ingber deduces *h(x)* as:

$$h(x) = \frac{\exp(-E_{k+1}/T)}{\exp(-E_{k+1}/T) + \exp(-E_k/T)} = \frac{1}{1 + \exp(\Delta E/T)}$$

This is also known as the Barker criterion [6]. This function has a "go", "no go" behavior that becomes stricter as the temperature decreases. This means that the lower the temperature the function will only accept an E_{k+1} that is lower than E_k.

3. *T(k)*: The "Cooling" schedule *T* in *k* annealing-time steps. As we showed above, changes in *T* change the volatility of the previous probability densities affecting the behavior of the acceptance function and the sampling function. The cooling schedule, working in conjunction with the

acceptance function, regulates how fast the SA algorithm will transition from an exploratory phase to a “hill climbing” phase.

Choosing a Test Objective Function:

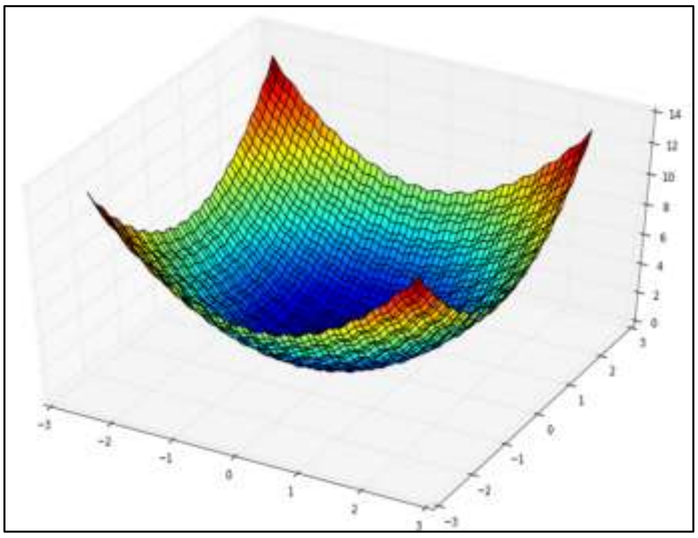
Now that the principal components of the method have been understood, the next step was to choose objective functions to analyze. Several functions were used with progressively increasing challenges. To make switching between functions easy, a dictionary was developed and a function could be chosen to be studied by changing the *FuncOption* in the SA object class (Simulated Annealing Python object developed):

```
def f(self, xVect):     # define objective functions
    x = xVect[self.DimX]
    y = xVect[self.DimY]
    Funct = {'A': 0.7 + x**2 + y**2 - 0.1*math.cos(6.0*3.1415*x) - 0.1*math.cos(6.0*3.1415*y),
             'B': (math.e**math.sin(50*x) + math.sin(60*math.e**y) + math.sin(70*math.sin(x)) +
                  math.sin(math.sin(80*y)) - math.sin(10*(x + y)) + 0.25*(x**2 + y**2)),
             'C': x**2+ y**2 + 5,
             'D': 10000*(math.e**math.sin(50*x) + math.sin(60*math.e**y) + math.sin(70*math.sin(x)) +
                  math.sin(math.sin(80*y)) - math.sin(10*(x + y)) + 0.25*(x**2 + y**2)),
             'E': (2*x**6-12.2*x**5+21.2*x**4+6.2*x-6.4*x**3-4.7*x**2+y**6-11*y**5+43.3*y**4-10*y-
                   74.8*y**3+56.9*y**2-4.1*x*y-0.1*y**2*x**2+0.4*x*y**2+0.4*x**2*y),
             'F': (1-x)**2 + 25*(y-x**2)**2 + (math.cos(3*math.pi*x+4*math.pi*y)+1),
             'G': (20*math.sin(x*np.pi/2-2*np.pi) + 20*math.sin(y*np.pi/2-2*np.pi)+(x-2*np.pi)**2+(y -
                   2*np.pi)**2)}
    return Funct[self.FuncOption]
```

The main functions studied were:

1A) `0.7 + x**2 + y**2 - 0.1*cos(6.0*3.1415*x) - 0.1*cos(6.0*3.1415*y)`

This was an initial function just to test the system. It has an obvious minimum at (0.0 , 0.0) with value 0.5.

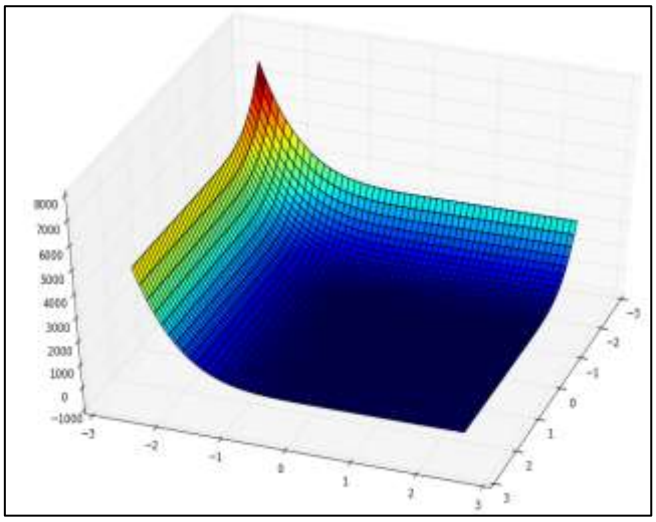


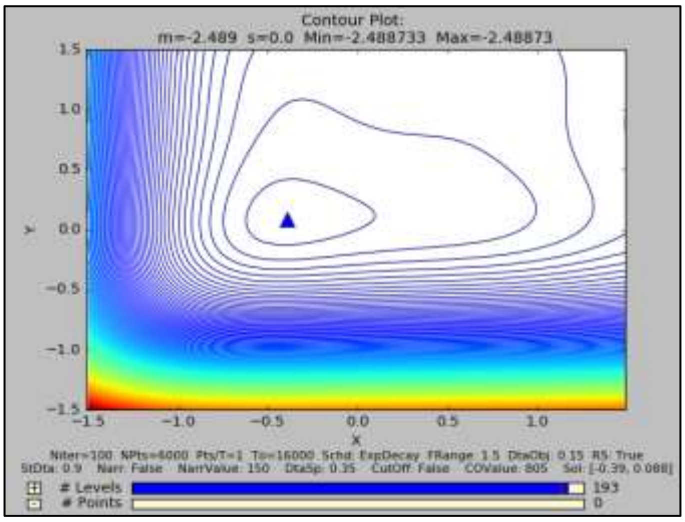
2E) `2*x**6-12.2*x**5+21.2*x**4+6.2*x-6.4*x**3-4.7*x**2+y**6-11*y**5+43.3*y**4-10*y-74.8*y**3+56.9*y**2-4.1*x*y-0.1*y**2*x**2+0.4*x*y**2+0.4*x**2*y`

This function (see figure below) was used to test how the system would perform finding the minimum of functions with plateaus. It has a minimum at (-0.39021193 , 0.08771491) with value -2.4887334209.

On the right, the contour graph shows the minimum found. The slider “# of Points” can be used to step through each of the guesses. This was developed in order to understand how the setting of each parameter affected the behavior of the search. The slider “# of Levels” controls how many levels to plot on the

Contour Graph. It is possible to control if this graph will apear or not at the end of the simulation by setting the *GRAPHcontour* property.



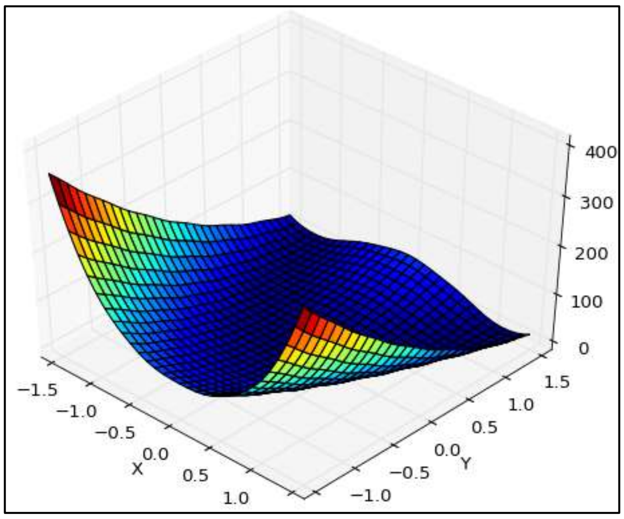


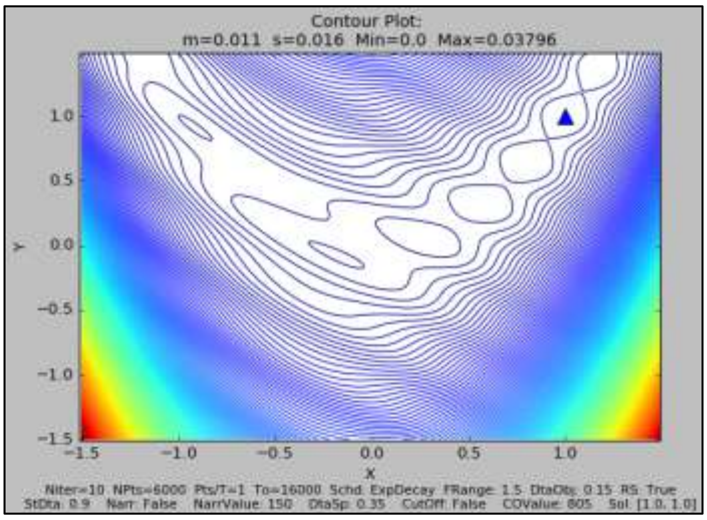
```
Niter=100 NPts=6000 Pts/T=1 To=16000 Schd: ExpDecay FRange: 1.5 DtaObj: 0.15 RS: True
StDta: 0.9 Narr: False NarrValue: 150 DtaSp: 0.35 CutOff: False COValue: 805 Sol: [-0.39, 0.088]
```

3F) `(1-x)**2 + 25*(y-x**2)**2 + (cos(3*pi*x+4*pi*y)+1)`

This function has many local minima at different distances from each other. It was important in understanding how the setting of the Delta Space Value affected the minimum that was found when using State Space Narrowing (defined in the “Implemented Features” section). When using uniform sampling it also helped to comprehend how the step size (Δ*x*, Δ*y*), set by the *DELTAObjFunc* property, impacted the number of times the minimum was found.

When analyzing the contour graph on the right, it is intuitive to understand that the ideal value for the Delta Space Setting will vary by function, depending on how far these local minima are from each other. The Global Minimum is located at (1.0 , 1.0) with value equal to 0.

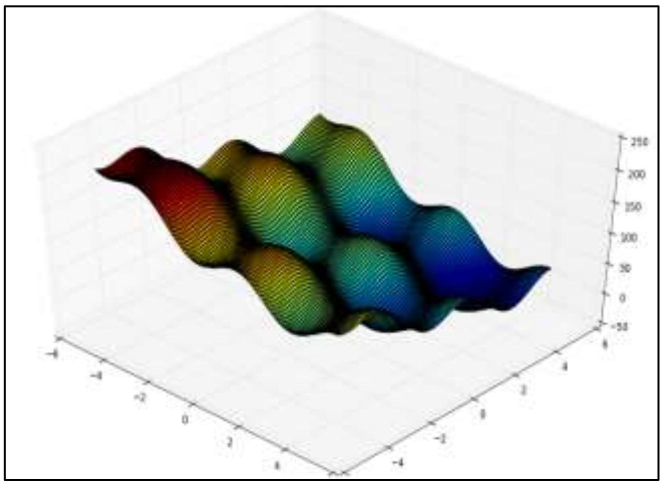


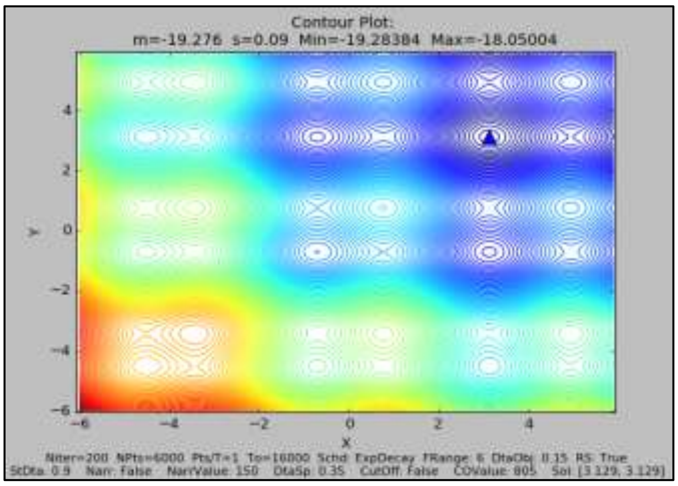


```
Niter=10 NPts=6000 Pts/T=1 To=16000 Schd: ExpDecay FRange: 1.5 DtaObj: 0.15 RS: True
StDta: 0.9 Narr: False NarrValue: 150 DtaSp: 0.35 CutOff: False COValue: 805 Sol: [1.0, 1.0]
```

4G) `20*sin(x*pi/2-2*pi) + 20*sin(y*pi/2-2*pi)+(x-2*pi)**2+(y - 2*pi)**2)`

The function on the next page is continuously decreasing. It was used to test if the objective function range parameter was working, since its global minimum will depend on the range chosen to study.

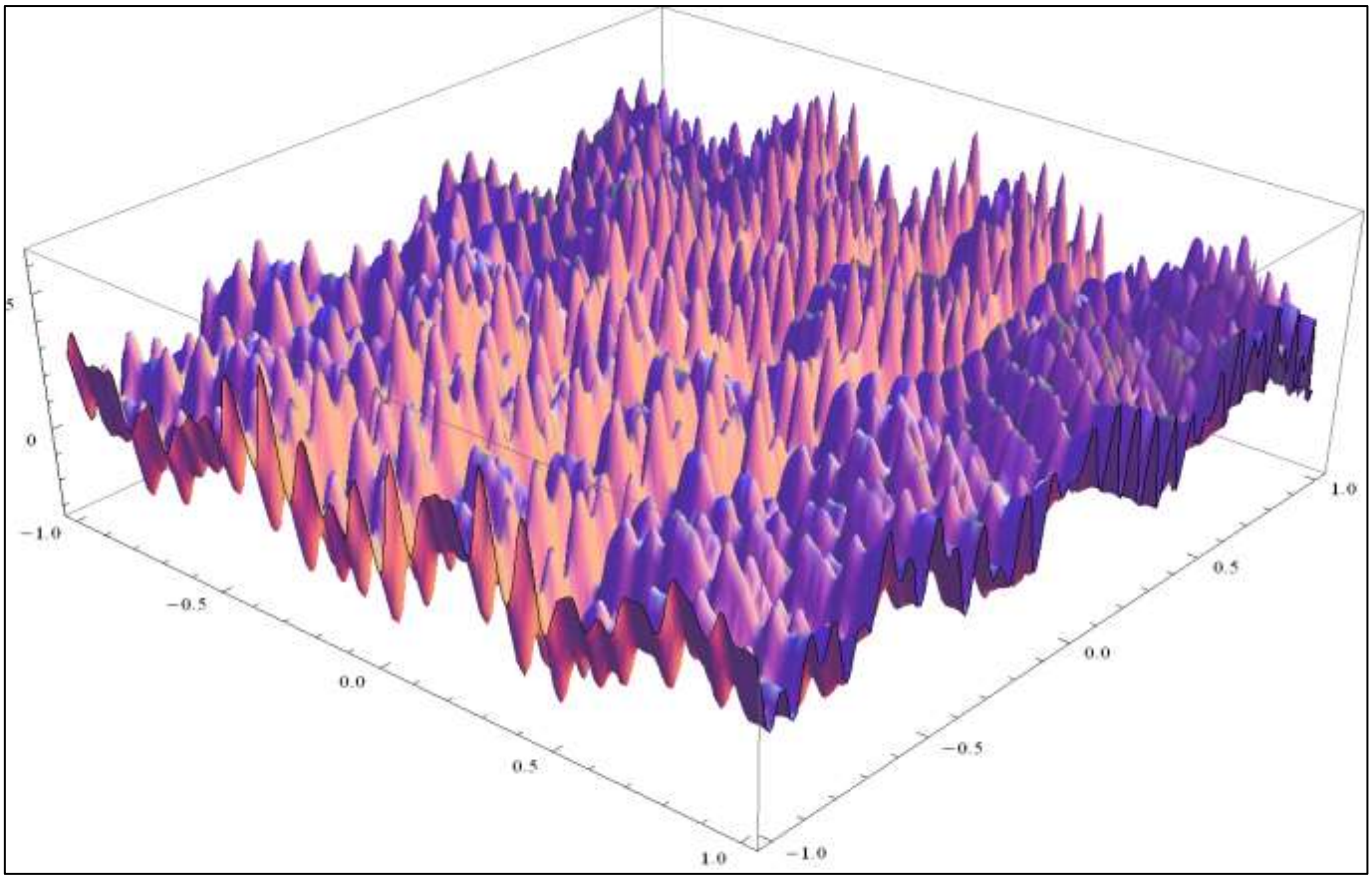




Niter=200 NPts=6000 Pts/T=1 To=16000 Schd: ExpDecay FRange: 6 DtaObj: 0.15 RS: True
StDta: 0.9 Narr: False NarrValue: 150 DtaSp: 0.35 CutOff: False COValue: 805 Sol: [3.129, 3.129]

```
5B)e**sin(50*x) + sin(60*e**y) + sin(70*sin(x)) + sin(sin(80*y)) –
sin(10*(x + y)) + 0.25*(x**2 + y**2)
```

Function 5B was chosen for most of the studies. It is used in a Mathematica to demonstrate different optimization methods available in the system. It made an excellent test bench function due to the immense number of local minima. The function has a parabolic term (0.25*(x^2+y^2)) which makes it have one global minimum located at (-0.02440308, 0.21061243) with value -3.30686864748.

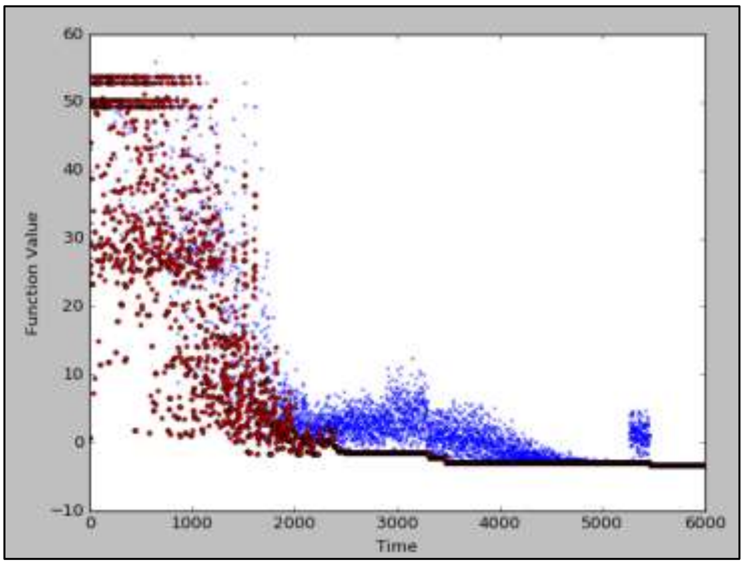


Auxiliary Graphs Implemented:

Several articles mention that one of hardest difficulties of working with the SA method is to identify the right parameters to use. Among them, usually identifying the right starting temperature and cooling schedule to use are the most commonly cited, especially if you do not know the shape of the objective function. To aid in the understanding of how these and other parameters affect the performance of the algorithm, the following graphs were developed:

Graph Objecive Value x Time:

Shows the function values guessed (blue) and values accepted (red) versus time (in the *x* axis):



*Gaussian Sampling, StateSpaceNarrowing=400, FRange=10

This graph is useful to evaluate if the function is converging with the given cooling schedule and to see if it is possible to reduce the number of time steps evaluated. For exemple, after t=5500 there isn't much improvement in the function value. The number of time steps can then be reduced to 5500 using the *NUMpoints* property saving some computational time. This graph also helps to evaluate if enough time is being spent exploring versus "hill climbing", therefore aiding in the setting of T_0.

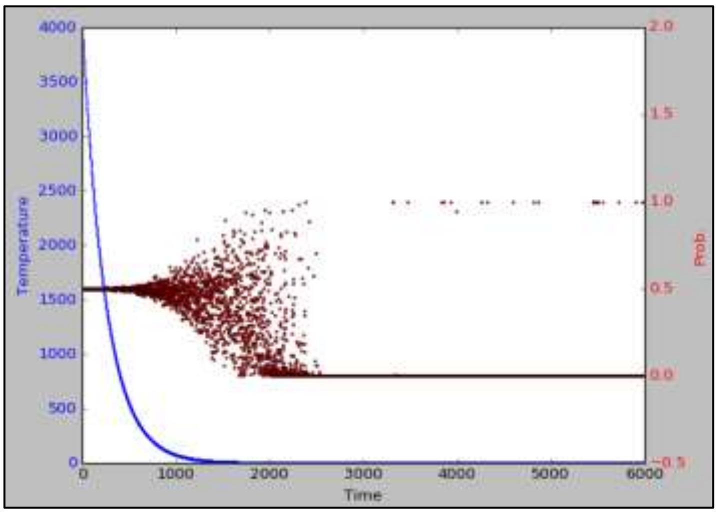
The plateaus shown in the left side of the graph appear when a guess is above the value set for the objective function range. The routine implemented clips the value to the maximum allowed by the parameter (see lines below). Lowering the starting temperature or increasing the objective function range will make this plateau disappear but at a cost of decreasing the number of times the global minimum will be found.

```
if xn[nd] < -self.OBJfunctionRANGE: xn[nd] = -self.OBJfunctionRANGE
if xn[nd] > self.OBJfunctionRANGE: xn[nd] = self.OBJfunctionRANGE
```

The little spike of blue dots around t=5500 is due to state space narrowing being on. As it can be seen this helped reduce the value of the objective function. How state space narrowing works will be explained in the "Implemented Features" section.

