Genetic Algorithm in Python source code - AI-Junkie tutorial (Python recipe) by David Adler

ActiveState Code (http://code.activestate.com/recipes/578128/)

▲ 2 ▼

A simple genetic algorithm program. I followed this tutorial to make the program http://www.ai-junkie.com/ga/intro/gat1.html.

The objective of the code is to evolve a mathematical expression which calculates a user-defined target integer.

KEY:

chromosome = binary list (this is translated/decoded into a protein in the format number --> operator --> number etc, any genes (chromosome is read in blocks of four) which do not conform to this are ignored.

protein = mathematical expression (this is evaluated from left to right in number + operator blocks of two)

output = output of protein (mathematical expression)

error = inverse of difference between output and target

fitness score = a fraction of sum of of total errors

OTHER:

One-point crossover is used.

I have incorporated **elitism** in my code, which somewhat deviates from the tutorial but made my code more efficient (top ~7% of population are carried through to next generation)

Python, 273 lines

```python
from operator import itemgetter, attrgetter
import random
import sys
import os
import math
import re


# GLOBAL VARIABLES

genetic_code = {
  '0000':'0',
  '0001':'1',
  '0010':'2',
  '0011':'3',
  '0100':'4',
  '0101':'5',
  '0110':'6',
  '0111':'7',
  '1000':'8',
  '1001':'9',
  '1010':'+',
  '1011':'-',
  '1100':'*',
  '1101':'/'
  }

solution_found = False
popN = 100 # n number of chromos per population
genesPerCh = 75
max_iterations = 1000
target = 1111.0
crossover_rate = 0.7
mutation_rate = 0.05

"""Generates random population of chromos"""
def generatePop ():
  chromos, chromo = [], []
  for eachChromo in range(popN):
    chromo = []
    for bit in range(genesPerCh * 4):
      chromo.append(random.randint(0,1))
    chromos.append(chromo)
  return chromos

"""Takes a binary list (chromo) and returns a protein (mathematical expression in string)"""
def translate (chromo):
  protein, chromo_string = '',''
  need_int = True
  a, b = 0, 4 # ie from point a to point b (start to stop point in string)
  for bit in chromo:
    chromo_string += str(bit)
  for gene in range(genesPerCh):
    if chromo_string[a:b] == '1111' or chromo_string[a:b] == '1110':
      continue
    elif chromo_string[a:b] != '1010' and chromo_string[a:b] != '1011' and chromo_string[a:b] != '1100' and chromo_string[a:b] != '1101':
      if need_int == True:
        protein += genetic_code[chromo_string[a:b]]
        need_int = False
        a += 4
        b += 4
        continue
      else:
        a += 4
        b += 4
        continue
    else:
      if need_int == False:
        protein += genetic_code[chromo_string[a:b]]
        need_int = True
        a += 4
        b += 4
        continue
      else:
        a += 4
        b += 4
        continue
  if len(protein) %2 == 0:
    protein = protein[:-1]
  return protein

"""Evaluates the mathematical expressions in number + operator blocks of two"""
def evaluate(protein):
  a = 3
  b = 5
  output = -1
  lenprotein = len(protein) # i imagine this is quicker than calling len everytime?
  if lenprotein == 0:
    output = 0
  if lenprotein == 1:
    output = int(protein)
  if lenprotein >= 3:
    try :
      output = eval(protein[0:3])
    except ZeroDivisionError:
      output = 0
    if lenprotein > 4:
      while b != lenprotein+2:
        try :
          output = eval(str(output)+protein[a:b])
        except ZeroDivisionError:
          output = 0
        a+=2
        b+=2
  return output

"""Calulates fitness as a fraction of the total fitness"""
def calcFitness (errors):
  fitnessScores = []
  totalError = sum(errors)
  i = 0
  # fitness scores are a fraction of the total error
  for error in errors:
    fitnessScores.append (float(errors[i])/float(totalError))
    i += 1
  return fitnessScores

def displayFit (error):
  bestFitDisplay = 100
  dashesN = int(error * bestFitDisplay)
  dashes = ''
  for j in range(bestFitDisplay-dashesN):
    dashes+=' '
  for i in range(dashesN):
    dashes+='+'
  return dashes


"""Takes a population of chromosomes and returns a list of tuples where each chromo is paired to its fitness scores and ranked accroding to its fitness
def rankPop (chromos):
  proteins, outputs, errors = [], [], []
  i = 1
  # translate each chromo into mathematical expression (protein), evaluate the output of the expression,
  # calculate the inverse error of the output
  print '%s: %s\t=%s \t%s %s' %('n'.rjust(5), 'PROTEIN'.rjust(30), 'OUTPUT'.rjust(10), 'INVERSE ERROR'.rjust(17), 'GRAPHICAL INVERSE ERROR'.rjust(105))
  for chromo in chromos:
    protein = translate(chromo)
    proteins.append(protein)

    output = evaluate(protein)
    outputs.append(output)

    try:
      error = 1/math.fabs(target-output)
    except ZeroDivisionError:
      global solution_found
      solution_found = True
      error = 0
      print '\nSOLUTION FOUND'
      print '%s: %s \t=%s %s' %(str(i).rjust(5), protein.rjust(30), str(output).rjust(10), displayFit(1.3).rjust(130))
      break
    else:
      #error = 1/math.fabs(target-output)
      errors.append(error)
    print '%s: %s \t=%s \t%s %s' %(str(i).rjust(5), protein.rjust(30), str(output).rjust(10), str(error).rjust(17), displayFit(error).rjust(105))
    i+=1
  fitnessScores = calcFitness (errors) # calc fitness scores from the erros calculated
  pairedPop = zip ( chromos, proteins, outputs, fitnessScores) # pair each chromo with its protein, ouput and fitness score
  rankedPop = sorted ( pairedPop,key = itemgetter(-1), reverse = True ) # sort the paired pop by ascending fitness score
  return rankedPop

""" taking a ranked population selects two of the fittest members using roulette method"""
def selectFittest (fitnessScores, rankedChromos):
  while 1 == 1: # ensure that the chromosomes selected for breeding are have different indexes in the population
    index1 = roulette (fitnessScores)
    index2 = roulette (fitnessScores)
    if index1 == index2:
      continue
    else:
      break


  ch1 = rankedChromos[index1] # select  and return chromosomes for breeding
  ch2 = rankedChromos[index2]
  return ch1, ch2

"""Fitness scores are fractions, their sum = 1. Fitter chromosomes have a larger fraction.  """
def roulette (fitnessScores):
  index = 0
  cumalativeFitness = 0.0
  r = random.random()

  for i in range(len(fitnessScores)): # for each chromosome's fitness score
    cumalativeFitness += fitnessScores[i] # add each chromosome's fitness score to cumalative fitness

    if cumalativeFitness > r: # in the event of cumalative fitness becoming greater than r, return index of that chromo
      return i


def crossover (ch1, ch2):
  # at a random chiasma
  r = random.randint(0,genesPerCh*4)
  return ch1[:r]+ch2[r:], ch2[:r]+ch1[r:]


def mutate (ch):
  mutatedCh = []
  for i in ch:
    if random.random() < mutation_rate:
      if i == 1:
        mutatedCh.append(0)
      else:
        mutatedCh.append(1)
    else:
      mutatedCh.append(i)
  #assert mutatedCh != ch
  return mutatedCh

"""Using breed and mutate it generates two new chromos from the selected pair"""
def breed (ch1, ch2):

  newCh1, newCh2 = [], []
  if random.random() < crossover_rate: # rate dependent crossover of selected chromosomes
    newCh1, newCh2 = crossover(ch1, ch2)
  else:
    newCh1, newCh2 = ch1, ch2
  newnewCh1 = mutate (newCh1) # mutate crossovered chromos
  newnewCh2 = mutate (newCh2)

  return newnewCh1, newnewCh2

""" Taking a ranked population return a new population by breeding the ranked one"""
def iteratePop (rankedPop):
  fitnessScores = [ item[-1] for item in rankedPop ] # extract fitness scores from ranked population
  rankedChromos = [ item[0] for item in rankedPop ] # extract chromosomes from ranked population

  newpop = []
  newpop.extend(rankedChromos[:popN/15]) # known as elitism, conserve the best solutions to new population

  while len(newpop) != popN:
    ch1, ch2 = [], []
    ch1, ch2 = selectFittest (fitnessScores, rankedChromos) # select two of the fittest chromos

    ch1, ch2 = breed (ch1, ch2) # breed them to create two new chromosomes
    newpop.append(ch1) # and append to new population
    newpop.append(ch2)
  return newpop


def configureSettings ():
  configure = raw_input ('T - Enter Target Number \tD - Default settings: ')
  match1 = re.search( 't',configure, re.IGNORECASE )
  if match1:
    global target
    target = input('Target int: ' )

def main():
  configureSettings ()
  chromos = generatePop() #generate new population of random chromosomes
  iterations = 0

  while iterations != max_iterations and solution_found != True:
    # take the pop of random chromos and rank them based on their fitness score/proximity to target output
    rankedPop = rankPop(chromos)

    print '\nCurrent iterations:', iterations

    if solution_found != True:
      # if solution is not found iterate a new population from previous ranked population
      chromos = []
      chromos = iteratePop(rankedPop)

      iterations += 1
    else:
      break






if __name__ == "__main__":
    main()
```

