Coursework B: Genetic Algorithms

1 Objectives

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

1. Learn how Genetic Algorithms (GA) optimization works using an example
2. Create a simple Genetic Algorithms (GA) function to optimize a number
3. Create a Genetic Algorithm (GA) to optimize a function

At the end of this laboratory session you should have the basic framework for Genetic Algorithm (GA) optimization and you could explore how to potentially apply this to optimization of neural network parameters.

2 Getting Started

2.1 Introduction

In this example we will take the case of trying to optimize a set of numbers (with size N) such that the sum of the numbers adds up to a target value. For example, we might have N=5 (5 numbers) and optimize them such that the sum of these 5 numbers adds up to 173.

2.2 Example Code

I have provided a basic example (Source: https://lethain.com/genetic-algorithms-cool-name-damn-simple/) which accomplishes this in Python. The source code for this is provided on Moodle.

```python
from random import randint, random
from operator import add
import matplotlib.pyplot as plt

def individual(length, min, max):
    'Create␣a␣member␣of␣the␣population.'
    return [ randint(min,max) for x in xrange(length) ]

def population(count, length, min, max):
    """
␣␣␣␣Create␣a␣number␣of␣individuals␣(i.e.␣a␣population).

␣␣␣␣count:␣the␣number␣of␣individuals␣in␣the␣population
␣␣␣␣length:␣the␣number␣of␣values␣per␣individual
␣␣␣␣min:␣the␣minimum␣possible␣value␣in␣an␣individual's␣list␣of␣values
␣␣␣␣max:␣the␣maximum␣possible␣value␣in␣an␣individual's␣list␣of␣values

␣␣␣␣"""
    return [ individual(length, min, max) for x in xrange(count) ]

def fitness(individual, target):
    """
␣␣␣␣Determine␣the␣fitness␣of␣an␣individual.␣Higher␣is␣better.

␣␣␣␣individual:␣the␣individual␣to␣evaluate
␣␣␣␣target:␣the␣target␣number␣individuals␣are␣aiming␣for
␣␣␣␣"""
    sum = reduce(add, individual, 0)
    return abs(target-sum)

def grade(pop, target):
    'Find␣average␣fitness␣for␣a␣population.'
    summed = reduce(add, (fitness(x, target) for x in pop))
    return summed / (len(pop) * 1.0)

def evolve(pop, target, retain=0.2, random_select=0.05, mutate=0.01):
    graded = [ (fitness(x, target), x) for x in pop]
    graded = [ x[1] for x in sorted(graded)]
    retain_length = int(len(graded)*retain)
    parents = graded[:retain_length]
    # randomly add other individuals to
    # promote genetic diversity
    for individual in graded[retain_length:]:
        if random_select > random():
            parents.append(individual)
    # mutate some individuals
    for individual in parents:
        if mutate > random():
            pos_to_mutate = randint(0, len(individual)-1)
            # this mutation is not ideal, because it
            # restricts the range of possible values,
            # but the function is unaware of the min/max
            # values used to create the individuals,
            individual[pos_to_mutate] = randint(
                min(individual), max(individual))
    # crossover parents to create children
    parents_length = len(parents)
    desired_length = len(pop) - parents_length
    children = []
    while len(children) < desired_length:
        male = randint(0, parents_length-1)
        female = randint(0, parents_length-1)
        if male != female:
            male = parents[male]
            female = parents[female]
            half = len(male) / 2
            child = male[:half] + female[half:]
            children.append(child)
    parents.extend(children)
    return parents


# Example usage
target = 550
p_count = 100
i_length = 6
i_min = 0
i_max = 100
generations = 100
p = population(p_count, i_length, i_min, i_max)
fitness_history = [grade(p, target),]
for i in xrange(generations):
    p = evolve(p, target)
    fitness_history.append(grade(p, target))

for datum in fitness_history:
   print datum

plt.plot(fitness_history)
plt.show()
```

The plot at the end of the simulation will plot the fitness history (i.e. how close the best population matches the target value). The default values for the code provided are as follows:

1. target = 550
2. p_count = 100
3. i_length = 6
4. i_min = 0
5. i_max = 100
6. generations=100

We can look at the sections of code and observe the functions mapping onto the key stages of the GA.