Graph Probability x Time:

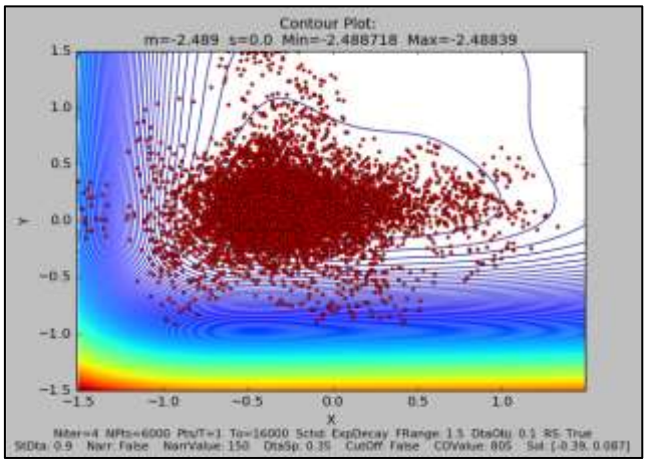
This graph shows a plot of the calculated probabilities and the temperature schedule on the same window. This graph also helps in the setting of the starting temperature by showing the user exactly when the algorithm starts to transition from the exploration phase into the "hill climbing" phase. As we can observe on either graph, this happened around t=2200.



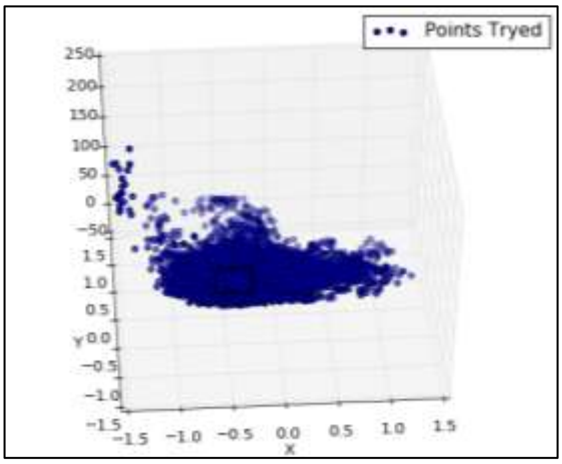
*Barker Criterion

Contour and 3D Plot Graphs:

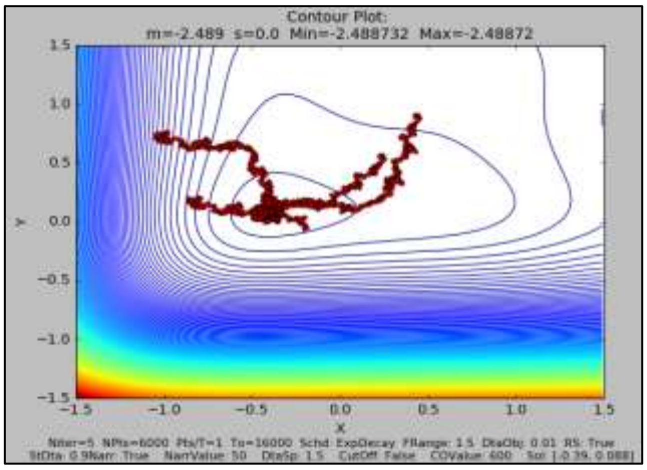
The contour and 3D plot graphs are helpful in observing the search behavior, which can be completely altered depending on the setting of the parameters.



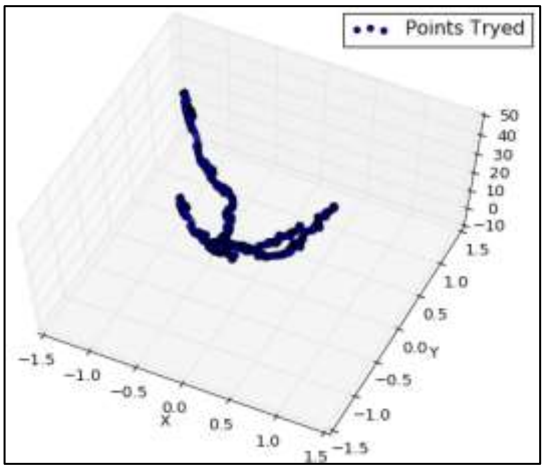
Niter=4 NPts=6000 Pts/T=1 To=16000 Schd: ExpDecay FRange: 1.5 DtaObj: 0.1 RS: True
StDta: 0.9 Narr: False NarrValue: 150 DtaSp: 0.35 CutOff: False COValue: 805 Sol: [-0.39, 0.087]



For example, by making the step size (*DELTAObjFunc*) small enough in a uniform *g(x)*, you can make the SA behave like a stochastic gradient descent:



Niter=5 NPts=6000 Pts/T=1 To=16000 Schd: ExpDecay FRange: 1.5 DtaObj: 0.01 RS: True
StDta: 0.9Narr: True NarrValue: 50 DtaSp: 1.5 CutOff: False COValue: 600 Sol: [-0.39, 0.088]

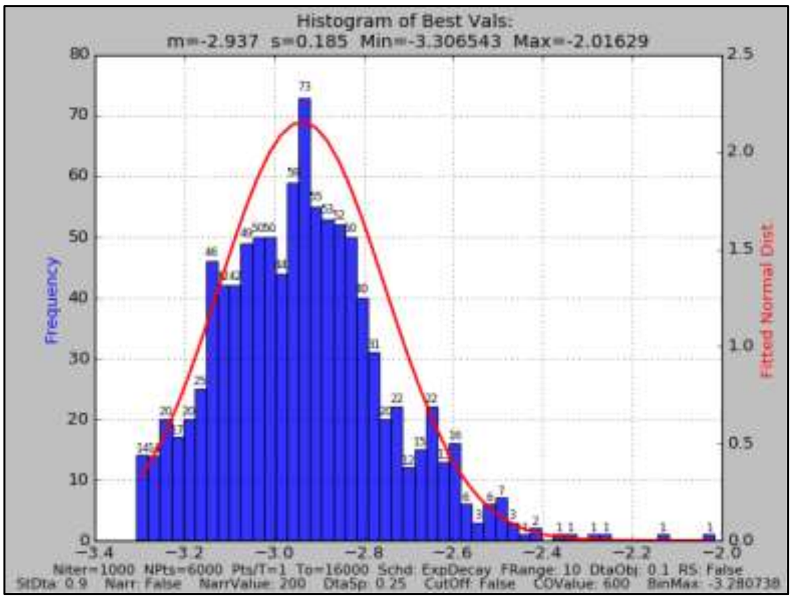


Histogram:

The histogram was fundamental in comparing two different configurations. Because SA is a stochastic method, two runs will never be the same. The histogram allows you to check how many of several runs were very close or found the true global minimum.

The histogram developed shows the summary statistics for that run including the best minimum, the maximum, the average and the standard deviation. The best fit normal distribution is also shown on the graph.

The lower part of the histogram records the important simulation parameters specific for that run. They include the number of iterations, the number of points per iteration, the number of points per temperature, the cooling schedule used, the objective function range, the maximum step size (*DELTAObjFunc* – only applicable when g(x) uses uniform sampling), if recursive input was used, the random noise applied to each recursive input, if State Space Narrowing was used, the state space narrowing value, the delta space narrowing value, if Cutoff was used, the Cutoff Value and the upper limit of the lowest bin.



Niter=1000 NPts=6000 Pts/T=1 To=16000 Schd: ExpDecay FRange: 10 DtaObj: 0.1 RS: False
StDta: 0.9 Narr: False NarrValue: 200 DtaSp: 0.25 CutOff: False COValue: 600 BinMax: -3.280738

In most cases, a thousand runs were used to compare each configuration. The quality of a configuration was evaluated by counting the number of occurrences in the lowest bin and dividing by the total number of runs. When the multiple mini-batch process could not be used, the execution time was used as the criteria to compare different simulations.

After developing these graphs, understanding and setting the SA parameters became a trivial task.

Choosing the Acceptance Function *h(∆E)*:

The method used to choose between the two acceptance function criteria was to first analyze the histogram and check if there were significant performance differences. Next, the 3D plot, the probability plot and the function evaluation graphs were examined to see if a good sampling of the search space was achieved.

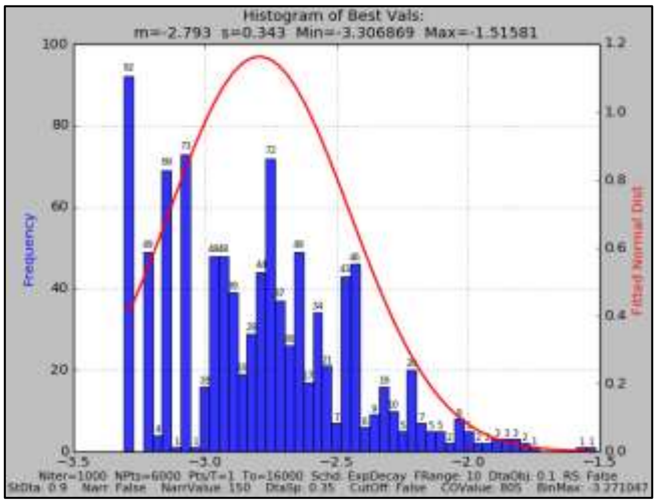
Tests were run for both Gaussian Sampling and Uniform Sampling with the same parameters but we show the graphs below only for Gaussian Sampling since it had better performance.

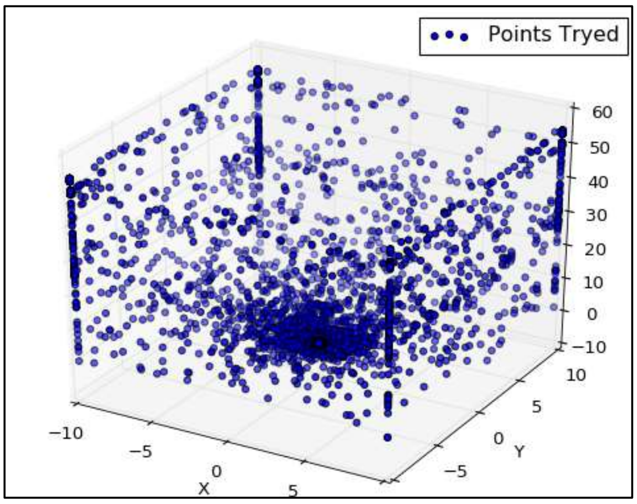
Barker Criterion:

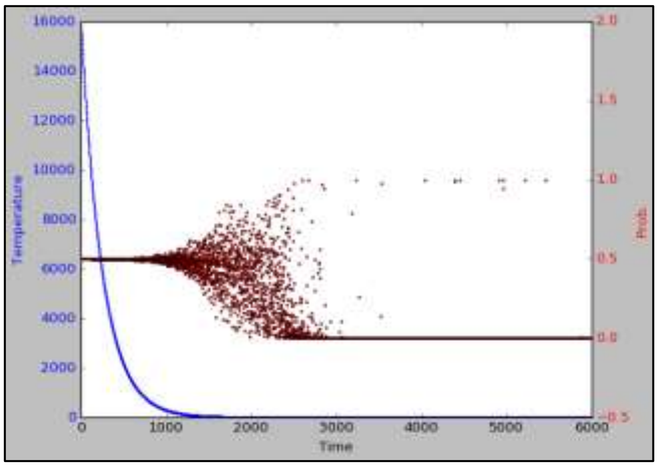
The Barker criterion acceptance function is described by the following equation:

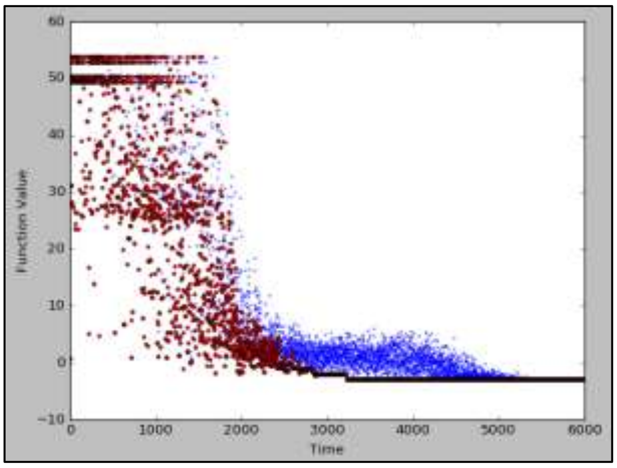
```
Prob = 1/(1+math.exp((fn - fbest) / T))
```

Histogram, 3D Plot and Probability and Function Values Evolution for the Barker Criterion









*These graphs show the behavior for 1 iteration.

*All graphs produced wih Gaussian Sampling.

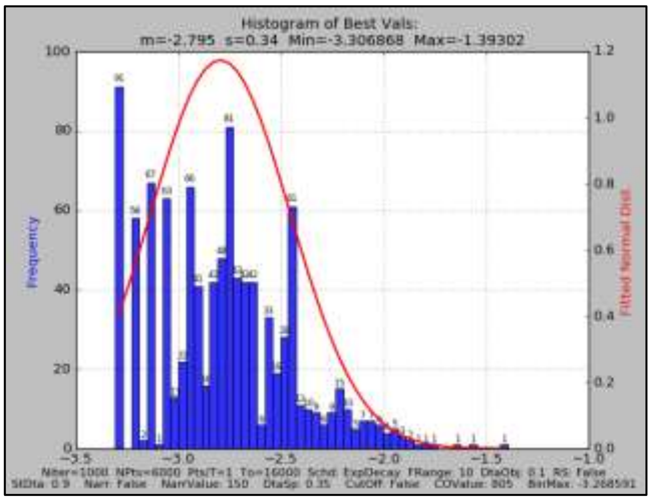
The interesting aspect of the Barker criterion is that it normalizes the probability, giving as a result always a number between 0 and 1.

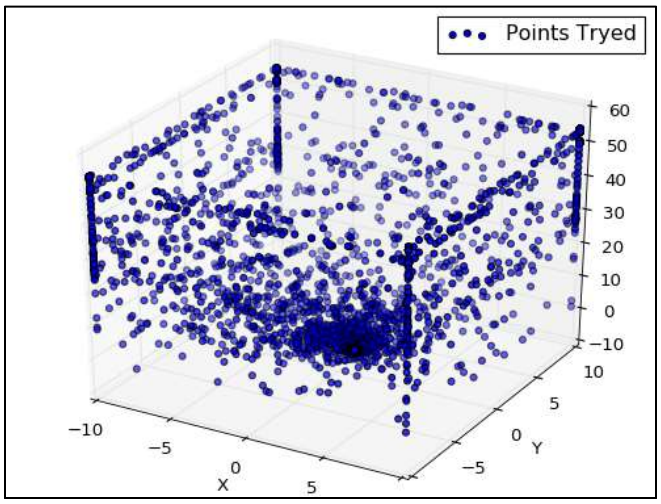
Metropolis Criterion:

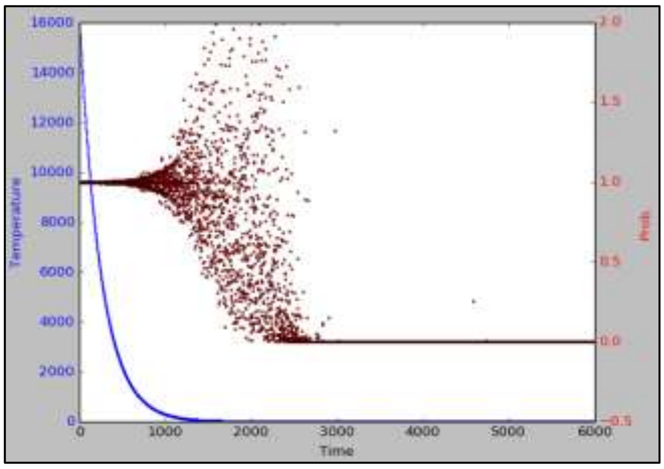
The Metropolis criterion is described by the following equation:

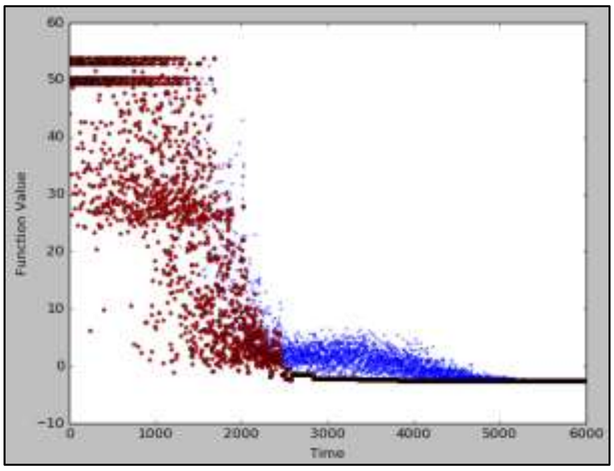
```
Prob = math.exp(-(fn-fbest) / T)
```

Histogram, 3D Plot and Probability and Function Values Evolution for the Metropolis Criterion









*These graphs show the behavior for 1 iteration.

*All graphs produced wih Gaussian Sampling.

Acceptance Criteria Results and Conclusion:

As can be seen by the graphs above both criteria do a good job sampling the search space and show very similar performances. The results were close for uniform sampling as well. They are summarized in the table below:

Sampling Method	Acceptance Criteria	Performance	%
Gaussian	Barker	92/1000	9.2%
Gaussian	Metropolis	91/1000	9.1%
Uniform	Barker	14/1000	1.4%
Uniform	Metropolis	15/1000	1.5%

For each sampling method there were no significant computational time differences. The Barker criterion was chosen because it normalizes the probability making the probability graph better behaved.

Defining the Search Space Size:

Having defined the sample distribution and the acceptance function for our test bench, the next step was to decide the size of the search space to be studied. A thousand simulations were run for the ranges (-1,1), (-10,10), (-1000,1000). To make changing this parameter easy for different functions and runs, a property called *OBJfunctionRANGE* was defined in the SA class where you could specify this search range.

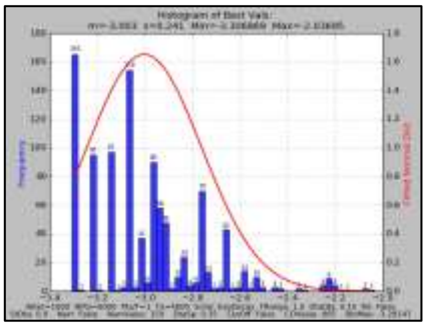
These simulations were run with Gaussian sampling, starting at (0.0, 0.0), (9.0,9.0), (90.0,90.0) respectively. They were run at T_0 = 4000, which worked well for all three ranges.

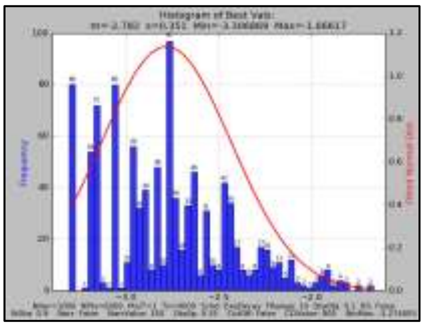
The table below shows that the method finds the global minimum no matter the size of the search space if a thousand iterations are used. This is due to the fact that the method does a good job exploring it at high temperatures as long as T_0 and the cooling schedule are set appropriately. The table below compares the results of the 3 simulations:

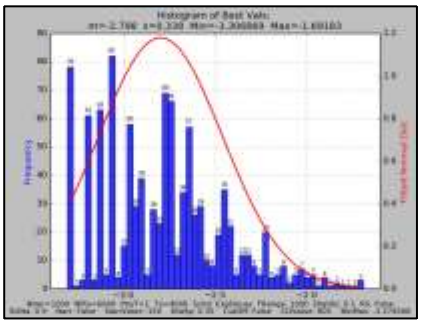
Func	Gauss	Niter	Npts	Pts/T	T_0	Sched	FRange	Min	Nmin	Time (s)	Mu	Sigma	Start
B	TRUE	1000	6000	1	4000	ExpDecay	**1**	-3.306868640	165	374.79	-3.003	0.241	(0.0,0.0)
B	TRUE	1000	6000	1	4000	ExpDecay	**10**	-3.306868582	80	400.11	-2.782	0.351	(9.0,9.0)
B	TRUE	1000	6000	1	4000	ExpDecay	**1000**	-3.306868615	78	399.65	-2.798	0.338	(90.0,90.0)

Nmin shows the number of times a solution fell into the lowest bin in the histogram which has 50 bins between the maximum and the minimum function values found. The other values in this table are self-explanatory.