I am happy to accept any criticism or comments for improvements.

Tags: algorithm, artificial, genetic, network, neural, python

4 comments

David Adler (author) 3 years, 9 months ago

Sorry that it is a bit littered with tests!

Darren Stanney 2 years, 12 months ago

Hi David,

in the selectFitness() function I don't see the need to ensure that the two chromo are different; in fact I feel that this actually hinders the GA from converging + introduces an unneeded bottleneck to the code.

If a self cross is weak the chromo will get weeded out naturally; and if the chromo is strong it will proliferate more quickly than if you didn't allow a self cross.

If you try the following it may improve your results:

```
def selectFittest (fitnessScores, rankedChromos):
  #while 1 == 1: # ensure that the chromosomes selected for breeding are have different indexes in the population
  index1 = roulette (fitnessScores)
  index2 = roulette (fitnessScores)
  #if index1 == index2:
  #continue
  #else:
  #break


  ch1 = rankedChromos[index1] # select  and return chromosomes for breeding
  ch2 = rankedChromos[index2]
  return ch1, ch2
```

Regards Darren.

David Adler (author) 2 years, 11 months ago

hi Darren,

Fair point, I haven't tested which one is more efficient but these are the reasons i did it the way i did:

1. from a biological perspective you can't cross the same gene with itself
2. we use elitism anyway so perhaps less concern about eliminating good ones and more conern about introducing variation
3. I originally introduced it for debugging

deep 2 years, 1 month ago

Hii David

How to implement Genetic Algorithm for classifying Biological database which contain DNA string??