2.3 Stage 1: Create an Initial Population

```
def individual(length, min, max):
    'Create a member of the population.'
    return [ randint(min,max) for x in xrange(length) ]
```

In this function a member of the population *individual* is created, limited between the min and max values. This function is then called *p_count* times to create the population:

```
def population(count, length, min, max):
    """
    Create a number of individuals (i.e. a population).

    count: the number of individuals in the population
    length: the number of values per individual
    min: the minimum possible value in an individual's list of values
    max: the maximum possible value in an individual's list of values

    """
    return [ individual(length, min, max) for x in xrange(count) ]
```

2.4 Stage 2: Iterate over the Generations

In the main loop, the evolve function is called a number of times (the "generations")

```
for i in xrange(generations):
    p = evolve(p, target)
    fitness_history.append(grade(p, target))
```

the evolve function is where the key stages of the GA occur

2.5 Stage 3: Ranking the Population

The Population is first graded:

```
    graded = [ (fitness(x, target), x) for x in pop]
    graded = [ x[1] for x in sorted(graded)]
    retain_length = int(len(graded)*retain)
    parents = graded[:retain_length]
```

and this requires calling the fitness function:

```
def fitness(individual, target):
    """
    Determine the fitness of an individual. Higher is better.

    individual: the individual to evaluate
    target: the target number individuals are aiming for
    """
    sum = reduce(add, individual, 0)
    return abs(target-sum)
```

In this case the fitness is the absolute value of the difference between the target and the current value of the summation. This would need to be modified for the specific application.

2.6 Stage 4: Random Parent Selection and Mutation

Before the crossover can take place we need to randomly select parents - remember you could aply some other techniques here such as elitism if you wish or a combination of techniques including roulette wheel selection.

```
    # randomly add other individuals to
    # promote genetic diversity
    for individual in graded[retain_length:]:
        if random_select > random():
            parents.append(individual)
    # mutate some individuals
    for individual in parents:
        if mutate > random():
            pos_to_mutate = randint(0, len(individual)-1)
            # this mutation is not ideal, because it
            # restricts the range of possible values,
            # but the function is unaware of the min/max
            # values used to create the individuals,
            individual[pos_to_mutate] = randint(
                min(individual), max(individual))
```

A random mutation is applied to the genetic material at this point, but we could equally well apply it to the offspring

2.7 Stage 5: Crossover/Breeding

The crossover is then completed and the resulting population is returned ready for the next iteration:

```
    # crossover parents to create children
    parents_length = len(parents)
    desired_length = len(pop) - parents_length
    children = []
    while len(children) < desired_length:
        male = randint(0, parents_length-1)
        female = randint(0, parents_length-1)
        if male != female:
            male = parents[male]
            female = parents[female]
            half = len(male) / 2
            child = male[:half] + female[half:]
            children.append(child)
    parents.extend(children)
    return parents
```

Note, that we have lots of options with the algorithm, we can allow the population to expand, we can limit to a certain size, we can implement elitism, we can implement mutation in a different stage (i.e. post cross-over) etc.

Exercises:

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

2. How do the population size, mutation and crossover parameters (such as mutation probability and crossover probability) affect how quickly a solution is found?

3. How could you stop the algorithm when a suitable solution had been found to avoid excessive computation time?

4. Create an optimization using GA to optimize a set of parameters for a curve of a 5th-order polynomial as given below:

$$y = 25x^5 + 18x^4 + 31x^3 - 14x^2 + 7x - 19$$

5. Show your understanding on the Holland's Schema Theorem. Explain the Holland's Schema Theorem based on Question 4 using GA with **binary encoding**.

Submission Requirements:

1. You need submit a report in PDF file.
2. An appendix of coding should be given (no page limit).

40% - quality of writing
60% - performance of the results

More Explanations on Coursework B

Actually, the exercises have open solutions. As different students have different programming capabilities, **I don't want to constrain the solutions in a strict format**.

1. For Exercises 1-4, binary encoding is optional. For Exercise 5, you should use binary encoding to explain Holland Schema Theorem with quantitative analysis.

2. Exercise 4:

You can do it based on two ideas:
i) The 5 coefficients and 1 constant term are set as targets, and you use GA to find them. It is similar to Exercise 1.
ii) You give a range for x, for example, -100 to 100, and randomly select some points of the polynomial provided, for example, 100 points as targets. You use GA to find these targets and generate a new line that should match the polynomial provided. It is similar to exercise 4 in Lab of Simulated Annealing.

You may do Exercise 4 using either one of the above two ideas.
You may use both ideas to do a comparison study if you can.

You may also use other ideas of GA to accomplish Exercise 4.
The solutions are open!

3. Report:

There is no page limit for the main report.

The code should be given in Appendix.

In main report, you may show the results and figures, explain the results, and analyse your design.
For example, regarding Exercise 4, at the beginning, what is the initial curve generated by GA? After 100, 200, 500, …. generations, what are the curves generated by GA? In final, how about the curve matching the target polynomial in a figure?

Test and explain the key parameter settings.
For example, if the mutation probability is changed, 1%, 5%, 10%, how about the effect on the final performance?
How about the effect of different crossover probabilities?
How about the results if you use different fitness functions?
How about the results if you use different selection methods, Roulette wheel, ranking, and elitism?
How about the results if you use different termination criteria?
……
……