Histogram Plots for the 3 State Space Sizes Studied

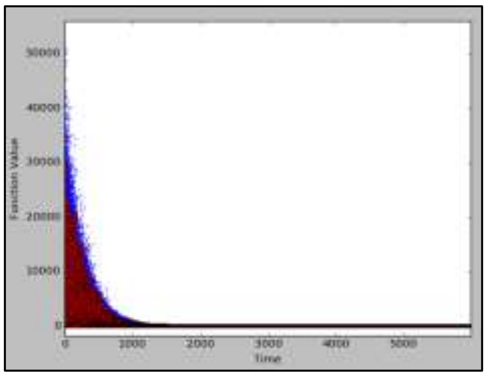

*Function Range = 1


*Function Range = 10

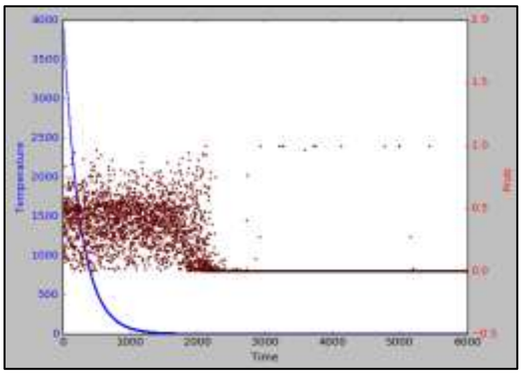

*Function Range = 1000

Range = 1000

Function Value Evolution Graph


*1000 iterations shown

Probability Value Evolution Graph

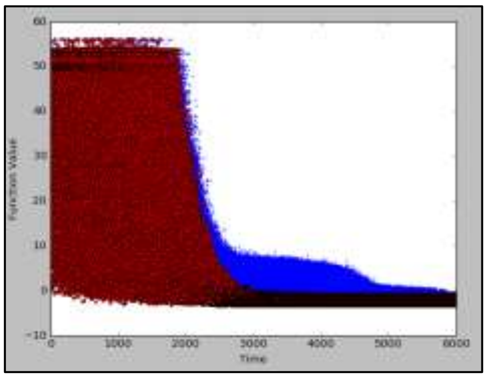

*1 iteration showing the "cloud of bad points" at high temperatures

It can be observed in the probability graph on right that there is a "cloud" of probabilities below 50% which means that more bad points are not accepted for the same temperature range in the larger state space.

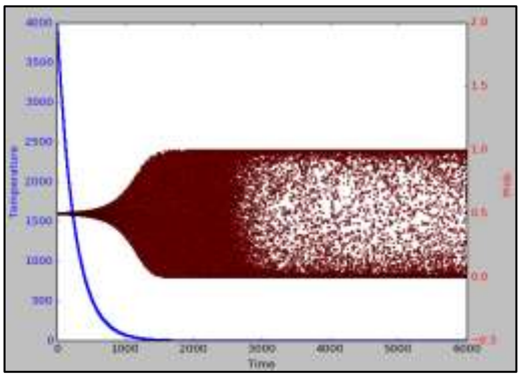
Range = 10

Function Value Evolution Graph

Probability Value Evolution Graph



*1000 iterations shown



*1000 iterations shown

The state space size of 10 was chosen for the studies because it presented a problem challenging enough for the method with a success rate of 8.0%. This gave us enough room to see if attempted changes to the algorithm would improve or decrease the efficiency.

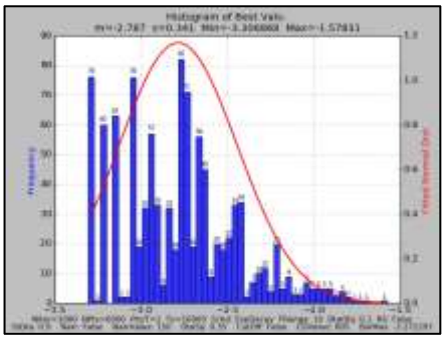
Understanding the Simulated Annealing Parameters:

In this section we will analyze how the starting temperature, the number of points sampled per temperature and the cooling schedule affect the success rate of the Simulated Annealing algorithm.

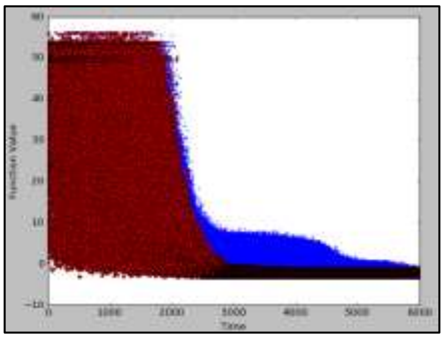
Starting Temperature T_0:

It can be seen from the graphs below, that the lower the starting temperature the less time the algorithm spends exploring the search space. Only when $T_0 = 1$, the exploration phase is clipped enough to start impacting the effectiveness of the SA.

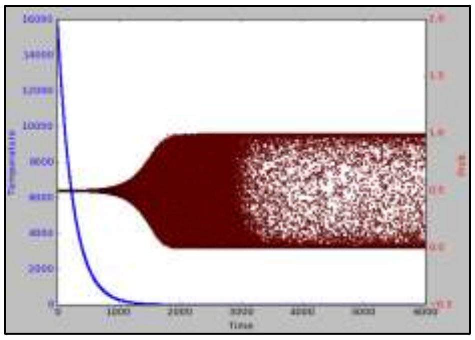
Graph Comparisons Between T_0 = 16,000 and T_0 =1



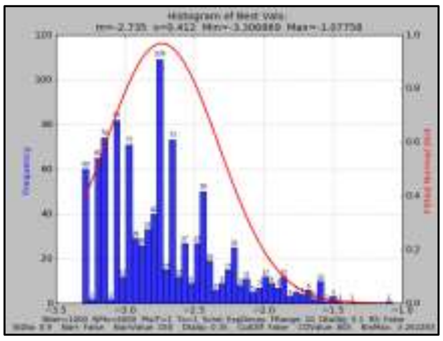
*T0 = 16,000



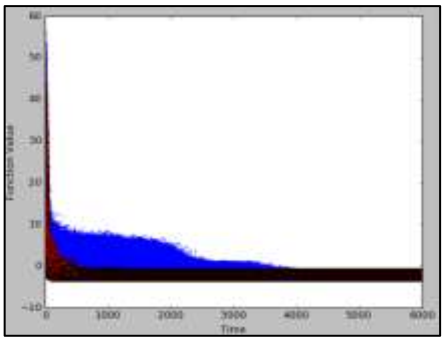
*T0 = 16,000



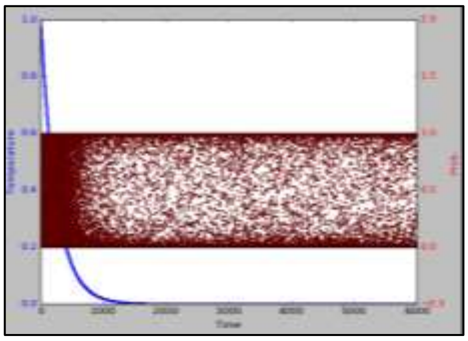
*T0 = 16,000



*T0 = 1



*T0 = 1



*T0 = 1

Func	Gauss	Niter	Npts	Pts/T	To	Sched	FRange	Min	Nmin	Time (s)	Mu	Sigma	Start
B	TRUE	1000	6000	1	1	ExpDecay	**10**	-3.306868647	60	409.34	-2.735	0.412	(9.0,9.0)

B	TRUE	1000	6000	1	25	ExpDecay	**10**	-3.306868646	75	398.79	-2,794	0.346	(9.0,9.0)
B	TRUE	1000	6000	1	50	ExpDecay	**10**	-3.306868645	87	404.47	-2.806	0.349	(9.0,9.0)
B	TRUE	1000	6000	1	500	ExpDecay	**10**	-3.306868624	77	393.83	-2.788	0.344	(9.0,9.0)
B	TRUE	1000	6000	1	16000	ExpDecay	**10**	-3.306868293	76	396.95	-2.787	0.341	(9.0,9.0)
B	TRUE	1000	6000	1	50	ExpDecay	**10**	-3.306868646	80	404.17	-2.820	0.334	(-7.0,9.0)

Note that the standard deviation changed very little until the T_0 was reduced to 1. This means that as long as T_0 is set high enough (above 25 in this case), for this function, you will have the same variation on your results with averages that are also pretty similar in value. Because T_0=50 had the best performance, the lowest average, and found the global minimum with a difference only in the 9th decimal place, it was chosen as the initial temperature for the tests relating to function '5B'.

Points per Temperature (L):

As can be observed in the table and graphs below, increasing the number of points per temperature greatly increases your chances of finding the global minimum. However, this comes with a cost in computational time since you are calculating the objective function L times for each temperature. This problem is worsened the more complex the objective function is to evaluate.

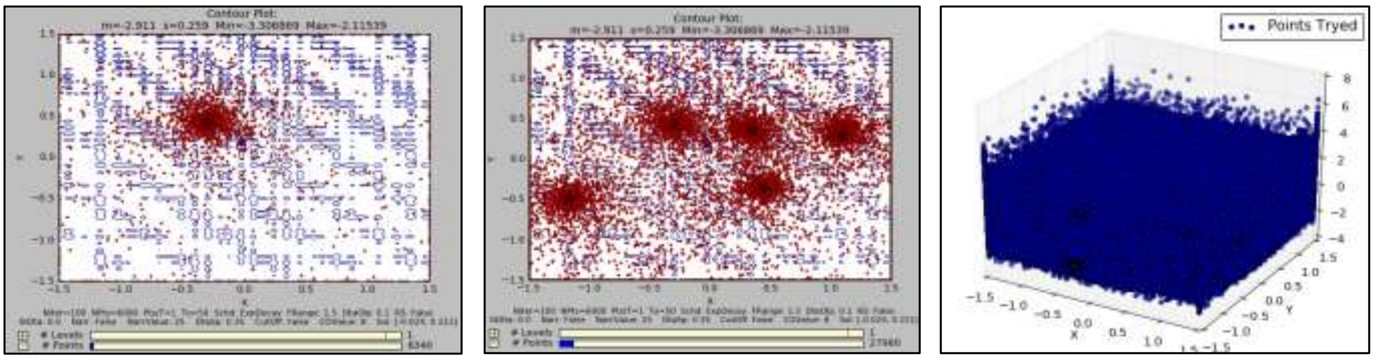
By increasing the number of assessments per temperature to 5, the success factor was improved 2.3 times but it took 4.8 times more to complete the same 1000 iterations. The average was lowered and the standard deviation was narrowed around this lower average.

Func	Gauss	Niter	Npts	Pts/T	To	Sched	FRange	Min	Nmin	Recur.	Time (s)	Mu	Sigma	Start
B	TRUE	1000	6000	1	50	ExpDecay	**10**	-3.306868645	87	FALSE	404.47	-2.806	0.349	(9.0,9.0)
B	TRUE	1000	6000	5	50	ExpDecay	**10**	-3.306868647	199	FALSE	1945.64	-3.044	0.220	(9.0,9.0)

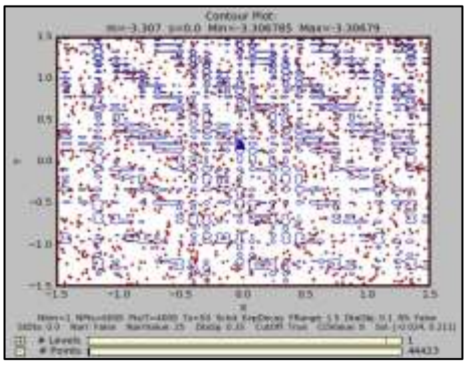
The points per temperature or L, as it is referred in some articles [9] opens the possibility of exploring the search space in 2 different ways: One is to run a single iteration with a large L, while slowly decreasing the temperature, and stop the program when the solution doesn't improve for a number of temperature cycles. In this case it is understood that the system has "frozen" and there is no gain in reducing the temperature further [2]. We will call this the "Cut-off" approach. Another is to run multiple mini-iterations, each finding one minimum. Statistically, if the correct cooling schedule is used and you run the system long enough, a global minimum will be found. We will call this the "Mini-batch" or batch approach.

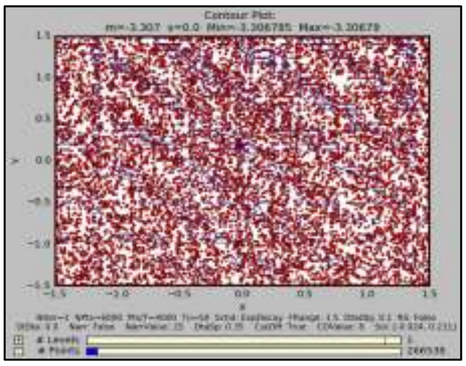
As can be seen on the first row of pictures below, mini-batch processing finds many local minima and depending on the number of batches run, it completely fills the serch space. You can see the many minima found on the 3D plot on the right (darker spots).

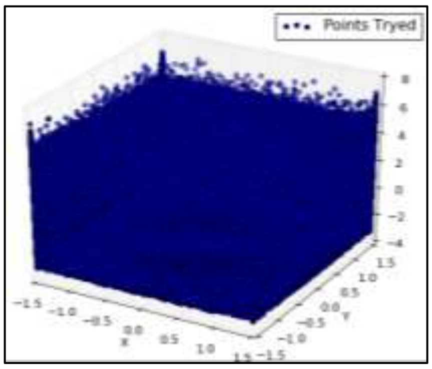
Mini-batch Approach Finds Minima in Stages



Cut-off Approach with High L: Thouroughly Searches Until it "freezes" and finds 1 Minimum







Cut-off processing, on the other hand, extends the exploration phase a lot, due to the high number of points per temperature. It explores the search space evenly, as can be seen on the bottom contour plots.

Both methods have a bit of "guessing" involved. In the Cut-off approach you have to make L large "enough for the system to reach a steady state" [2]. With the mini-batch process, the number of mini-batches has to be guessed.

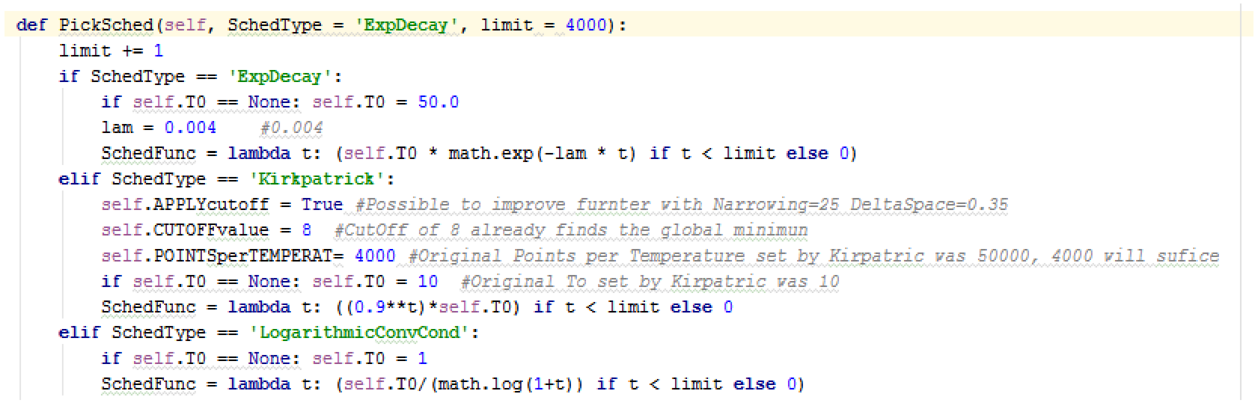
Each approach can be useful in different situations. The mini-batch approach has proven to be valuable when studying how the setting of each parameter affects the performance of the method.

The cut-off approach has the advantage of not having to "guess" how many mini-batches have to be run. It is a "run and forget" method after L and the cut-off have been set. To be able to work in this manner, two new properties were implemented in the SA class: *APPLYcutoff* and *CUTOFFvalue*. The later stops the iteration after *CUTOFFvalue* number of iterations without change in the objective function value.

Cooling Schedule:

There are a number of articles [8][9] discussing different types of cooling schedules that can be used in Simulated Annealing. We felt that it would be unproductive to rehash what those articles have already covered, in this project. We will limit ourselves in studying the two main schedules used.

The figure below shows a procedure that was developed to make switching between cooling schedules easier. Using this function it is possible to set all of the configuration parameters for a determined cooling schedule in one place. To choose which cooling schedule to work with, all that is needed is to set the *TSCHEDULE* property of the SA class.



```
def PickSched(self, SchedType = 'ExpDecay', limit = 4000):
    limit += 1
    if SchedType == 'ExpDecay':
        if self.T0 == None: self.T0 = 50.0
        lam = 0.004     #0.004
        SchedFunc = lambda t: (self.T0 * math.exp(-lam * t) if t < limit else 0)
    elif SchedType == 'Kirkpatrick':
        self.APPLYcutoff = True #Possible to improve furnter with Narrowing=25 DeltaSpace=0.35
        self.CUTOFFvalue = 8  #CutOff of 8 already finds the global minimun
        self.POINTSperTEMPERAT= 4000 #Original Points per Temperature set by Kirpatric was 50000, 4000 will sufice
        if self.T0 == None: self.T0 = 10  #Original To set by Kirpatric was 10
        SchedFunc = lambda t: ((0.9**t)*self.T0) if t < limit else 0
    elif SchedType == 'LogarithmicConvCond':
        if self.T0 == None: self.T0 = 1
        SchedFunc = lambda t: (self.T0/(math.log(1+t)) if t < limit else 0)
```

For this section we will evaluate the cooling schedules for a function range of 1.5 since we saw, in the section "Defining the search Space Size", that SA finds the global minimum equally well for large and small state spaces. Furthermore, we will compare the Cut-off approach, since this was the way the method was originally proposed, with the batch approach where appropriate.

Geometric Cooling and the Kirkpatrick Methodology:

Geometric cooling was the temperature schedule suggested by Kirkpatrick in his article. It is given by the following equation:

$$T_{k+1} = 0.9^k T_0$$

He used the cut-off approach terminating when there was no improvement in three consecutive temperatures. The initial temperature was set to T_0 = 10 and L was set to 50,000.

His methodology was replicated for our sample function and compared with the batch approach. The table below summarizes the results:

Geometric Cooling Results:

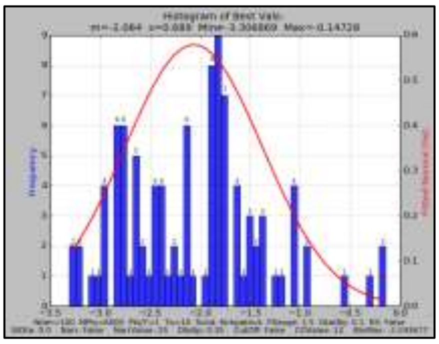
Func	Gauss	Niter	Npts	Pts/T	To	Sched	Frange	Min	Nmin	Time	CutOff	COVal	Start
B	TRUE	100	6000	1	10	Kirkpatrick	1.5	-3.3068686	2	39.91	FALSE		(1.2,1.2)
B	TRUE	100	6000	1	50	Kirkpatrick	1.5	-3.1440794	1	40.10	FALSE		(1.2,1.2)
B	TRUE	100	6000	1	16000	Kirkpatrick	1.5	-3.3068686	1	39.99	FALSE		(1.2,1.2)
B	TRUE	100	6000	1	64000	Kirkpatrick	1.5	-3.3068686	2	39.71	FALSE		(1.2,1.2)
B	TRUE	1	6000	4000	10	Kirkpatrick	1.5	-3.3067284	1	22.77	TRUE	5	(1.2,1.2)
B	TRUE	1	6000	4000	10	Kirkpatrick	1.5	-3.3067199	1	25.86	TRUE	7	(1.2,1.2)
B	TRUE	1	6000	4000	10	Kirkpatrick	1.5	-3.2903795	1	87.55	TRUE	8	(1.2,1.2)
B	TRUE	1	6000	4000	10	Kirkpatrick	1.5	-3.3068686	1	91.83	TRUE	8	(1.2,1.2)
B	TRUE	1	6000	4000	10	Kirkpatrick	1.5	-3.3068686	1	102.93	TRUE	8	(1.2,1.2)
B	TRUE	1	6000	4000	10	Kirkpatrick	1.5	-3.3068686	1	85.10	TRUE	9	(1.2,1.2)
B	TRUE	1	6000	4000	10	Kirkpatrick	1.5	-3.3068686	1	106.91	TRUE	10	(1.2,1.2)
B	TRUE	1	6000	4000	10	Kirkpatrick	1.5	-3.3068686	1	110.68	TRUE	10	(1.2,1.2)

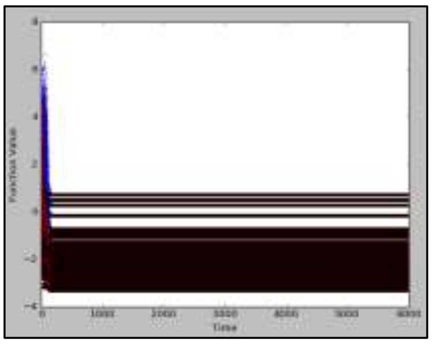
As can be seen, geometric cooling found the global minimum by the batch method (CutOff = False) and the Cutoff method (CutOff = True). What is interesting to notice is that for the same settings (highlighted in blue), different results and execution times can be found. This is one of the difficulties of working with stochastic methods.

The best result with the Cut-off method was with T_0=10, Points per Temperature = 4000 and a Cut-off Value of 9. The objective function was evaluated 1,604,000 times and a global minimum was found with a value of -3.30686864748 in 85 seconds at t_k=419 (highlighted in orange).

