Отчет студента о практике

Введение

Алгоритм дифференциальной эволюции – это стохастический алгоритм (то есть работает с использованием случайных чисел), а так же прямой означающий, что он использует только значения целевой функций, а не её производную (градиент или гессиан). Это делает его устойчивым ко многим поверхностным особенностям, что является в некоторых случаях преимуществом.

Из названия – эволюционный метод. В этих семействах симулируется естественный отбор “лучших” особей. Особенностью алгоритмов дифференциальной эволюции является использование различий между индивидами (значениями аргумента целевой функции), реализованное линейным оператором, называемым «дифференциацией».

Описание метода

Сначала на множестве допустимых решений случайным образом генерируется конечный набор

*I*0 ={*x* ,*k* =1,2,...,*N\*P*}⊂ *D*

векторов, называемый начальной популяцией, где N\*P – размер популяции. Далее идет циклический процесс замены текущей популяции новой.

Для формирования новой популяции последовательно выбирается каждый элемент текущей популяции, и принимается решение, остается ли он в новой популяции или его заменяет сгенерированный специальным образом вектор-образец.

Алгоритм заканчивает работу по выполнению заданного количества итераций.

Алгоритм с описанием

Шаг 0.

Задание функции fobj и границ bounds определения аргументов

Шаг 1.

Задание параметров алгоритма

mut – параметр мутации

crossp – параметр вероятности

popsize – количество векторов популяции

its – количество итераций

Шаг 2.

Инициализация популяции векторов

Запишем размерность bounds в переменную dimensions, создаем массив pop неотрицательных случайных чисел < 1 размерности popsize x dimensions. Каждая строка массива – вектор популяции.

Шаг 3.

Денормализация популяции относительно границ каждого аргумента

Воспользуемся формулой pop\_denorm = min\_b + pop \* diff, где diff – абсолютная разность min\_b и min\_a. min\_a и min\_b – нижняя и верхняя границы аргумента соответственно.

Шаг 4.

Оценка каждого вектора

Подставляя значения каждого вектора популяции в функцию, получаем значение функции во всех векторах.

Шаг 5.

Мутация и рекомбинация

Запускается цикл итераций и внутри него – цикл по векторам. Для каждого вектора популяции проводится процедура: 1) создается *вектор мутации* из трех случайных векторов популяции, исключая выбранный, путем формулы mutant = a + mut \* (b - c), где a, b, c – эти случайные векторы. 2) Вектор мутациии “обрезается” (элементы > 1 становятся равны 1, < 0 – равны 0).   
3) Создается вектор случайных чисел (0;1), который поэлементно сравнивается с параметром crossp. Записываются номера индексов, где элемент этого вектора < crossp. 4) Наконец в выбранном изначально векторе делается замена элементов на элементы вектора мутации в определенных в п.3 индексах. Получаем вектор- образец.

Шаг 6.

Проверка оптимальности

Денормализуется вектор, полученный в п.5, подставляется в функцию fobj, и если он минимизирует функцию лучше, чем изначальный вектор, то происходит замена.

Шаг 7.

Продвижение по циклу по всем векторам популяции

Шаг 8.

Продвижение по циклу по всем итерациям

Шаг 9.

Результатом будет наиболее приближенный в последней итерации вектор

Реализация на Python

import numpy as np

def de(fobj, bounds, mut=0.8, crossp=0.7, popsize=20, its=1000):

dimensions = len(bounds)

pop = np.random.rand(popsize, dimensions)

min\_b, max\_b = np.asarray(bounds).T

diff = np.fabs(min\_b - max\_b)

pop\_denorm = min\_b + pop \* diff

fitness = np.asarray([fobj(\*np.full(dimensions, ind)) for ind in pop\_denorm])

best\_idx = np.argmin(fitness)

best = pop\_denorm[best\_idx]

for i in range(its):

for j in range(popsize):

idxs = [idx for idx in range(popsize) if idx != j]

a, b, c = pop[np.random.choice(idxs, 3, replace = False)]

mutant = np.clip(a + mut \* (b - c), 0, 1)

cross\_points = np.random.rand(dimensions) < crossp

if not np.any(cross\_points):

cross\_points[np.random.randint(0, dimensions)] = True

trial = np.where(cross\_points, mutant, pop[j])

trial\_denorm = min\_b + trial \* diff

f = fobj(\*np.full(dimensions, trial\_denorm))

if f < fitness[j]:

fitness[j] = f

pop[j] = trial

if f < fitness[best\_idx]:

best\_idx = j

best = trial\_denorm

yield best, fitness[best\_idx]

func = lambda x, y: x\*\*4 + y\*\*2 + 1

res = list(de(func, bounds=[(-100, 100)] \* 2, its = 500))

print(res[-1])

>> (array([-7.88433434e-05, -2.31722197e-10]), 1.0)

*Искомый вектор [x, y], значение в котором равно W.*

Пример работы программы

1. Зададим нетривиальную функцию

Получаем:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № | 1 | 2 | ... | 60 | ... | 500 |
| x | -9.64820514 | -9.64820514 | ... | -2.9035576 | … | -2.9035340 |
| y | 1.49539298 | 1.49539298 | … | -2.9035222 | … | -2.90353404 |
| f | 3552.200822 | 3552.200822 |  | -78.332331 |  | -78.33233140 |

То есть уже на 60-ой итерации точность уже достигает порядка 10^-7



Двумерный график функции:

Темные области “глубже”, зеленая точка – глобальный минимум

Вариации алгоритма

1. По вариантам мутации

Индекс r – случайный, b – наиболее приближенный

**Случайный с тремя векторами** : Xmut = Xr1 + mut\*(Xr2 − Xr3)

**Случайный с пятью векторами**: Xmut=Xr1 + mut\*(Xr2 − Xr3 + Xr4 − Xr5)

**Лучший с пятью векторами**: Xmut=Xb + mut\*(Xr2 − Xr3 + Xr4 − Xr5)

**Смешаный**: Xmut=Xr1 + mut1\*(Xr2 − Xr3) + mut2\*(Xb − Xr1)

…

1. По вариантам перехода

Биномиальный (вектор-образец получается при сравнении случайного вектора с параметром crossp)

Экспоненциальный (выбираются два случайных индекса целевого вектора и все значения между заполняются элементами вектора мутации)