In contrast, compared with the batch approach, the solution was found 2 times in 40 seconds. The objective function was evaluated 100x6000=600,000 times, but this batch approach has the disadvantage of not guaranteeing that the global minimum will be in one of the iterations (for this schedule, used with the Cut-off method, there is a statistical proof that this will happen if you run the simulation long enough). Therefore, for functions where the landscape is not known, working with a cut-off in this manner might be the better method, if you can afford the extra computational time.

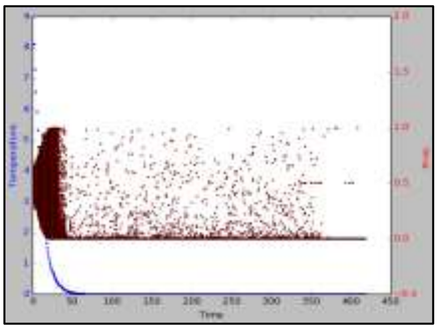
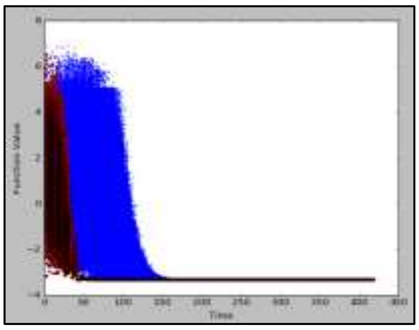
Batch Method





When the batch method is used, it can be seen that this schedule converges too quickly if L is set to 1. Even at a starting temperature of 64000, the global minimum was found only two times.

Cut-Off Method

When using the cutoff approach, L is set high and the simulation has time to explore the search space at each temperature. This can be seen clearly on the function evaluation graph on the right, where the red dots extend further in time on the bottom graph.

Exponential Decay:

The format for this cooling schedule was suggested in the source code samples for the book "Artificial Intelligence a Modern Approach – 3rd Edition" by Russel and Norvig (available at http://aima.cs.berkeley.edu/python/search.html). It is given by the equation:

$$T_{k+1} = T_0 e^{-\lambda t_k}$$

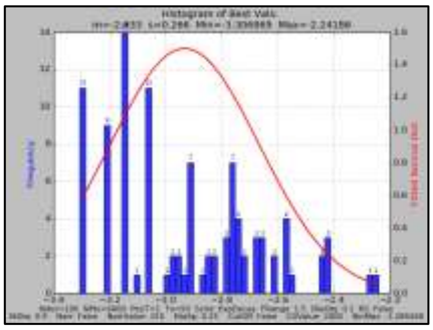
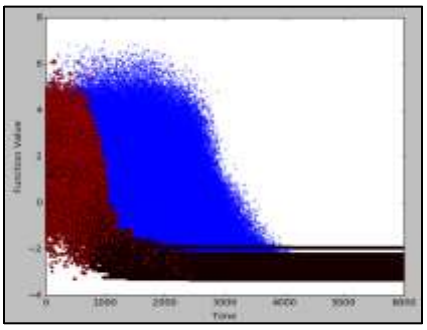
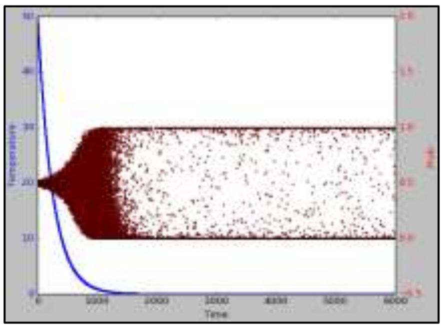
Eight tests were run with the Exponential Decay schedule. Two with batch iteration and six with the cut-off approach. The batch method found the global minimum 11 times in 42 seconds. For this schedule the Cut-off method did not find the global minimum but was pretty close with the best value being -3.3068272. Interestingly enough this configuration was neither the one with the highest Cut off value or the highest temperature. Again, these types of surprises can happen in stochastic methods. The table below summarizes the results for the Exponential decay schedule:

Exponential Decay Results:

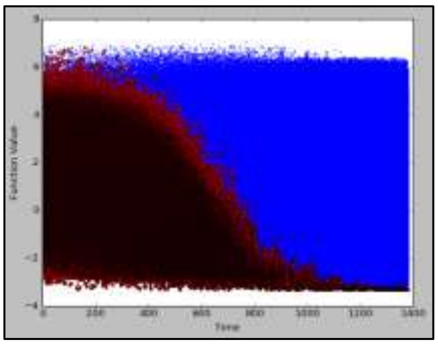
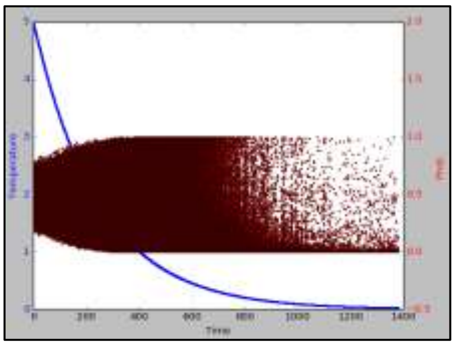
Func	Gauss	Niter	Npts	Pts/T	To	Sched	Frange	Min	Nmin	Time(s)	CutOff	COVal	Start
B	TRUE	3	6000	1	50	ExpDecay	1.5	-3.1440793	1	1.51	FALSE		(1.2,1.2)
B	TRUE	100	6000	1	50	ExpDecay	1.5	-3.3068686	11	42.26	FALSE		(1.2,1.2)
B	TRUE	1	6000	4000	50	ExpDecay	1.5	-3.3067853	1	511.72	TRUE	8	(1.2,1.2)
B	TRUE	1	6000	4000	3	ExpDecay	1.5	-3.3063532	1	227.50	TRUE	8	(1.2,1.2)
B	TRUE	1	6000	4000	3	ExpDecay	1.5	-3.3068346	1	288.72	TRUE	10	(1.2,1.2)
B	TRUE	1	6000	4000	5	ExpDecay	1.5	-3.3068272	1	342.35	TRUE	10	(1.2,1.2)
B	TRUE	1	6000	4000	10	ExpDecay	1.5	-3.3065326	1	335.47	TRUE	10	(1.2,1.2)
B	TRUE	1	6000	4000	12	ExpDecay	1.5	-3.3058954	1	298.61	TRUE	10	(1.2,1.2)
B	TRUE	1	6000	4000	12	ExpDecay	1.5	-3.3066417	1	333.92	TRUE	12	(1.2,1.2)

Different starting temperatures and cut-off values where used in the attempt of finding the true global minimum but for the exponential decay schedule, the batch approach found a better global minimum in less time.

Batch Method

Cut Off Method

It is interesting to note that even for the same schedule, different initial temperatures may be needed depending on the approach chosen to be used in the simulation. For this schedule, for example, T_0 was equal to 50 for the batch approach and 10 for the Cut-off approach.

Implemented Features

State Space Narrowing

State space narrowing reduces (or amplifies if the value is set greater than 1) the search range after a set number of iterations without improvement within the same temperature cycle.

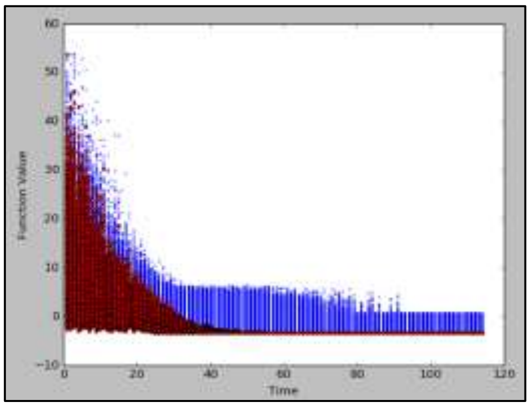
State Space Narrowing is turned on by setting the *STATEspaceNARROWING* property in the SA class to True and setting the number of iterations in the *STATEspaceNARvalue* property. If a better solution is not found in *STATEspaceNARvalue* iterations, the algorithm randomly searches *DELTASpace* % above or below the current position in each dimension. In a second stage, the procedure reopens the search space if a better solution is found near the current coordinate. If the method still has not found a better solution after after 2* *STATEspaceNARvalue* iterations it goes to the best minimum found so far and continues the search.

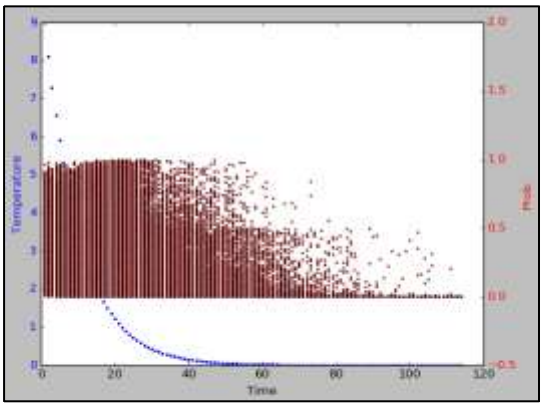
State space narrowing can improve the computational time of the cut-off method or the performance of the batch method.

The table below shows the results for various settings of the *STATEspaceNARvalue* and the *DELTASpace* properties for the Cut-off method:

Func	Gauss	Npts	Pts/T	To	Sched	Frange	Narr	NarrVal	DeltaSp	Min	CutOff	COVal	Time
B	TRUE	6000	4000	10	Kirkpatrick	1.5	TRUE	15	0.35	-3.30686539	TRUE	8	30.24
B	TRUE	6000	4000	10	Kirkpatrick	1.5	TRUE	25	0.15	-3.30686735	TRUE	8	37.57
B	TRUE	6000	4000	10	Kirkpatrick	1.5	TRUE	25	0.35	-3.30686845	TRUE	8	31.36
B	TRUE	6000	4000	10	Kirkpatrick	1.5	TRUE	25	0.35	-3.30686810	TRUE	16	36.93
B	TRUE	6000	4000	10	Kirkpatrick	1.5	TRUE	25	0.35	-3.30686509	TRUE	400	153.75
B	TRUE	6000	4000	10	Kirkpatrick	1.5	TRUE	25	0.35	-3.30686818	TRUE	501	181.90
B	TRUE	6000	4000	10	Kirkpatrick	1.5	TRUE	25	0.35	-3.30686456	TRUE	8	29.97
B	TRUE	6000	4000	10	Kirkpatrick	1.5	TRUE	25	0.35	-3.30685600	TRUE	32	36.95

B	TRUE	6000	4000	10	Kirkpatrick	1.5	TRUE	25	0.45	-3.30685393	TRUE	8	30.77
B	TRUE	6000	4000	10	Kirkpatrick	1.5	TRUE	50	0.25	-3.30686736	TRUE	8	34.05
B	TRUE	6000	4000	10	Kirkpatrick	1.5	TRUE	50	0.35	-3.30686649	TRUE	8	32.01
B	TRUE	6000	4000	10	Kirkpatrick	1.5	TRUE	150	0.25	-3.30685055	TRUE	8	31.73
B	TRUE	6000	4000	10	Kirkpatrick	1.5	TRUE	200	0.25	-3.30686659	TRUE	8	29.27
B	TRUE	6000	4000	10	Kirkpatrick	1.5	TRUE	200	0.25	-3.30686607	TRUE	8	31.98





*It can be seen that narrowing started working around t=80.

When comparing the best run with narrowing and without narrowing (see the table Geometric Cooling Results), the difference was on the 7th decimal place (-3.3068684 against -3.3068686) but it took 53.5 seconds less to compute. As can be observed, increasing the cut-off value does not necessarily improve the quality of the solution. A narrowing value of 25 with a *DELTASpace* of 0.35 and a cut-off of 8 was the best configuration found for the geometric schedule.

When applied to the mini-batch method, the number of iterations is fixed, in our case study to 6000 points. The benefit in this situation comes in the form of an increase in the number of times a solution very close to the global minimum is found. The table below illustrates these results:

Func	Gauss	Niter	Npts	Pts/T	To	Sched	Frange	Narr	NarrVal	DeltaSp	Min	Nmin	Time(s)
B	TRUE	1000	6000	1	16000	ExpDecay	10	FALSE			-3.30686863	89	410.26
B	TRUE	1000	6000	1	16000	ExpDecay	10	TRUE	50	0.25	-3.30686591	93	415.52
B	TRUE	1000	6000	1	16000	ExpDecay	10	TRUE	150	0.15	-3.30686779	92	415.12
B	TRUE	1000	6000	1	16000	ExpDecay	10	TRUE	150	0.25	-3.30686755	95	408.00
B	TRUE	1000	6000	1	16000	ExpDecay	10	TRUE	200	0.25	-3.30686762	105	410.51
B	TRUE	1000	6000	1	16000	ExpDecay	10	TRUE	200	0.35	-3.30686820	102	415.67

The first row is our base line. It shows the exponential decay schedule without narrowing applied. With the use of Gaussian sampling the only parameter that can be set is the initial temperature which was set at 16000, since we know from previous experience that it offers a good balance between “exploration” and “hill climbing” behavior. With narrowing turned off the minimum was found 89 times.

For the same initial temperature with narrowing turned on, the minimum was found 105 times, 16 occurrences above, with the same computational effort.

Recursive Start

Recursive start takes the result of the last iteration and uses as the starting point for the next annealing run. Some noise can be introduced to the start value by setting the property *STARTdelta* of the SA class or by working with a high *DELTASpace* (1.2 for example). This will avoid you getting stuck on the local minimum found in the prior run.

Other Features

Two routines to export the result of the simulation to a CSV file were also implemented. By setting the *WRITEtoFILE* property on, the result of each annealing in a batch process is saved to a file. By setting the *WRITEtoFILEshort* property only the result of the best annealing is saved.

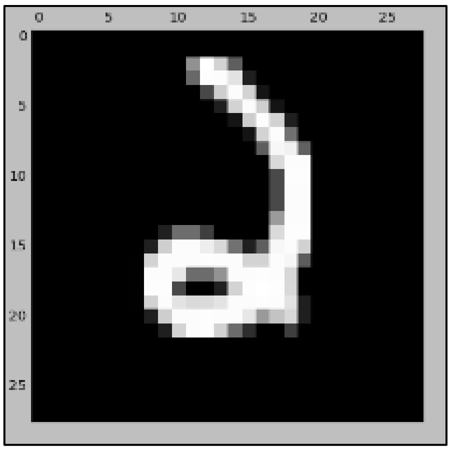
Other features implemented in the SA class were discussed in the relevant sections where they were needed. Examples include Contour Plot, Histogram, 3D plot, stop by cut-off or number of points, probability graph, function value graph.

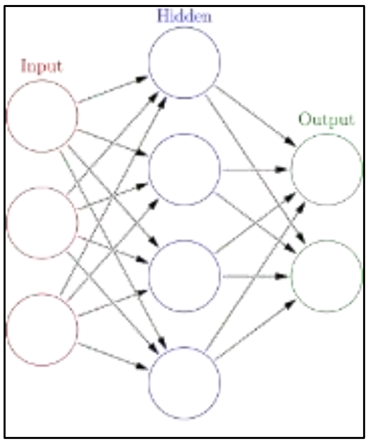
Simulated Annealing Applications

Training a Neural Network to Recognize Handwritten Digits:

Given what was learned in the previous sections, the next step was to apply the algorithm to solve increasingly challenging problems. It was then used to find the best route between 123 cities, a combinatorial problem known as the "Traveling Salesman Problem" and to optimize a small neural network that predicts the score of a student in a test given how many hours he slept and the number of hours he studied. The dissertation for these two sample applications are in the attached slides that were presented in class. The codes for all applications are in appendix B. It is more productive to talk about the third application that was developed since our last meeting.

It consists of a neural network with 844 neurons trained to recognize the MNIST data set. A sample of the data set can be seen on the left and it has 60.000 images with 28 by 28 pixels of handwritten digits. The neural network was structured to have 784 neurons in the input (visible) layer, 50 in the hidden layer and 10 in the output layer.





To train this network, $\varepsilon^{(2)}$ has to be minimized according to the following set of equations:

$$Z^{(2)} = sigmoid(X \cdot W^{(1)})$$
$$OL = sigmoid(Z^{(2)} \cdot W^{(2)})$$
$$\varepsilon = \sum \frac{(-1(Y * \log(OL) + (1-Y)\log(1-OL)))}{n}$$
$$\varepsilon^{(2)} = \varepsilon + \left(\frac{\lambda}{2n}\right)\sum({W^{(1)}}^2) + \sum({W^{(2)}}^2) \qquad (1)$$

Where:

X = Matrix with training data and the visible layer bias vector added
$W^{(1)}$ = Matrix of weights from the visible layer to the hidden layer
$Z^{(2)}$ = Matrix with activation function for the first layer applied and hidden layer bias vector added
$W^{(2)}$ = Matrix of weights from hidden layer to Output Layer
OL = Output Layer with activation function applied
ε = Error
$\varepsilon^{(2)}$ = Error with regularization function applied

This is a very challenging problem because:

> $Z^{(2)}$ has dimension $X \cdot W^{(1)} = [50.000 \times 784] \cdot [784 \times 50] = [50.000 \times 50]$
> OL has dimension $Z^{(2)} \cdot W^2 = [50.000 \times 50] \cdot [50 \times 10] = [50.000 \times 10]$

Besides having these very large matrices, a number of complex functions have to be applied on them.

Training a neural network consists of basically taking your training data and feeding it forward through the network and comparing this result with the training label, which is given in the data set. The error added with a regularization parameter (λ), to reduce over fitting, is used as the objective function. This is the resulting equation shown in (1).

The implemented function also needs to calculate the gradient to be used by a gradient method which could be the BFGS or the Conjugate Gradient. When optimizing with Simulated Annealing, the gradient part does not need to be calculated. The implemented function in Python is shown below and the complete code can be found in appendix B:

```python
def nnObjFunction(params, *args):

    n_input, n_hidden, n_class, training_data, training_label, lambdaval = args

    w1 = params[0:n_hidden * (n_input + 1)].reshape((n_hidden, (n_input + 1)))
    w2 = params[(n_hidden * (n_input + 1)):].reshape((n_class, (n_hidden + 1)))
    obj_val = 0

    training_label = np.array(training_label)
    rows = training_label.shape[0]
    rowsIndex = np.arange(rows, dtype="int")

    tempLabel = np.zeros((rows, 10))
    tempLabel[rowsIndex, training_label.astype(int)] = 1
    training_label = tempLabel

    # nnFeedForwardward propogation
    # adding bias to the input data
    training_data = np.column_stack((training_data, np.ones(training_data.shape[0])))
    number_of_samples = training_data.shape[0]

    # passing the input data to the Hidden layer [calculating a2 = sigmoid(X.W1)]
    zj = sigmoid(np.dot(training_data, w1.T))

    # adding bias to the hidden layer
    zj = np.column_stack((zj, np.ones(zj.shape[0])))
    # passing the hidden layer data to the output layer  [calculating OL or Yhat = sigmoid(a2.W2)]
    ol = sigmoid(np.dot(zj, w2.T))

    # Back propogation
    deltaOutput = ol - training_label
    error = np.sum(-1 * (training_label * np.log(ol) + (1 - training_label) * np.log(1 - ol)))
    error = error / number_of_samples
    gradient_of_w2 = np.dot(deltaOutput.T, zj)
    gradient_of_w2 = gradient_of_w2 / number_of_samples

    gradient_of_w1 = np.dot(((1 - zj) * zj * (np.dot(deltaOutput, w2))).T, training_data)
    gradient_of_w1 = gradient_of_w1 / number_of_samples
    gradient_of_w1 = np.delete(gradient_of_w1, n_hidden, 0)
    obj_grad = np.concatenate((gradient_of_w1.flatten(), gradient_of_w2.flatten()), 0)

    error = error + (lambdaval / (2 * number_of_samples)) * (np.sum(np.square(w1)) + np.sum(np.square(w2)))
    obj_val = error
    return (obj_val, obj_grad)
```

To be able to pass this function to the Annealing optimizer, a new option ‘Z’ had to be created in the dictionary of functions of the procedure f() of the SA class.

```python
def f(self, xVect):     # define objective functions
    x = xVect[self.DimX]
    y = xVect[self.DimY]
    Funct = {'A': 0.7 + x**2 + y**2 - 0.1*math.cos(6.0*3.1415*x) - 0.1*math.cos(6.0*3.1415*y),
             'B': (math.e**math.sin(50*x) + math.sin(60*math.e**y) + math.sin(70*math.sin(x)) +
math.sin(math.sin(80*y)) - math.sin(10*(x + y)) + 0.25*(x**2 + y**2)),
             'C': x**2+ y**2 + 5,
             'D': 10000*(math.e**math.sin(50*x) + math.sin(60*math.e**y) + math.sin(70*math.sin(x)) +
math.sin(math.sin(80*y)) - math.sin(10*(x + y)) + 0.25*(x**2 + y**2)),
             'E': (2*x**6-12.2*x**5+21.2*x**4+6.2*x-6.4*x**3-4.7*x**2+y**6-11*y**5+43.3*y**4-10*y-
74.8*y**3+56.9*y**2-4.1*x*y-0.1*y**2*x**2+0.4*x*y**2+0.4*x**2*y),
             'F': (1-x)**2 + 25*(y-x**2)**2 + (math.cos(3*math.pi*x+4*math.pi*y)+1),
             'Z': (self.CustomFuction(xVect,**self.kwargs)),
             'G': (20*math.sin(x*np.pi/2-2*np.pi) + 20*math.sin(y*np.pi/2-2*np.pi)+(x-2*np.pi)**2+(y-
2*np.pi)**2)}
    return Funct[self.FuncOption]
```

This option and the custom function can be used as a parameter in the call to the SA class:

```
#initialWeights = Anneal(initialWeights)
SimAnneal = sam.SA("Z",costFunctionWrapper2,2,6000,args=args)
initialWeights, SolValue = SimAnneal.Anneal(initialWeights)
```

The commented line calls an annealing function in the file NeuralNetwMNARV2_2.py which was the first version on the annealing procedure developed specifically to solve this neural network. Later, with the improvements in the SA class and the modifications to accept custom functions, the two lines in the bottom became the standard way to call the Annealing optimization.

The Exponential decay cooling schedule was used to solve this problem since the “Kirkpatrick” schedule ran overnight and didn’t finish. For this problem, exponential decay has the advantage that you can set the number of points to sample, which in this case were 6000, with the points per temperature set to 1.

As a base case, the problem was solved initially without annealing, using the conjugate gradient descent, because it was easily available in Python and which was the second best method suggested LeCun [19] for very large problems (stochastic gradient descent was the first).

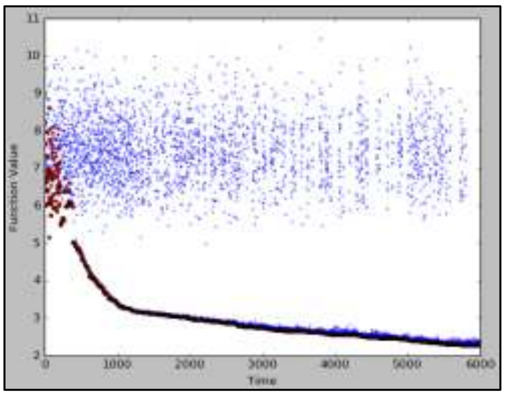
This method worked pretty well and it yielded a result for the full training data set of 50.000 images in 82 seconds. The best minimum found was 0.3414. With this value for the objective function the training set accuracy reached 94.5 % and the test set accuracy (10.000 images) reached 94.3%.

Because Simulated Annealing is a stochastic method, we had already found out from tests in a smaller network with 6 neurons, that the best approach was to use SA to find the general region for the best minimum and the run a gradient descent using the result of the annealing as a starting point. This method worked well for this small network. Combining both methods yielded results better than using each method alone for this simple case.

The MNIST data set requires a much larger computational effort for each evaluation of the objective function. The input layer is a vector with the value of 784 pixels and it requires the dot product of a matrix with 39,200,000 elements with a vector of 39,200 elements and another dot product of a matrix with 2,500,000 elements with a vector of 500 elements. The sigmoid function is then applied on each of these resulting matrices (one with 2,500,000 elements and another with 500,000 elements). Even with these huge sizes, Numpy does a very efficient job in crunching all these numbers and calculating a result in less than a second!

Regardless of all this performance, for the standard configuration that has been used, 6000 evaluations have to happen. Many of these guesses do not improve the solution and are not accepted. With this in mind, the best objective found with the annealing was 2.4474 in 4734 seconds. Much worse than the gradient alone. The function evaluation graph for this simulation is shown below:

Optimization Progress in the MNIST Data Set Training



*50,000 images

The solution with Annealing followed by the conjugate gradient descent took 4807 seconds, yielding a best minimum of 0.4189. With this value for the objective function the training set accuracy reached 93.2 % and the test set accuracy also reached 93.2%. Unlike the 6 neuron test bench example, in this case the annealing was not able to improve the solution found by the gradient descent.

It is believed that because the state space is much larger, 6000 evaluations are not enough to bring the annealing close to the global minimum. Furthermore, because the objective function is much more complex to calculate, and many guesses are wasted, it takes much longer to solve the problem with simulated annealing. Although the number of evaluations could be increased to try to find a better global minimum, there is no point in doing this since the gradient descent finds a better solution with much less effort.

Randomly Reducing the Training Set

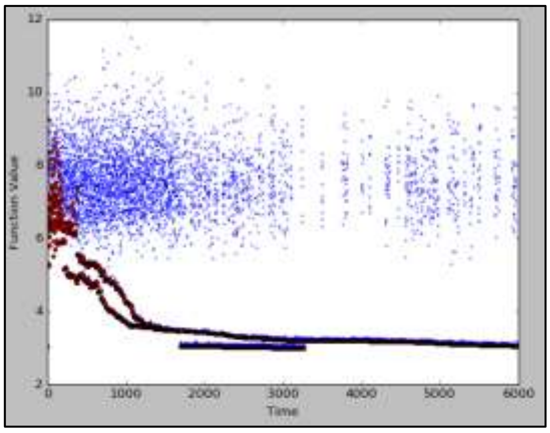
These initial results, led to the decision of creating a smaller set by randomly sampling the full training set. For this purpose, the following procedure was developed:

```
if REDUCEDsample:  #Resize training data set to TRAIN_NEWsize points
    TRAIN_NEWsize = 100
    nsamples = len(train_data)
    x = train_data.reshape((nsamples, -1))
    Y = train_label
    #Create Random indices
    valid_index = random.sample(range(int(len(x))), TRAIN_NEWsize)
    # Resize training  set
    train_data = [x[i] for i in valid_index]
    train_data = np.array(train_data)
    # Resize training label set
    train_label = [Y[i] for i in valid_index]
    train_label = np.array(train_label)
```

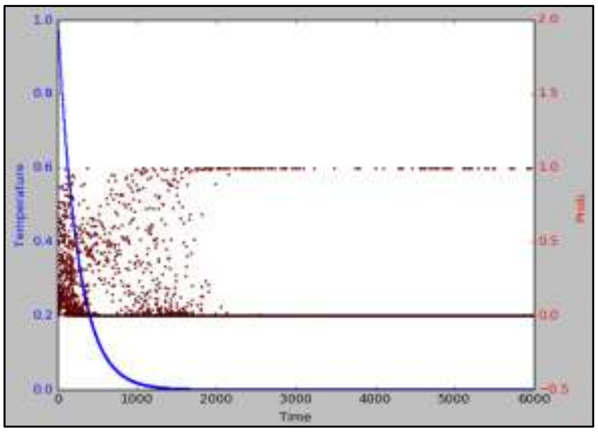
With this procedure, 2 batches were run with recursive input, a *STATEspaceNARROWING* of 10 and a *DELTASpace* of 0.2. The first reduced sample was composed of 100 images:

Optimization Progress in the MNIST Data Set Training

Optimization Progress in the MNIST Data Set Training



Calculated Acceptance Probabilities



*100 images, 2 runs

*100 images

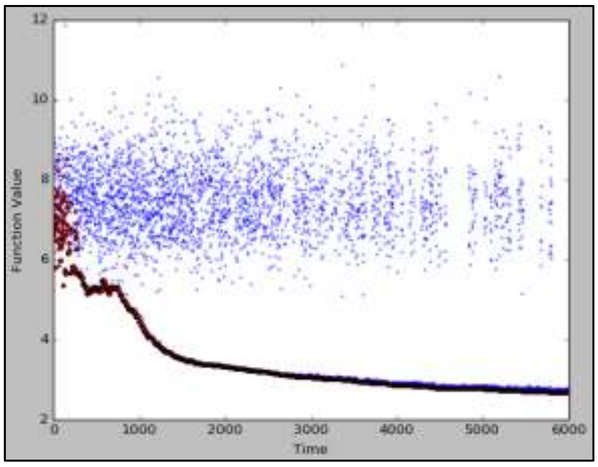
The results of this annealing simulation were:

```
Number of Iterations: 2
T Schedule: ExpDecay
Batch Time elapsed:  934.035000086
Best Batch solution: [ 0.01090648 -0.00233342 ..., -0.02327452  0.02537042 -0.47726067]
Best Batch objective: 2.99680369779
Value after Optimization Gradient: 1.14764284708
Solution: [ 0.01090648 -0.00233342  0.0300867  ...,  0.04218696 -0.64945199]
Training set Accuracy:100.0%
Test set Accuracy: 71.83%
```

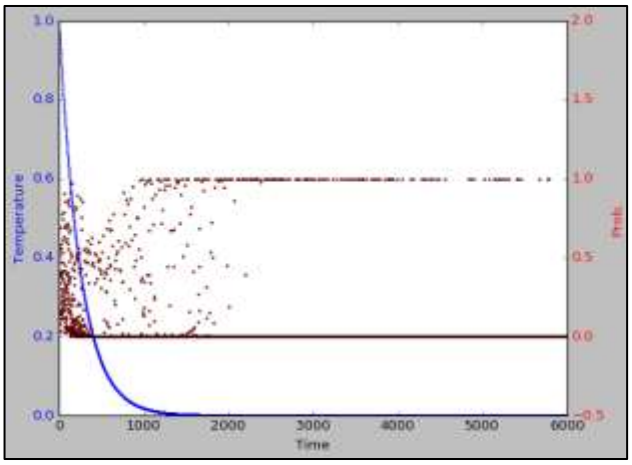
Therefore, even with a smaller training set, the SA takes longer to execute and throws the gradient descent into a local minimum that ends up worsening the final solution.

To attempt to increase the test set accuracy it was decided to try a sample with 1000 images and 1iteration:

Optimization Progress in the MNIST Data Set Training



Calculated Acceptance Probabilities



*1000 images, 1 run

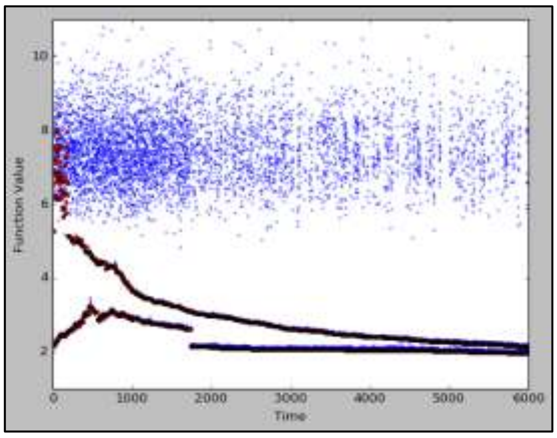
*1000 images

```
Number of Iterations: 1
Best Minimum Found:
T Schedule: ExpDecay
Batch Time elapsed:  635.004999876
Best Batch solution: [-0.00122332 -0.00544729 -0.31673822 ..., -0.00211314  -0.14853024]
Best Batch objective: 2.68114393911
Value after Optimization Gradient: 0.280678803599
Solution: [-0.00122332 -0.00544729 -0.31673822 ..., 0.16805153 -0.51030969]
Training set Accuracy:99.8%
Test set Accuracy:88.17%
```

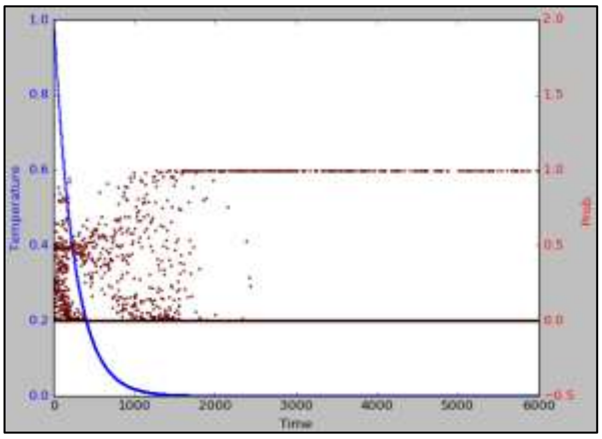
The decrease in the value of the optimization after the application of the gradient descent to 0.28 and increase in the test set accuracy to 88.17% encouraged 1 more run with 10,000 images and two iterations. The results are shown below:

Optimization Progress in the MNIST Data Set Training

Calculated Acceptance Probabilities



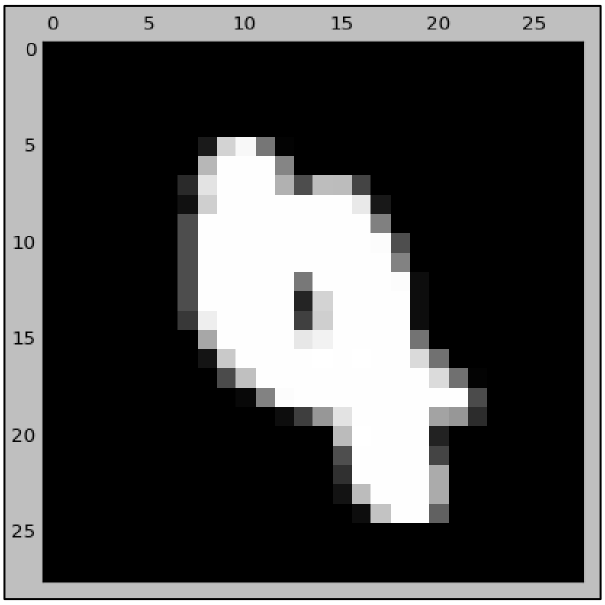
*10,000 images, 2 runs



*10,000 images

```
Number of Iterations: 2
T Schedule: ExpDecay
Batch Time elapsed:  3094.13000011
Best Batch solution: [ 0.00021444  0.02645701 -0.01578192 ..., -0.00056168  0.0140971 ]
Best Batch objective: 1.9764822912
Value after Optimization Gradient: 0.385678509152
Solution: [  2.14441069e-04   2.64570064e-02  -1.57819161e-02 ..., -9.14066114e-01  -4.41101697e-01]
Training set Accuracy:94.8%
Test set Accuracy:92.63%
```

Therefore, even with this configuration the combined methods found a best objective value of 0.3857. This is still worse than the conjugate gradient value of 0.3414 and it took 38 times longer to compute. However, the valuable lesson learned was that 10000 images seem to be a good sample size to represent the full data set. An accuracy of 92.63% was achieved and the system was able to correctly recognize one of the hard images of the data set:

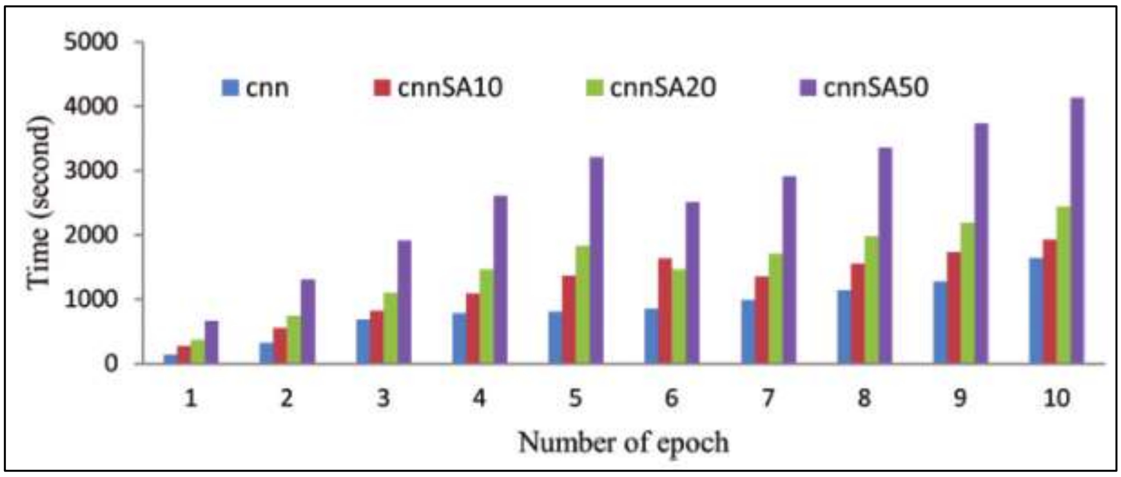


*The image above was correctly identified as a 4 by the NN trained with SA.

```
Enter a Test image index from 0 to 9999: 5000
Test Label:  4
Predicted Label:  4
```

In order to find new solutions to the problem, the next approach tried was to search the literature for articles on neural networks trained by Simulated Annealing. Initially three articles were revised [20][21][22] but none of them gave enough details on the network optimization portion of the problem so that it could be implemented. In fact, Rere *et al* [20] also found that SA increased the computational time to solve, in his case, convolutional networks.

Rere compared the computational times of solving the original cnn with Simulated Annealing exploring 3 different neighborhood sizes of 10, 20 and 50 units (equivalent to *DELTAObjFunc* in our SA class). Because he has a Δx type parameter in his implementation, this indicates that he used uniform sampling in his methodology since Gaussian sampling only requires the temperature as its input.



He concludes by mentioning "Illustration from this chart shows that the original CNN (cnn) is better than CNN by SA and the time will be increased by increasing the size of the neighborhood on SA", which is coherent with our findings.

CONCLUSIONS

In this paper many of the virtues and difficulties of working with Simulated Annealing were reviewed. It is an extremely flexible and powerful method for optimizing functions. Its flexibility is what makes it powerful and at the same time creates the difficulties of working with the method. The most obvious parameters, which are usually mentioned in the literature, are the initial temperature and the cooling schedule.

However, we have discovered that other settings also greatly impact the performance of the Simulated Annealing algorithm. Among them we can note:

- Choice of how you will sample the search space: Uniform or Gaussian distribution;
- When using uniform sampling: setting the step size (Δx, Δy) can alter the behavior of the search;
- For either choice of sampling: setting the points/temperature and cut-off value;
- Deciding if you will use a multiple quick runs (batch) or 1 long run (cut-off) approach;
- And as has been metioned, choosing T_0 and the cooling schedule.

It is not always obvious how to combine these settings to reduce the execution time or increase the chances of finding the global minimum, especially for multidimensional problems. With the purpose of understanding how these variables impact the SA algorithm, each setting was individually tested and evaluated in regards to its performance.

After understanding the method, and creating some tools to help fine tune the parameters for each problem, a combinatorial problem (TSP) and two neural networks were optimized. As a sample application, only the training of the larger neural network was discussed in this document since it was the most challenging.

Simulated Annealing worked very well to optimize two dimensional functions and the combinatorial problem. It optimized the TSP problem with 123 cities in 90 seconds. The challenges started with the training of the smaller neural network which had 6 neurons and 9 dimensions to optimize. In this case the best approach was to search the state space with SA and then hand of the best minimum to a gradient descent method which would then improve the solution found by the annealing process. This combined approach yielded results better than using each method by itself.

SA was then used to optimize a fairly large neural network with 844 neurons and 50.000 images in the training data set. In this case the method runs into the challenge of the computational time to evaluate the objective function. For this problem each evaluation takes almost a second to calculate. To run an annealing with 6000 points took over 4700 seconds, much longer than just using the gradient descent method alone.

A solution was tried by sampling the training data set and making it smaller to reduce the time to compute the objective functions. Traning sets with 100, 1000 and 10,000 images were created and optimized. The smaller sets did not generate accuracy high enough and the largest data set, although comparably as accurate as the gradient method, still took too long to compute (3094s versus 82s with gradient descent alone). This conclusion was reinforced by other literature found on the topic of training neural networks with SA [20].

APPENDIX A – REFERENCES

1. Wikipedia.com (2016, September 26th), Traveling Sales Problem. Available in < https://en.wikipedia.org/wiki/Travelling_salesman_problem >. Accessed on September 27th, 2016.
2. Kirkpatric, S.; Gellat, C.D.; Vecchi M. P. (1983, May 13th). Optimization by Simulated Annealing. *Science Magazine*, United States, V.220, Issue 4598, p. 671-680, 1983. Available in < http://citeseer.ist.psu.edu/viewdoc/download?doi=10.1.1.123.7607&rep=rep1&type=pdf >
3. Wikipedia.com (2016, September 26th), Knapsack Problem. Available in < https://en.wikipedia.org/wiki/Knapsack_problem >. Accessed in September 27th, 2016.
4. Ingber, L (1993). Simulated Annealing: Practice versus theory. Journal of Mathematical Computational Modeling, United States, V.18, p. 29-57.
5. Ingber, L (1989). Very Fast Simulated Re-Annealing. Journal of Mathematical Computational Modeling, United states, V.12, p. 967-973.
6. Stander, N.; Roux, W.; Goel, T. (2012). LS-OPT A Design Optimization and Probabilistic Analysis Tool for the Engineering Analyst. *Livenmore Software Technology Corporation*, United States, Version 4.2, p. 87-92.
7. Russel, S.; Norvig P. (2010). Artificial Intelligence a Modern Approach, United States, 3rd ed. Prentice hall.
8. What-when-how.com (2016), A Comparison of Cooling Schedules for Simulated Annealing (Artificial Intelligence). Available in < http://what-when-how.com/artificial-intelligence/a-comparison-of-cooling-schedules-for-simulated-annealing-artificial-intelligence/ >. Accessed in November 11th, 2016.
9. Ortner, M.; Descombes, X.; Zerubia J. (2007, October). An Adaptive Simulated Annealing Cooling Schedule for Object Detection in Images. *Rapport de Recherche*, France, N.6336.
10. Mathworks.com (2016), How Simulated Annealing Works. Available in < http://www.mathworks.com/help/gads/how-simulated-annealing-works.html >. Accessed on September 27th,2016.
11. Scipy.org (2016, September 26th), Scipy.optimize.anneal. Available in < http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.optimize.anneal.html >Accessed on September 27th,2016.
12. Scipy.org (2016, September 26th), scipy.optimize.basinhopping. Available in < http://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.optimize.basinhopping.html >. Accessed on September 27th,2016.
13. http://doye.chem.ox.ac.uk (1997), Global Optimization by Basin-Hopping and the Lowest Energy Structures of Lennard-Jones Clusters Containing up to 110 Atoms. Available in < http://doye.chem.ox.ac.uk/abstracts/jpc97.html > . Accessed on September 27th, 2016.
14. Ocw.mit.edu (2016), Simulated Annealing, a Basic Introduction. Available in < https://ocw.mit.edu/courses/engineering-systems-division/esd-77-multidisciplinary-system-design-optimization-spring-2010/lecture-notes/MITESD_77S10_lec10.pdf >. Accessed on September 27th, 2016.
15. App.cs.amherst.edu (2016), Simulated Annealing Algorithms. Available in < https://app.cs.amherst.edu/~ccmcgeoch/cs34/papers/rutenbar.pdf >. Accessed in September 27th,2016.
16. Wikipedia.com (2016, September 22nd), Simulated Annealing. Available in < https://en.wikipedia.org/wiki/Simulated_annealing) >. Accessed on September 27th, 2016.
17. Wikipedia.com (2015, October 22nd), Adaptive Simulated Annealing. Available in < https://en.wikipedia.org/wiki/Adaptive_simulated_annealing >. Accessed on September 27th, 2016.

18. http://www.sciencedirect.com (1990, June 15th), Simulated annealing: A tool for operational research. Available in < http://www.sciencedirect.com/science/article/pii/037722179090001R > Accessed on September 27th, 2016.
19. LeCun, Y.; Bottou, L.; Orr G. B. (1998). Efficient BackProp. *Image Processing Research Department AT&T Labs*, United States, p. 1-44, 1998.
20. Rere, L.M.; Fanany, M.; Arymurthy, A. (2015). Simulated Annealing Algorythm for Deep Learning. *Procedia Computer Science*, Indonesia, V. 72, p. 137-144.
21. Sexton, R.; Dorsey, R.E.; Johnson J. D. (1999). Beyond Backpropagation: Using Simulated Annealing For Training Neural Networks. *Journal of Organizational and End User Computing (JOEUC)*, V.11 (I.3), p. 3-10,1999
22. Bernal, J.; Torrez-Jimenez, J. (2015, June 17th). SAGRAD: A Program for Neural Network Training with Simulated Annealing and the Conjugate Gradient Method. *Journal of the National Institute of Standards and Technology*, Mexico, V. 120 (2015), p. 113-128.

APPENDIX B – SOURCE CODE

File RodsAnnealMultiV2_2.py

```
from __future__ import print_function
import numpy as np
import math
import csv
import time
from matplotlib import cm
import matplotlib.animation as animation
from matplotlib.ticker import LinearLocator, FormatStrFormatter
from mpl_toolkits.mplot3d import axis3d
import matplotlib.pyplot as plt
import random
#from functools import partial

def DefaultCustomFunc(xVect,**kwargs):
    x = xVect[0]
    y = xVect[1]
    if kwargs["FuncName"] == "A":
        Cost = x**2+y**2
    if kwargs["FuncName"] == "B":
        Cost = x+y
    return Cost

# def Perform(funct, **kwargs):
#     funct(kwargs)

class SA():
    def __init__(self, FuncOption = 'B',CustomFuction=DefaultCustomFunc, NUMITER=1, NUMpoints=6000,
DimX=0,DimY=1,**kwargs):
        self.Xs = []
        self.Ys = []
        self.Zs = []
        self.DimX = DimX
        self.DimY = DimY
        self.Xtime = []
        self.Ytemperature = []
        self.YtempLimit = []
        self.YProbGraph = []
        self.YbestOBJ = []
        self.YOBJ = []
        self.ItValue = []
        self.ItNumber = 0
        # System Parameters
        self.FuncOption = FuncOption
        self.CustomFuction = CustomFuction
        self.kwargs = kwargs
        self.FIXseed = False  # establishes random seeds for all random numbers
        self.OBJfunctionRANGE = 10  # Explores Objetive function from -OBJfunctionRANGE to +OBJfunctionRANGE
in x and y directions
        self.DELTAObjFunc = 0.1    # Used to calculate the next guess for x and y:  xn = xbest +
random.uniform(-DeltaObjFunc*OBJfunctionRANGE, DeltaObjFunc*OBJfunctionRANGE)
        self.WRITEtoFILE = False  # Exports the result of each Anneal Iteration to .csv file
        self.WRITEtoFILEshort = True  #Exports only the final result of all the Anneal Iterations
        self.GRAPH = 0
        self.GRAPHsurface = 0
        self.GRAPHresolution = 0.01
        self.GRAPHtemperSCHED = 1
        self.GRAPHProb = 1
        self.GRAPHobjValue = 1
        self.GRAPHhistogram = 1
        self.GRAPHcontour = 0
        self.PRINTiter = 0
        self.PRINTSOLUTION = 1
        self.PRINTbatchSOLUTION = True
        self.NUMpoints = NUMpoints  # Number of random points per Annealing simulation
        self.POINTSperTEMPERAT = 1
        self.NUMITER = NUMITER  # Number of times to run the Annealing Function
        self.APPLYcutoff = False
        self.CUTOFFvalue = 8
        self.GAUSSIANsampling = True
        self.STATEspaceNARROWING = True  # Turns space narrowing on and off
        self.STATEspaceNARvalue = 150.0  # Narrows the search space if Best Solution has not chenged in
STATEspaceNARvalue trials
        self.DELTASpace = 0.15
        self.TSCHEDULE = 'ExpDecay'
        self.T0 = 16000
        self.RECURSIVEstart = False
        self.STARTdelta = 0.0
    def PickSched(self, SchedType = 'ExpDecay', limit = 4000):
        limit += 1
        if SchedType == 'ExpDecay':
            if self.T0 == None: self.T0 = 50.0
            lam = 0.004     #0.004
            SchedFunc = lambda t: (self.T0 * math.exp(-lam * t) if t < limit else 0)
        elif SchedType == 'Kirkpatrick':
            self.APPLYcutoff = True #Possible to improve furnter with Narrowing=25 DeltaSpace=0.35
            self.CUTOFFvalue = 501  #CutOff of 8 already finds the global minimun
            self.POINTSperTEMPERAT= 4000 #Original Points per Temperature set by Kirpatric was 50000, 4000
will sufice
            if self.T0 == None: self.T0 = 10  #Original To set by Kirpatric was 10
            SchedFunc = lambda t: ((0.9**t)*self.T0) if t < limit else 0
        elif SchedType == 'LogarithmicConvCond':
            if self.T0 == None: self.T0 = 1
            SchedFunc = lambda t: (self.T0/(math.log(1+t)) if t < limit else 0)
        elif SchedType == 'ConvCond-Log2':
            if self.T0 == None: self.T0 = 1
            SchedFunc = lambda t: (self.T0/(1+math.log(1+t)) if t < limit else 0)
        elif SchedType == 'Geometric':
            if self.T0 == None: self.T0 = 100000.0
            SchedFunc = lambda t: (self.T0 * 0.996**t) if t < limit else 0
        elif SchedType == 'LogMultip':
            if self.T0 == None: self.T0 = 5
            Alpha = 1.0001 # Alpha>1
            SchedFunc = lambda t: (self.T0/(1+Alpha*math.log(1+t)) if t < limit else 0)
        elif SchedType == 'LinearMultip':
            if self.T0 == None: self.T0 = 2000.0
            Alpha = 2  # Alpha>0
            SchedFunc = lambda t: (self.T0/(1+Alpha*t) if t < limit else 0)
        elif SchedType == 'QuadraticMultip':
            if self.T0 == None: self.T0 = 4000.0
            Alpha = 1.2  # Alpha>0
            SchedFunc = lambda t: (self.T0/(1+Alpha*t**2) if t < limit else 0)
        elif SchedType == 'LinearInvTime':
             if self.T0 == None: self.T0 = 2000.0
             SchedFunc = lambda t: (self.T0/t if t < limit else 0)
        elif SchedType == 'DivFactor':
             if self.T0 == None: self.T0 = 16000.0
             Mu = 1.004
             SchedFunc = lambda t: (self.T0/(Mu**t) if t < limit else 0)
        self.TSCHEDULE = SchedType
        return SchedFunc

    def Anneal(self, StartC=[0.0,0.0]):
        if self.WRITEtoFILE:
            self.csvADD(['ITER', 't', 'Points', 'x', 'y', 'Objective', 'To', 'Schedule:', self.TSCHEDULE,
'ObjRange:',
                    self.OBJfunctionRANGE,
                    'Narrowing:', self.STATEspaceNARROWING, 'NarrValue:', self.STATEspaceNARvalue,
                    'DeltaSp:', self.DELTASpace, 'CutOff:', self.APPLYcutoff, 'COvalue:', self.CUTOFFvalue,
'Points/T:',
                    self.POINTSperTEMPERAT])
        if self.FIXseed: random.seed(0.55)
        else: random.seed()
        if StartC == []:
            StartC = np.random.uniform(-self.OBJfunctionRANGE, self.OBJfunctionRANGE, 2)
            #StartC = np.zeros(len(StartC))
        Bxbest = StartC
        Bfbest = self.f(Bxbest)
        batchTime = time.time()
        ItValue=[]
        xns = np.zeros(len(StartC))
        for self.ItNumber in range(self.NUMITER):
            print("It:", self.ItNumber + 1)
            print("Starting Coordinate:",StartC)
            Ixbest, Ifbest = self.simulated_annealing(self.PickSched(self.TSCHEDULE, self.NUMpoints), StartC)
            if self.GRAPHhistogram or self.WRITEtoFILEshort or self.GRAPHcontour: ItValue.append(Ifbest)
            if Ifbest < Bfbest: Bxbest, Bfbest = Ixbest, Ifbest
            if self.RECURSIVEstart:
                if self.FIXseed: random.seed(self.ItNumber)
                for nd in range(len(StartC)):
                    xns[nd] = (Bxbest[nd] + random.uniform(-self.STARTdelta, self.STARTdelta))
                    if xns[nd] < (-1 * self.OBJfunctionRANGE): xns[nd] = (-1 * self.OBJfunctionRANGE)
                    if xns[nd] > self.OBJfunctionRANGE: xns[nd] = self.OBJfunctionRANGE
                StartC = np.array(xns)

        BFTime = time.time() - batchTime
        if self.PRINTbatchSOLUTION:
            print("")
            print("Number of Iterations:", self.NUMITER)
            print("Number of Points / Iteration: ", self.NUMpoints)
            print("T Schedule:", self.TSCHEDULE)
            print("Batch Time elapsed: ", BFTime)
            print('Best Batch solution: ' + str(Bxbest))
            print('Best Batch objective: ' + str(Bfbest))

        if self.GRAPHtemperSCHED:
            fig, ax1 = plt.subplots()
            ax1.plot(self.Xtime, self.Ytemperature, 'b.', markersize=2)
            ax1.set_xlabel('Time')
            # Make the y-axis label and tick labels match the line color.
            ax1.set_ylabel('Temperature', color='b')
            for tl in ax1.get_yticklabels():
                tl.set_color('b')
            if False:  #if True plots To/ln(t) limit
                ax1.plot(self.Xtime,self.YtempLimit, 'g.', markersize=1)
                ax1.set_ylim([0,self.T0])
            if self.GRAPHProb:
                ax2 = ax1.twinx()
                ax2.plot(self.Xtime, self.YProbGraph, 'ro', markersize=2)
                ax2.set_ylabel('Prob.', color='r')
                ax2.set_ylim([-0.5, 2])
                for tl in ax2.get_yticklabels():
                    tl.set_color('r')

        if self.GRAPHobjValue:
            fig, ax1 = plt.subplots()
            ax1.plot(self.Xtime, self.YOBJ, 'bx', markersize=2)
            ax1.plot(self.Xtime, self.YbestOBJ, 'ro', markersize=3)
            ax1.set_xlabel('Time')
            # Make the y-axis label and tick labels match the line color.
            ax1.set_ylabel('Function Value')

        if self.GRAPH:
            fig = plt.figure()
            ax1 = fig.add_subplot(111, projection='3d')
            ax1.scatter(self.Xs, self.Ys, self.Zs, c='b', marker='o')
            ax1.legend(['Points Tryed'])
            ax1.set_xlim([-self.OBJfunctionRANGE, self.OBJfunctionRANGE])
            ax1.set_ylim([-self.OBJfunctionRANGE, self.OBJfunctionRANGE])
            ax1.set_xlabel('X')
            ax1.set_ylabel('Y')

        if self.GRAPHhistogram or self.WRITEtoFILE or self.WRITEtoFILEshort or self.GRAPHcontour:
            ItValue = np.array(ItValue)
            mu = np.mean(ItValue)
            sigma = np.std(ItValue)
            MaxValue = np.max(ItValue)
            MinValue = np.min(ItValue)
            #OverallMax = np.max(self.Zs)
            #OverallMin = np.min(self.Zs)
            #print("Max by Iteration: ",MaxValue,"   Max Achieved: ", OverallMax)
            #print("Min by Iteration: ", MinValue,"   Min Achieved:", OverallMin)
            if self.WRITEtoFILE or self.WRITEtoFILEshort : self.GRAPHhistogram=True

        if self.GRAPHsurface or self.GRAPHcontour:
            # Design variables at mesh points
            i1 = np.arange(-self.OBJfunctionRANGE, self.OBJfunctionRANGE, self.GRAPHresolution)
            i2 = np.arange(-self.OBJfunctionRANGE, self.OBJfunctionRANGE, self.GRAPHresolution)
            x1m, x2m = np.meshgrid(i1, i2)
            fm = np.zeros(x1m.shape)
            for i in range(x1m.shape[0]):
                for j in range(x1m.shape[1]):
                    xVect=[x1m[i][j], x2m[i][j]]
                    fm[i][j] = self.f(xVect)

        if self.GRAPHcontour:
            from matplotlib.widgets import Slider, Button
            # Create a contour plot
            axis_color = 'lightgoldenrodyellow'
            figC = plt.figure()
            axC = figC.add_subplot(111)
            figC.subplots_adjust(bottom=0.21)
            NumLines = 190
            def DrawContour():
                CS = axC.contour(x1m, x2m, fm, int(NumLines_Slider.val), cmap=cm.jet)
                title_string = 'Contour Plot: \nm=' + str(round(mu, 3)) + '  s=' + str(
                    round(sigma, 3)) + '  Min=' + str(round(MinValue, 6)) + \
                               '  Max=' + str(round(MaxValue, 5))
                axC.set_title(title_string, fontsize=13)
```

```
                axC.set_xlim([-self.OBJfunctionRANGE, self.OBJfunctionRANGE])
                axC.set_ylim([-self.OBJfunctionRANGE, self.OBJfunctionRANGE])
                axC.set_xlabel('X')
                axC.set_ylabel('Y')
                axC.plot(Bxbest[0], Bxbest[1], 'b^', markersize=13)
            NumLines_Slider_ax = figC.add_axes([0.19, 0.05, 0.65, 0.02], axisbg=axis_color)
            NumLines_Slider = Slider(NumLines_Slider_ax, '# Levels', 1, 200, valinit=NumLines, valfmt="%.0f")
            NumPoints_Slider_ax = figC.add_axes([0.19, 0.02, 0.65, 0.02], axisbg=axis_color)
            NumPoints_Slider = Slider(NumPoints_Slider_ax, '# Points', 0, len(self.Xs), valinit=0,
valfmt="%.0f")
            DrawContour()
            plt.figtext(.50, 0.09,
                        'Niter=' + str(self.NUMITER) + '  NPts=' + str(self.NUMpoints) + '  Pts/T=' + str(
                            self.POINTSperTEMPERAT) +
                        '  To=' + str(int(self.T0)) + '  Schd: ' + self.TSCHEDULE + '  FRange: ' + str(
                            self.OBJfunctionRANGE) + '  DtaObj: ' + str(self.DELTAObjFunc) + '  RS: ' + str(
                            self.RECURSIVEstart) +
                        '\nStDta: '+str(self.STARTdelta)+ '    Narr: ' + str(self.STATEspaceNARROWING) + '
NarrValue: ' + str(
                            int(self.STATEspaceNARvalue)) +
                        '    DtaSp: ' + str(self.DELTASpace) + '    CutOff: ' + str(
                            self.APPLYcutoff) + '    COValue: ' + str(
                            self.CUTOFFvalue)+ '    Sol: '+str([round(float(Bxbest[self.DimX]),3),
round(float(Bxbest[self.DimY]),3)]), fontsize=10, ha='center')
            #axC.plot(self.Xns, self.Yns, 'bx', markersize=4)
            axC.plot(self.Xs, self.Ys, 'ro', markersize=3)
            # Add a button
            Plus_button_ax = figC.add_axes([0.04, 0.05, 0.02, 0.02])
            Plus_button = Button(Plus_button_ax, '+', color=axis_color, hovercolor='0.975')
            def Plus_button_on_clicked(mouse_event):
                NumPoints_Slider.set_val(int(NumPoints_Slider.val)+1)
                #print(NumPoints_Slider.val)
                axC.plot(self.Xs[int(NumPoints_Slider.val)], self.Ys[int(NumPoints_Slider.val)], 'ro',
markersize=3)
                #axC.plot(self.Xns[0:int(NumPoints_Slider.val)], self.Yns[0:int(NumPoints_Slider.val)], 'bx',
markersize=4)
            Plus_button.on_clicked(Plus_button_on_clicked)

            Minus_button_ax = figC.add_axes([0.04, 0.02, 0.02, 0.02])
            Minus_button = Button(Minus_button_ax, '-', color=axis_color, hovercolor='0.975')
            def Minus_button_on_clicked(mouse_event):
                NumPoints_Slider.set_val(int(NumPoints_Slider.val)-1)
                #print(NumPoints_Slider.val)
                axC.plot(self.Xs[int(NumPoints_Slider.val)], self.Ys[int(NumPoints_Slider.val)], 'ro',
markersize=3)
            Minus_button.on_clicked(Minus_button_on_clicked)
            def sliders_on_changed(val):
                axC.clear()
                DrawContour()
                axC.plot(self.Xs[0:int(NumPoints_Slider.val)], self.Ys[0:int(NumPoints_Slider.val)], 'ro',
markersize=3)
                #axC.plot(self.Xns[0:int(NumPoints_Slider.val)], self.Yns[0:int(NumPoints_Slider.val)], 'bx',
markersize=4)
                plt.draw()
            NumLines_Slider.on_changed(sliders_on_changed)
            NumPoints_Slider.on_changed(sliders_on_changed)

        if self.GRAPHsurface:
            # Design variables at mesh points
            i1 = np.arange(-self.OBJfunctionRANGE, self.OBJfunctionRANGE, self.GRAPHresolution)
            i2 = np.arange(-self.OBJfunctionRANGE, self.OBJfunctionRANGE, self.GRAPHresolution)
            x1m, x2m = np.meshgrid(i1, i2)
            fm = np.zeros(x1m.shape)
            for i in range(x1m.shape[0]):
                for j in range(x1m.shape[1]):
                    xVect=[x1m[i][j], x2m[i][j]]
                    fm[i][j] = self.f(xVect)
            fig = plt.figure()
            ax1 = fig.gca(projection='3d')
            ax1.plot_surface(x1m, x2m, fm, cmap=cm.jet)
            ax1.set_xlabel('X')
            ax1.set_xlim([-self.OBJfunctionRANGE, self.OBJfunctionRANGE])
            ax1.set_ylim([-self.OBJfunctionRANGE, self.OBJfunctionRANGE])
            ax1.set_ylabel('Y')

        if self.GRAPHhistogram:
            import matplotlib.mlab as mlab
            # the histogram of the data
            fig = plt.figure()
            ax1 = fig.add_subplot(111)
            numBins = 50
            n, bins, patches = ax1.hist(ItValue, numBins, color='blue', alpha=0.8)
            plt.grid(True)
            rects = ax1.patches
            # Now make some labels
            for rect in rects:
                height = rect.get_height()
                if height > 0:
                    ax1.text(rect.get_x() + rect.get_width() / 2, 1.01 * height, '%d' % int(height),
ha='center',
                             va='bottom', fontsize=8)
            # add a 'best fit' line
            ItValue = np.array(ItValue)

            MaxValue = np.max(ItValue)
            MinValue = np.min(ItValue)
            bincenters = 0.5 * (bins[1:] + bins[:-1])
            SmallestBin = bins[0:2]
            #print(bins[0:2])
            ax2 = ax1.twinx()
            if self.NUMITER > 1:
                mu = np.mean(ItValue)
                sigma = np.std(ItValue)
                NormCurve = mlab.normpdf(bincenters, mu, sigma)
                l = ax2.plot(bincenters, NormCurve, 'r-', linewidth=2)
            title_string = 'Histogram of Best Vals: \nm=' + str(round(mu, 3)) + '  s=' + str(
                round(sigma, 3)) + '  Min=' + str(round(MinValue, 6)) + \
                           '  Max=' + str(round(MaxValue, 5))
            plt.figtext(.5, 0.02,
                        'Niter=' + str(self.NUMITER) + '  NPts=' + str(self.NUMpoints) + '  Pts/T=' +
str(self.POINTSperTEMPERAT) +
                        '  To=' + str(int(self.T0)) + '  Schd: ' + self.TSCHEDULE + '  FRange: ' + str(
                            self.OBJfunctionRANGE) + '  DtaObj: ' + str(self.DELTAObjFunc) + '  RS: ' +
str(self.RECURSIVEstart) +
                        '\nStDta: '+str(self.STARTdelta) + '    Narr: ' + str(self.STATEspaceNARROWING) + '
NarrValue: ' + str(int(self.STATEspaceNARvalue)) +
                        '    DtaSp: ' + str(self.DELTASpace) + '    CutOff: ' + str(self.APPLYcutoff) + '
COValue: ' + str(
                            self.CUTOFFvalue) + '    BinMax: ' + str(round(bins[1], 6)), fontsize=10,
ha='center')
            plt.title(title_string, fontsize=13)
            #ax2 = ax1.twinx()
            #l = ax2.plot(bincenters, NormCurve, 'r-', linewidth=2)
            ax1.set_ylabel('Frequency', color='b')
            ax2.set_ylabel('Fitted Normal Dist.', color='r')
            Nmim=(int(ax1.patches[0].get_height()))
            print(Nmim, SmallestBin)

        if self.WRITEtoFILEshort:
            self.csvADD(
                ['Func','Gauss' ,'Niter', 'Npts', 'Pts/T', 'To', 'Sched', 'Frange', 'DeltaObjF', 'Narr',
'NarrVal', 'DeltaSp',
                 'CutOff', 'COValue', 'Min', 'Nmin', 'Recur.', 'RcDelta', 'Time', 'FixSeed', 'SBin', 'Mu',
'Sigma',
                 'Max', 'Sol','COTemp'])
            self.csvADD([self.FuncOption, self.GAUSSIANsampling, self.NUMITER, self.NUMpoints,
self.POINTSperTEMPERAT, self.T0, self.TSCHEDULE,
                         self.OBJfunctionRANGE, self.DELTAObjFunc, self.STATEspaceNARROWING,
self.STATEspaceNARvalue,
                         self.DELTASpace, self.APPLYcutoff, self.CUTOFFvalue, Bfbest, Nmim,
self.RECURSIVEstart,
                         self.STARTdelta, BFTime, self.FIXseed, SmallestBin, mu, sigma, MaxValue, [Bxbest[0],
Bxbest[1]],self.COt])

        if self.WRITEtoFILE:
            self.csvADD(
                ['Process Time:', BFTime, 'Nmim:', Nmim, 'SBin:', SmallestBin, 'Mu:', mu, 'Sigma:', sigma,
'Max:', MaxValue, 'Min:', MinValue])

        if self.GRAPH or self.GRAPHtemperSCHED or self.GRAPHsurface or self.GRAPHhistogram:
            plt.show()
        return [Bxbest, Bfbest]


    def f(self, xVect):     # define objective functions
        x = xVect[self.DimX]
        y = xVect[self.DimY]
        Funct = {'A': 0.7 + x**2 + y**2 - 0.1*math.cos(6.0*3.1415*x) - 0.1*math.cos(6.0*3.1415*y),
                 'B': (math.e**math.sin(50*x) + math.sin(60*math.e**y) + math.sin(70*math.sin(x)) +
math.sin(math.sin(80*y)) - math.sin(10*(x + y)) + 0.25*(x**2 + y**2)),
                 'C': x**2+ y**2 + 5,
                 'D': 10000*(math.e**math.sin(50*x) + math.sin(60*math.e**y) + math.sin(70*math.sin(x)) +
math.sin(math.sin(80*y)) - math.sin(10*(x + y)) + 0.25*(x**2 + y**2)),
                 'E': (2*x**6-12.2*x**5+21.2*x**4+6.2*x-6.4*x**3-4.7*x**2+y**6-11*y**5+43.3*y**4-10*y-
74.8*y**3+56.9*y**2-4.1*x*y-0.1*y**2*x**2+0.4*x*y**2+0.4*x**2*y),
                 'F': (1-x)**2 + 25*(y-x**2)**2 + (math.cos(3*math.pi*x+4*math.pi*y)+1),
                 'Z': (self.CustomFuction(xVect,**self.kwargs)),
                 'G': (20*math.sin(x*np.pi/2-2*np.pi) + 20*math.sin(y*np.pi/2-2*np.pi)+(x-2*np.pi)**2+(y-
2*np.pi)**2)}
        return Funct[self.FuncOption]



    def csvADD (self, line):
        import sys
        if sys.version_info > (3,4):
            with open('SimAnneal.csv', 'w', newline='') as file:
                writer = csv.writer(file)
                writer.writerows(line)
        else:
            with open(r'SimAnneal.csv', 'ab') as file:
                writer = csv.writer(file)
                writer.writerow(line)

    def simulated_annealing(self, schedule, StartCoord):
        random.seed()
        xbest = np.array(StartCoord)
        xmin = np.zeros(len(StartCoord),dtype=float)
        xmax = np.zeros(len(StartCoord),dtype=float)
        xn = np.array(StartCoord)
        Gxbest = np.array(xbest)
        fbest = self.f(xbest)
        Gfbest = fbest  # In certain situations we guess a global min. but the temperature is too high and we
don't accept it. Gfbest captures this situation.
        t=1; CutOff = 0; StateSpaceCutOff = 0
        start = time.time()
        while True:
            T = schedule(t)
            for SameTemperature in range(0, self.POINTSperTEMPERAT):
                if T == 0 or (self.APPLYcutoff==True and CutOff == (self.CUTOFFvalue+1)):
                    elapsed = time.time() - start
                    if Gfbest < fbest: xbest, fbest = Gxbest, Gfbest
                    if self.PRINTSOLUTION:
                        # print("T Schedule:", TSCHEDULE)
                        print("Time elapsed: ", elapsed)
                        print('Best solution: ' + str(xbest))
                        print('Best objective: ' + str(fbest))
                        print()
                    if self.WRITEtoFILE:
                        Result = [self.ItNumber + 1, elapsed, t - 1, xbest, fbest, self.T0]
                        self.csvADD(Result)
                    self.COt = t
                    if self.APPLYcutoff: print("Temperature/time cycles (t):", t)
                    return (xbest,fbest)
                self.Xtime.append(t)
                if self.GRAPHtemperSCHED:
                    self.Ytemperature.append(T)
                    self.YtempLimit.append(self.T0/math.log(t+0.1))

                if self.FIXseed: random.seed(t)
                if self.STATEspaceNARROWING and StateSpaceCutOff > self.STATEspaceNARvalue:
                    for nds in range(int(len(StartCoord))):
                        xmin[nds] = -1*(self.DELTASpace * abs(xbest[nds]))
                        xmax[nds] = (self.DELTASpace * abs(xbest[nds]))
                        xn[nds] = (xbest[nds] + random.uniform(xmin[nds], xmax[nds]))
                        if xn[nds] < (-1 * self.OBJfunctionRANGE): xn[nds] = (-1 * self.OBJfunctionRANGE)
                        if xn[nds] > self.OBJfunctionRANGE: xn[nds] = self.OBJfunctionRANGE
                        #xn[nds] = random.uniform(xmin[nds], xmax[nds])
                    if self.PRINTiter: print("Narrowing")
                    if StateSpaceCutOff/self.STATEspaceNARvalue > 2.0:  #If stuck on this local minimum
(fbest), explore Gfbest
                        #StateSpaceCutOff = 0  ####
                        if self.PRINTiter: print("Testing against lowest value found...")
                        if Gfbest < fbest:
                            xbest, fbest = np.array(Gxbest), Gfbest
                            if self.PRINTiter: print ("Exploring around lowest value found...")
                else:
                    for nd in range(int(len(StartCoord))):
                        if self.GAUSSIANsampling:
                            xn[nd] = (random.gauss(xbest[nd],math.sqrt(T)))
                        else:
                            xn[nd] = (xbest[nd] + random.uniform(-self.DELTAObjFunc * self.OBJfunctionRANGE,
self.DELTAObjFunc * self.OBJfunctionRANGE))
```

```
                    #xn[nd] = (random.gauss(xbest[nd],math.sqrt(T)))
                    if xn[nd] < -self.OBJfunctionRANGE: xn[nd] = -self.OBJfunctionRANGE
                    if xn[nd] > self.OBJfunctionRANGE: xn[nd] = self.OBJfunctionRANGE
                if self.PRINTiter: print("Not narrowing")
            fn = self.f(xn)
            try:
                #Prob = math.exp(-(fn-fbest) / (1.00 * T))     #Metropolis Criterion
                Prob = 1/(1+math.exp((fn - fbest) / T))         #Barker Criterion
            except OverflowError:
                Prob = 0
            if self.GRAPHProb:
                self.YProbGraph.append(Prob)
            if self.PRINTiter:
                # Monitor the temperature & cost
                print("t:", "%.0f" % round(t, 0), "Temp:", "%.10fC" % round(T, 10),
                      #"xbest:", np.around(xbest, 5),
                      "xn:", np.around(xn, 5),
                      #"nd:", np.around(nd, 5),
                      "zn:", "%.5f" % round(fn, 5),
                      "zbest:", "%.5f" % round(fbest, 5),
                      "P:", "%.5f" % round(Prob, 5),
                      #"DeltaSpace:", "%.4f" % round(DELTASpace, 4),
                      "StateCutOff:", "%.0f" % StateSpaceCutOff,
                      "CutOff:", "%.0f" % CutOff)
            if self.GRAPH or self.GRAPHcontour:
                self.Xs.append(np.array(xn[self.DimX]))
                self.Ys.append(np.array(xn[self.DimY]))
                self.Zs.append(fbest)


            if fn < Gfbest: Gxbest, Gfbest = np.array(xn), fn #if fn < Gfbest always accept smallest cost
            if fn < fbest or (random.uniform(0.0, 1.0) < Prob):
                xbest, fbest = np.array(xn), fn
                CutOff=0
                StateSpaceCutOff = 0  ####
                # if self.GRAPH:  #Tirar comentario se quiser plotar so os pontos aceitos
                #     self.Xs.append(xbest[self.DimX])
                #     self.Ys.append(xbest[self.DimY])
                #     self.Zs.append(fbest)
            else:
                StateSpaceCutOff += 1
            if self.GRAPHobjValue:
                self.YbestOBJ.append(fbest)
                self.YOBJ.append(fn)
        CutOff += 1
        t += 1
```

File CallAnnealV2_2.py

```
import RodsAnnealMultiV2_2 as saM

def TestFunction(xVect,**kwargs): #This function was used to test a function that called it from within the
SA Class.
    x = xVect[0]
    y = xVect[1]
    if kwargs["FuncName"] == "A":
        Cost = x**3+y**3
    if kwargs["FuncName"] == "B":
        Cost = x**3+y
    return Cost

SimAnneal = saM.SA("B",None,1000)
a=SimAnneal.Anneal([9.0,9.0])
```

File NeuralNetwMNISTV2_2.py

```
from __future__ import print_function
import numpy as np
from scipy import optimize
from scipy.optimize import minimize
import matplotlib.pyplot as plt
import pylab as pl
import math
import csv
import time
from matplotlib import cm
import random
from scipy.io import loadmat
from math import sqrt
import pickle
import RodsAnnealMultiV2_2 as sam

class Neural_Network(object):
    def __init__(self, Lambda=0.000032):  #Controls the trade-off beteween variance and bias
        #high Lambda = little variation, high bias; Low Lambda = high variance, low bias
    #def __init__(self, Lambda=0.0000):
        # Define Hyperparameters
        self.inputLayerSize = 784
        self.outputLayerSize = 10
        self.hiddenLayerSize = 50
        self.W1 = initializeWeights(n_input, n_hidden)
        self.W2 = initializeWeights(n_hidden, 10)
        # Regularization Parameter:
        self.Lambda = Lambda

    def sigmoid(self, z):
        # Apply sigmoid activation function to scalar, vector, or matrix
        return 1 / (1 + np.exp(-z))

    def forward(self, X):
        # Propogate inputs though network
        self.z2 = np.dot(X, self.W1)  # multiplica a input layer pelos pesos
        self.a2 = self.sigmoid(self.z2)  # aplica a activation function
        self.z3 = np.dot(self.a2, self.W2) # multiplica a segunda camada layer pelos pesos W2
        yHat = self.sigmoid(self.z3)  # aplica a segunda activation function e calcula a previsao
        return yHat

    def costFunction(self, X, y):
        # Compute cost for given X,y, use weights already stored in class.
        self.yHat = self.forward(X)
        J = 0.5 * np.sum((y - self.yHat) ** 2) / X.shape[0] + (self.Lambda / 2) * (np.sum(self.W1 ** 2) +
np.sum(self.W2 ** 2))
        #print(J)
        return float(J)

    def getParams(self):
        # Get W1 and W2 Rolled into vector:  (.ravel flattens the array)
        params = np.concatenate((self.W1.ravel(), self.W2.ravel()))
        return params

    def setParams(self, params):
        # Set W1 and W2 using single parameter vector: (reverse the flattened array oper above)
        W1_start = 0
        W1_end = self.hiddenLayerSize * self.inputLayerSize
        self.W1 = np.reshape(params[W1_start:W1_end], \
                             (self.inputLayerSize, self.hiddenLayerSize))
        W2_end = W1_end + self.hiddenLayerSize * self.outputLayerSize
        self.W2 = np.reshape(params[W1_end:W2_end], \
                             (self.hiddenLayerSize, self.outputLayerSize))
    def setWeights(self):
        self.W1 = initializeWeights(n_input, n_hidden)
        self.W2 = initializeWeights(n_hidden, 10)


def costFunctionWrapper(params,**kwargs):
    n_input, n_hidden, n_class, X, y, lambdaval = kwargs["args"]
    NN.setParams(params)
    cost = NN.costFunction(X, y)
    return cost

def costFunctionWrapper2(Wts,**kwargs):
    NN.setWeights()
    Wts = NN.getParams()
    args = kwargs["args"]
    #global args
    cost, grad = nnObjFunction(Wts, *args)
    return cost

# define objective function
def f(Wts):
    global args
    cost, grad = nnObjFunction(Wts, *args)
    return cost

def csvADD (line):
    with open(r'SimAnneal.csv', 'ab') as file:
        writer = csv.writer(file)
        writer.writerow(list(line))

def PickSched(SchedType = 'Boltzman', limit = 4000):
    global T0
    limit += 1
    if SchedType == 'ExpDecay':
        To = 1
        lam = 0.004
        SchedFunc = lambda t: (To * math.exp(-lam * t) if t < limit else 0)
    elif SchedType == 'Boltzman':
        To = 1.0
        SchedFunc = lambda t: (To/(1+math.log(1+t)) if t < limit else 0)
    elif SchedType == 'ExpMultiplicative':
        To = 2
        SchedFunc = lambda t: (To * 0.993**t) if t < limit else 0
    elif SchedType == 'LogMultip':
        To = 35.0
        Alpha = 1.1 # Alpha>1
        SchedFunc = lambda t: (To/(1+Alpha*math.log(1+t)) if t < limit else 0)
    elif SchedType == 'LinearMultip':
        To = 35.0
        Alpha = 2  # Alpha>0
        SchedFunc = lambda t: (To/(1+Alpha*t) if t < limit else 0)
    elif SchedType == 'QuadraticMultip':
        To = 35.0
        Alpha = 2  # Alpha>0
        SchedFunc = lambda t: (To/(1+Alpha*t**2) if t < limit else 0)
    elif SchedType == 'LinearInvTime':
         To = 1.0
         SchedFunc = lambda t: (To/t if t < limit else 0)
    global TSCHEDULE
    TSCHEDULE = SchedType
    T0 = To   #T0 is the initial temperature which is recorded in the .csv file
    return SchedFunc

def simulated_annealing(StartCoord, schedule=PickSched(),  DeltaObjFunc = 0.25):
    StartSim = time.time()
    random.seed()
    xbest = StartCoord
    xmin = np.zeros(len(StartCoord))
    xmax = np.zeros(len(StartCoord))
    xn = StartCoord
    Gxbest = xbest
    fbest = f(xbest)
    Gfbest = fbest  # In certain situations we guess a global min. but the temperature is too high and we
don't accept it. Gfbest captures this situation.
    t = 1.0;
    CutOff = 0;
    StateSpaceCutOff = 0
    global Xs, Ys, Zs, Xtime, Ytemperature, YProbGraph, YbestOBJ, YOBJ, DELTASpace, start
    while True:
        T = schedule(t)
        for SameTemperature in range(0, POINTSperTEMPERAT):
            if T == 0 or (APPLYcutoff==True and CutOff == (CUTOFFvalue+1)):
                elapsed = time.time() - StartSim
                if Gfbest < fbest: xbest, fbest = Gxbest, Gfbest
                if PRINTSOLUTION:
                    #print("T Schedule:", TSCHEDULE)
                    print("Time elapsed: ", elapsed)
                    print ('Best solution: ' + str(xbest))
                    print ('Best objective: ' + str(fbest))
                    print ()
                if WRITEtoFILE:
                    Result = [Iteration + 1, elapsed, t - 1, xbest, fbest, T0]
                    csvADD(Result)
                return (xbest, fbest)
            Xtime.append(t)
            if GRAPHtemperSCHED:
                Ytemperature.append(T)
            if STATEspaceNARROWING and StateSpaceCutOff > STATEspaceNARvalue:
                for nd in range(len(StartCoord)):
                    xmax[nd] = (DELTASpace * abs(xbest[nd]))
                    xmin[nd] = -xmax[nd]  # Apply STATEspace Narrowing
                    xn[nd] = xbest[nd]+ random.uniform(xmin[nd], xmax[nd])
                    if xn[nd] < -OBJfunctionRANGE: xn[nd] = -OBJfunctionRANGE
                    if xn[nd] > OBJfunctionRANGE: xn[nd] = OBJfunctionRANGE
                print("Narrowing...")
                if StateSpaceCutOff / STATEspaceNARvalue > 2.0:  # If stuck on this local minimum (fbest),
explore Gfbest
                    if Gfbest < fbest:
                        xbest, fbest = Gxbest, Gfbest
                        print("Exploring around lowest value found...")
            else:
                NN.setWeights()
                xn = NN.getParams()

            fn = f(xn)
            try:
                #Prob = math.exp(-(fn-fbest)/ T)
                Prob = 1/(1+math.exp((fn - fbest) / T))
            except OverflowError:
                Prob = 0
            if GRAPHProb:
                YProbGraph.append(Prob)
            if PRINTiter:
                # Monitor the temperature & cost
                print("t:", "%.0f" % round(t, 0), "Temp:", "%.10fC" % round(T, 10),
                      "x:", np.around(xn, 5),
                      #"nd:", np.around(nd, 5),
                      "z:", "%.5f" % round(fn, 5),
                      "zbest:", "%.5f" % round(fbest, 5),
                      "P:", "%.5f" % round(Prob, 5),
                      #"DeltaSpace:", "%.4f" % round(DELTASpace, 4),
                      "StateCutOff:", "%.0f" % StateSpaceCutOff,
                      "CutOff:", "%.0f" % CutOff)
            CutOff += 1
            if fn < Gfbest: Gxbest, Gfbest = xn, fn #if fn < Gfbest always accept smallest cost
            if fn < fbest or (random.uniform(0.0, 1.0) < Prob):
                xbest, fbest = xn, fn
                CutOff=0
                StateSpaceCutOff = 0
                if GRAPH:
                    Xs.append(xbest)
                    Zs.append(fbest)
            else:
                StateSpaceCutOff += 1
            if GRAPHobjValue:
                YbestOBJ.append(fbest)
                YOBJ.append(fn)
        t += 1


Xs=[]; Ys=[]; Zs=[]; Xtime=[]; Ytemperature=[]; YProbGraph=[]; YbestOBJ=[]; YOBJ=[]; ItValue = []
#System Parameters
OBJfunctionRANGE = 6.0  #Explores Objetive function from -OBJfunctionRANGE to +OBJfunctionRANGE in x and y
directions
DELTAObjFunc = 0.10 #Used to calculate the next guess for x and y:  xn = xbest + random.uniform(-
DeltaObjFunc*OBJfunctionRANGE, DeltaObjFunc*OBJfunctionRANGE)
WRITEtoFILE = False  #Exports the result of each Anneal to .csv file
GRAPH = 0
GRAPHtemperSCHED = 1
GRAPHProb = 1
GRAPHsurface = 0
GRAPHobjValue = 1
GRAPHhistogram = 0
PRINTiter = 1
PRINTSOLUTION = 1
PRINTbatchSOLUTION = True
# Usar NUMpoints = 3000 T0 = 50 PointsperTEMPERAT = 1 NUMITER=5 para Neural networks
NUMpoints = 6000  #Number of random points per Annealing simulation
POINTSperTEMPERAT = 1
NUMITER = 2 #Number of times to run the Annealing Function
APPLYcutoff = 0
CUTOFFvalue = 405
STATEspaceNARROWING = 1 #Turns space narrowing on and off
STATEspaceNARvalue = 10 #Narrows the search space if Best Solution has not chenged in STATEspaceNARvalue
trials
DELTASpace = 0.2
TSCHEDULE = 'ExpDecay'
T0 = 50
RECURSIVEstart = True
```

```
STARTdelta = 0
USEANNEAL = True
REDUCEDsample = True


Iteration = 0
def Anneal(XVect):
    ItValue = []
    global STARTdelta
    if WRITEtoFILE:

csvADD(['ITER','t','Points','x','y','Objective','To','Schedule:',TSCHEDULE,'ObjRange:',OBJfunctionRANGE,
                'Narrowing:',STATEspaceNARROWING,'NarrValue:',STATEspaceNARvalue,
                'DeltaSp:', DELTASpace,'CutOff:',APPLYcutoff, 'COvalue:',CUTOFFvalue,'Points/T:',
POINTSperTEMPERAT])
    Bxbest = XVect
    Bfbest = f(Bxbest)
    batchTime = time.time()
    for Iteration in range(NUMITER):
        print("It:", Iteration+1)
        print ("Start Guess:", XVect)
        Ixbest, Ifbest = simulated_annealing(XVect, PickSched(TSCHEDULE, NUMpoints), DELTAObjFunc)
        if GRAPHhistogram: ItValue.append(Ifbest)
        if Ifbest < Bfbest: Bxbest, Bfbest = Ixbest,  Ifbest
        if RECURSIVEstart:
            XVect = Bxbest + np.random.uniform(-STARTdelta, STARTdelta,len(Bxbest))

    if PRINTbatchSOLUTION:
        print("")
        print("Number of Iterations:", NUMITER)
        print("Best Minimum Found:")
        print("T Schedule:", TSCHEDULE)
        print("Batch Time elapsed: ", time.time()-batchTime)
        print('Best Batch solution: ' + str(Bxbest))
        print('Best Batch objective: ' + str(Bfbest))

    if WRITEtoFILE: csvADD(['Process Time:',time.time()-batchTime])

    if GRAPH:
        fig = plt.figure()
        ax1 = fig.add_subplot(111, projection='3d')
        ax1.scatter(Xs,Ys,Zs, c='b', marker='o')
        ax1.legend(['Points Tryed'])

    if GRAPHtemperSCHED:
        fig, ax1 = plt.subplots()
        ax1.plot(Xtime, Ytemperature, 'b.', markersize=2)
        ax1.set_xlabel('Time')
        ax1.set_ylabel('Temperature', color='b')
        for tl in ax1.get_yticklabels():
            tl.set_color('b')
        if GRAPHProb:
            ax2 = ax1.twinx()
            ax2.plot(Xtime, YProbGraph, 'ro',markersize=2)
            ax2.set_ylabel('Prob.', color='r')
            ax2.set_ylim([-0.5, 2])
            for tl in ax2.get_yticklabels():
                tl.set_color('r')
    if GRAPHobjValue:
        fig3, ax3 = plt.subplots()
        ax3.plot(Xtime, YOBJ, 'bx',markersize=2)
        ax3.plot(Xtime, YbestOBJ, 'ro', markersize=3)
        ax3.set_xlabel('Time')
        ax3.set_ylabel('Function Value')

    if GRAPHsurface:
        # Design variables at mesh points
        i1 = np.arange(-OBJfunctionRANGE, OBJfunctionRANGE, 0.1)
        i2 = np.arange(-OBJfunctionRANGE, OBJfunctionRANGE, 0.1)
        x1m, x2m = np.meshgrid(i1, i2)
        fm = np.zeros(x1m.shape)
        #xg = np.zeros(x1m.shape)
        for i in range(x1m.shape[0]):
            for j in range(x1m.shape[1]):
                fm[i][j], gr = f([x1m[i][j],x2m[i][j],1,1,1,1,1,1,0,0])
        fig2 = plt.figure()
        ax = fig2.gca(projection='3d')
        ax.plot_surface(x1m, x2m, fm, cmap=cm.jet)
    if GRAPHhistogram:
        import matplotlib.mlab as mlab
        # the histogram of the data
        fig = plt.figure()
        ax = fig.add_subplot(111)
        numBins = 50
        n, bins, patches = ax.hist(ItValue, numBins, color='blue', alpha=0.8)
        plt.grid(True)
        rects = ax.patches
        # Now make some labels
        for rect in rects:
            height = rect.get_height()
            if height > 0:
                ax.text(rect.get_x() + rect.get_width() / 2, 1.01 * height, '%d' % int(height), ha='center',
                        va='bottom', fontsize=8)

        # add a 'best fit' line
        ItValue = np.array(ItValue)
        mu = np.mean(ItValue)
        sigma = np.std(ItValue)
        MaxValue = np.max(ItValue)
        MinValue = np.min(ItValue)
        bincenters = 0.5 * (bins[1:] + bins[:-1])
        print(bins[0:2])
        NormCurve = mlab.normpdf(bincenters, mu, sigma)
        title_string = 'Histogram of Best Vals: \nm=' + str(round(mu, 3)) + '  s=' + str(
            round(sigma, 3)) + '  Min=' + str(round(MinValue, 6)) + \
                       '  Max=' + str(round(MaxValue, 5))
        # plt.axes([.1, .1, .8, .7])
        plt.figtext(.5, 0.02,
                    'Niter=' + str(NUMITER) + '  NPts=' + str(NUMpoints) + '  Pts/T=' +
str(POINTSperTEMPERAT) +
                    '  To=' + str(int(T0)) + '  Schd: ' + TSCHEDULE + '  FRange: ' + str(
                        OBJfunctionRANGE) + '  DtaObj: ' + str(DELTAObjFunc) + '  RS: ' + str(RECURSIVEstart)
+
                    '\nNarr: ' + str(STATEspaceNARROWING) + '    NarrValue: ' + str(int(STATEspaceNARvalue))
+
                    '    DtaSp: ' + str(DELTASpace) + '    CutOff: ' + str(APPLYcutoff) + '    COValue: ' +
str(
                        CUTOFFvalue) + '    BinMax: ' + str(round(bins[1], 6)), fontsize=10, ha='center')
        # plt.suptitle(title_string, y=1.05, fontsize=10)
        plt.title(title_string, fontsize=13)
        # plt.axis([-4, 0,0,50])
        # plt.grid(True)
        ax2 = ax.twinx()
        l = ax2.plot(bincenters, NormCurve, 'r-', linewidth=2)
        ax.set_ylabel('Frequency', color='b')
        ax2.set_ylabel('Fitted Normal Dist.', color='r')
        print(int(ax.patches[0].get_height()))

    if GRAPH or GRAPHtemperSCHED or GRAPHsurface or GRAPHhistogram:
        plt.show()
    return Bxbest

def initializeWeights(n_in, n_out):
    """
    # initializeWeights return the random weights for Neural Network given the
    # number of node in the input layer and output layer
    # Input:
    # n_in: number of nodes of the input layer
    # n_out: number of nodes of the output layer
    # Output:
    # W: matrix of random initial weights with size (n_out x (n_in + 1))"""
    epsilon = sqrt(6) / sqrt(n_in + n_out + 1);
    W = (np.random.rand(n_out, n_in + 1) * 2 * epsilon) - epsilon;
    return W

def sigmoid(z):
    """# Notice that z can be a scalar, a vector or a matrix
    # return the sigmoid of input z"""
    sigmoid_result = 1.0 / (1.0 + np.exp(-1.0 * z));
    return sigmoid_result

def preprocess():
    """ Input:
     Although this function doesn't have any input, you are required to load
     the MNIST data set from file 'mnist_all.mat'.
     Output:
     train_data: matrix of training set. Each row of train_data contains
       feature vector of a image
     train_label: vector of label corresponding to each image in the training
       set
     test_data: matrix of training set. Each row of test_data contains
       feature vector of a image
```

```
 test_label: vector of label corresponding to each image in the testing
   set
 Some suggestions for preprocessing step:
 - divide the original data set to training, validation and testing set
       with corresponding labels
 - convert original data set from integer to double by using double()
       function
 - normalize the data to [0, 1]
 - feature selection"""
mat = loadmat('mnist_all.mat')
# Dividing the data into training, test and Validation data
data_train = np.empty((0, 784))
trn_lab = np.empty((0, 1))
data_test = np.empty((0, 784))
tes_lab = np.empty((0, 1))
data_val = np.empty((0, 784))
val_lab = np.empty((0, 1))
for i in range(10):
    m1 = mat.get('test' + str(i))
    m2 = mat.get('train' + str(i))
    num1 = m1.shape[0]
    num2 = m2.shape[0]
    num3 = int(0.83342 * num2)
    num4 = num2 - num3
    b = range(m2.shape[0])
    permut_b = np.random.permutation(b)
    Z1 = m2[permut_b[0:num3], :]
    Z2 = m2[permut_b[num3:], :]
    data_train = np.vstack([data_train, Z1])
    data_val = np.vstack([data_val, Z2])
    data_test = np.vstack([data_test, m1])
    for p in range(num3):
        trn_lab = np.append(trn_lab, i)
    for q in range(num4):
        val_lab = np.append(val_lab, i)
    for r in range(num1):
        tes_lab = np.append(tes_lab, i)

# normalizing the data to values between to 0-1.
data_test = data_test / 255
data_train = data_train / 255
data_val = data_val / 255

train_data = data_train
train_label = trn_lab
validation_data = data_val
validation_label = val_lab
test_data = data_test
test_label = tes_lab

print("Train Data Size: ",train_data.shape)
print("Train Label Size: ",train_label.shape)
print("Validation Data Size: ",validation_data.shape)
print("Validation Lable Size: ",validation_label.shape)
print("Test Data Size: ",test_data.shape)
print("Test Lable Size: ",test_label.shape)
print()

if REDUCEDsample:  #Resize training data set to TRAIN_NEWsize points
    TRAIN_NEWsize = 10000
    nsamples = len(train_data)
    x = train_data.reshape((nsamples, -1))
    Y = train_label
    #Create Random indices
    valid_index = random.sample(range(int(len(x))), TRAIN_NEWsize)
    train_data = [x[i] for i in valid_index]
    train_data = np.array(train_data)
    # validation targets
    train_label = [Y[i] for i in valid_index]
    train_label = np.array(train_label)

    VALIDATION_NEWsize = 50
    nsamples = len(validation_data)
    x = validation_data.reshape((nsamples, -1))
    Y = validation_label
    # #Create Random indices
    valid_index = random.sample(range(int(len(x))), VALIDATION_NEWsize)
    validation_data = [x[i] for i in valid_index]
    validation_data = np.array(validation_data)
    # validation targets
    validation_label = [Y[i] for i in valid_index]
```

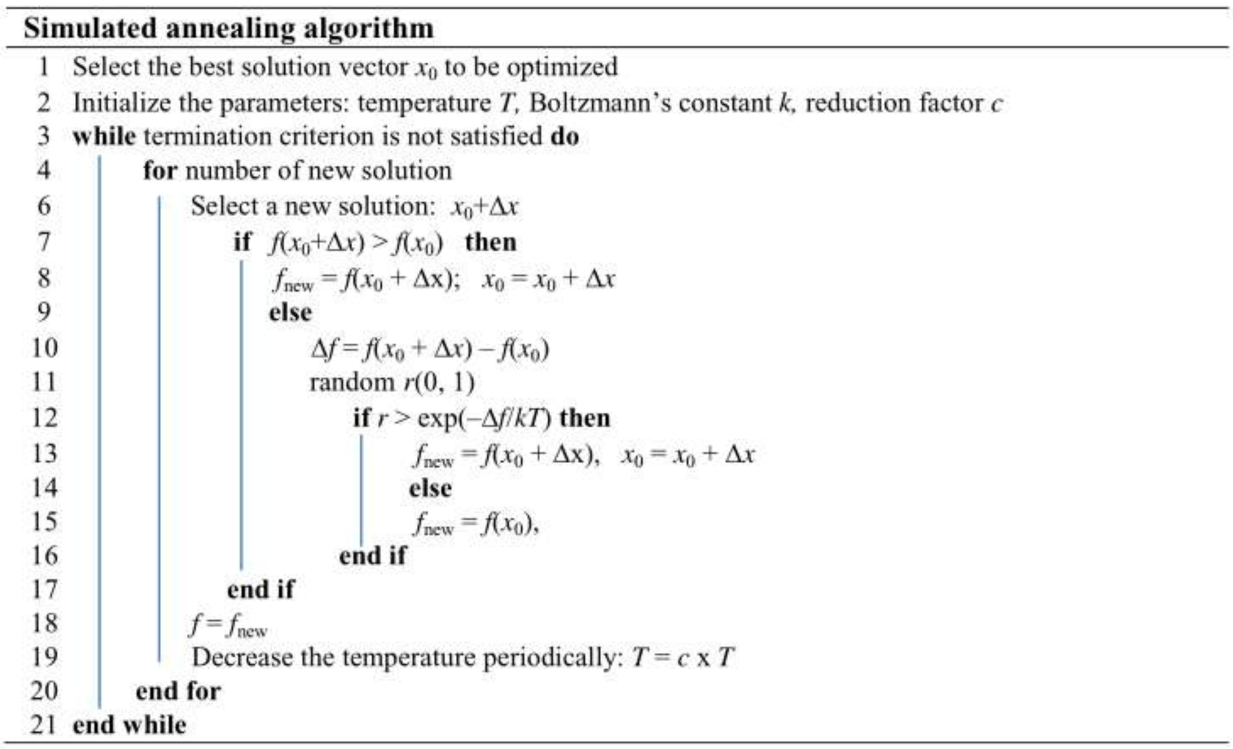
```
        validation_label = np.array(validation_label)

        # Train images and labels
        # train_index = [i for i in range(len(x)) if i not in valid_index]
        # train_data =[x[i] for i in train_index]
        # train_label=[Y[i] for i in sample_index]
        # I do not reduce the test set size so that I have as many samples as I can to test the model

        print("Train Data New Size: ", train_data.shape)
        print("Train Label New Size: ", train_label.shape)
        print("Validation New Data Size: ", validation_data.shape)
        print("Validation New Data Size: ", validation_label.shape)
        print("Test Data New Size: ", test_data.shape)
        print("Test Data New Size: ", test_label.shape)

    return train_data, train_label, validation_data, validation_label, test_data, test_label


def nnObjFunction(params, *args):
    n_input, n_hidden, n_class, training_data, training_label, lambdaval = args

    w1 = params[0:n_hidden * (n_input + 1)].reshape((n_hidden, (n_input + 1)))
    w2 = params[(n_hidden * (n_input + 1)):].reshape((n_class, (n_hidden + 1)))
    obj_val = 0

    training_label = np.array(training_label)
    rows = training_label.shape[0]
    rowsIndex = np.arange(rows, dtype="int")

    tempLabel = np.zeros((rows, 10))
    tempLabel[rowsIndex, training_label.astype(int)] = 1
    training_label = tempLabel

    # nnFeedForwardward propogation
    # adding bias to the input data
    training_data = np.column_stack((training_data, np.ones(training_data.shape[0])))
    number_of_samples = training_data.shape[0]

    # passing the input data to the Hidden layer [calculating a2 = sigmoid(X.W1)]
    zj = sigmoid(np.dot(training_data, w1.T))
    #print(zj.shape)
    # adding bias to the hidden layer
    zj = np.column_stack((zj, np.ones(zj.shape[0])))
    # passing the hidden layer data to the output layer  [calculating Yhat = sigmoid(a2.W2)]

    ol = sigmoid(np.dot(zj, w2.T))

    # Back propogation
    deltaOutput = ol - training_label
    error = np.sum(-1 * (training_label * np.log(ol) + (1 - training_label) * np.log(1 - ol)))
    error = error / number_of_samples
    gradient_of_w2 = np.dot(deltaOutput.T, zj)
    gradient_of_w2 = gradient_of_w2 / number_of_samples

    gradient_of_w1 = np.dot(((1 - zj) * zj * (np.dot(deltaOutput, w2))).T, training_data)
    gradient_of_w1 = gradient_of_w1 / number_of_samples
    gradient_of_w1 = np.delete(gradient_of_w1, n_hidden, 0)
    obj_grad = np.concatenate((gradient_of_w1.flatten(), gradient_of_w2.flatten()), 0)

    error = error + (lambdaval / (2 * number_of_samples)) * (np.sum(np.square(w1)) + np.sum(np.square(w2)))
    obj_val = error
    return (obj_val, obj_grad)


def nnFeedForward(data, w1, w2):
    a = np.dot(data, w1.T)
    z = sigmoid(a)
    z = np.append(z, np.zeros([len(z), 1]), 1)
    b = np.dot(z, w2.T)
    o = sigmoid(b)
    index = np.argmax(o, axis=1)
    label = np.zeros((o.shape[0], 10))
    for i in range(label.shape[0]):
        label[i][index[i]] = 1
    return (z, o)


def nnPredict(w1, w2, data):
    """% nnPredict predicts the label of data given the parameter w1, w2 of Neural
    % Network.
    % Input:
```

```
    % w1: matrix of weights of connections from input layer to hidden layers.
    % w1(i, j) represents the weight of connection from unit i in input
    % layer to unit j in hidden layer.
    % w2: matrix of weights of connections from hidden layer to output layers.
    % w2(i, j) represents the weight of connection from unit i in input
    % layer to unit j in hidden layer.
    % data: matrix of data. Each row of this matrix represents the feature
    % vector of a particular image
    % Output:
    % label: a column vector of predicted labels"""
    data = np.append(data, np.zeros([len(data), 1]), 1)
    n = data.shape[0]
    z, o = nnFeedForward(data, w1, w2);
    label = np.empty((0, 1))
    for i in range(n):
        index = np.argmax(o[i]);
        label = np.append(label, index);
    return label


"""**************Neural Network Script Starts here********************************"""

start_Time = time.time()
train_data, train_label, validation_data, validation_label, test_data, test_label = preprocess();

#  Train Neural Network

# set the number of nodes in input unit (not including bias unit)
n_input = train_data.shape[1]
print("Input Layer size:",n_input)
# set the number of nodes in hidden unit (not including bias unit)
n_hidden = 50

# set the number of nodes in output unit
n_class = 10

# initialize the weights into some random matrices
initial_w1 = initializeWeights(n_input, n_hidden)
initial_w2 = initializeWeights(n_hidden, n_class)

# unroll 2 weight matrices into single column vector
initialWeights = np.concatenate((initial_w1.flatten(), initial_w2.flatten()), 0)
print("Initial Weights before Anneal:")
print("Initial Weights Shape: ",initialWeights.shape)
print("Initial Weights Value: ",initialWeights)

lambdaval = 0.2;
args = (n_input, n_hidden, n_class, train_data, train_label, lambdaval)
if USEANNEAL:
    NN = Neural_Network()
    initialWeights = Anneal(initialWeights)
    # SimAnneal = sam.SA("Z",costFunctionWrapper2,2,6000,args=args)
    # initialWeights, SolValue = SimAnneal.Anneal(initialWeights)
    print("Anneal Optimized Weights: ", initialWeights)
    #print("Anneal Solution: ", SolValue)

# TRAIN NEURAL NETWORK using fmin_cg or minimize from scipy,optimize module. Check documentation for a
working example
opts = {'maxiter': 50}  # Preferred value.  method CG Training set Accuracy:94.784%
nn_params = minimize(nnObjFunction, initialWeights, jac=True, args=args, method='CG', options=opts)

print("Value after Optimization Gradient:", nn_params.fun)
print("Solution:", nn_params.x)
print("Max:", max(nn_params.x))
print("Min:", min(nn_params.x))
print

# Reshape nnParams from 1D vector into w1 and w2 matrices
w1 = nn_params.x[0:n_hidden * (n_input + 1)].reshape((n_hidden, (n_input + 1)))
w2 = nn_params.x[(n_hidden * (n_input + 1)):].reshape((n_class, (n_hidden + 1)))
# Test the computed parameters

# find the accuracy on Training Dataset
predicted_label = nnPredict(w1, w2, train_data)
print('Training set Accuracy:' + str(100 * np.mean((predicted_label == train_label).astype(float))) + '%')

# # find the accuracy on Validation Dataset
predicted_label = nnPredict(w1, w2, validation_data)
print('Validation set Accuracy:' + str(100 * np.mean((predicted_label == validation_label).astype(float))) +
'%')
```

```
# find the accuracy on the Test Dataset
predicted_label = nnPredict(w1, w2, test_data)
print('Test set Accuracy:' + str(100 * np.mean((predicted_label == test_label).astype(float))) + '%')

print()
total_Time = time.time() - start_Time
print("Size test_label: ", len(test_label))
print("Size predicted_label: ", len(predicted_label))
print()
print("Runing Time:", total_Time)
print()
while True:
    ImageIndex = input('Enter a Test image index from 0 to 9999: ')
    if ImageIndex == 'x': break
    ImageIndex = int(ImageIndex)
    print("Test Label: ", int(test_label[ImageIndex]))
    #predicted_label = nnPredict(w1, w2, test_data[ImageIndex:ImageIndex+1])
    print("Pred Label: ", int(predicted_label[ImageIndex]))
    pl.matshow(test_data[ImageIndex].reshape(28, 28), cmap=plt.cm.gray)
    pl.show()
```

APPENDIX C – GENERIC SIMULATED ANNEALING ALGORITHM

Generic Simulated Annealing algorithm taken from [20]:



Simulated annealing algorithm

```
Select the best solution vector $x_0$ to be optimized
Initialize the parameters: temperature $T$, Boltzmann's constant $k$, reduction factor $c$
while termination criterion is not satisfied do
    for number of new solution
        Select a new solution: $x_0+\Delta x$
        if $f(x_0+\Delta x) > f(x_0)$ then
            $f_{new} = f(x_0 + \Delta x)$;   $x_0 = x_0 + \Delta x$
        else
            $\Delta f = f(x_0 + \Delta x) - f(x_0)$
            random $r(0, 1)$
            if $r > \exp(-\Delta f/kT)$ then
                $f_{new} = f(x_0 + \Delta x)$,   $x_0 = x_0 + \Delta x$
            else
                $f_{new} = f(x_0)$,
            end if
        end if
        $f = f_{new}$
        Decrease the temperature periodically: $T = c \times T$
    end for
end while
```