**A. Изучить эволюционные алгоритмы оптимизации.**

**1. Генетический алгоритм**

(<https://pypi.org/project/geneticalgorithm/>).

\*\* Генетический алгоритм (Genetic Algorithm - GA)\*\* - это метод поиска и оптимизации, основанный на принципах естественной эволюции, включающий процессы отбора, кроссовера (скрещивания) и мутации (изменения).

\*\* Преимущества GA:

1. GA способен искать в больших и сложных пространствах.

2. GA не требует информации о производной или непрерывности функции, поэтому его можно применять к различным задачам оптимизации.

3. GA способен находить несколько хороших решений одновременно, что помогает избежать застревания в локальных оптимумах.

\*\* Недостатки GA:

1. GA может занять много времени для поиска наилучшего решения, особенно в задачах с большим пространством поиска.

2. GA не может гарантировать нахождение глобального оптимума.

3. Выбор параметров GA (таких как вероятность кроссовера, вероятность мутации, размер популяции) может влиять на эффективность алгоритма.

from geneticalgorithm *import* geneticalgorithm as ga

*import* numpy as np

def **f**(*X*):

*return* np.sum(*X*\*\*2)

varbound = np.array([[-100, 100]]\*2)  *# Giới hạn của biến*

algorithm\_param = {'max\_num\_iteration': 1000,

                   'population\_size':100,

                   'mutation\_probability':0.1,

                   'elit\_ratio': 0.01,

                   'crossover\_probability': 0.5,

                   'parents\_portion': 0.3,

                   'crossover\_type':'uniform',

                   'max\_iteration\_without\_improv': None}

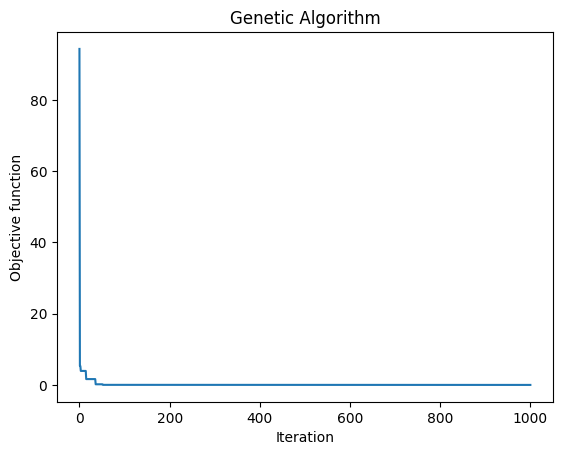
model = ga(*function*=f, *dimension*=2, *variable\_type*='real', *variable\_boundaries*=varbound, *algorithm\_parameters*=algorithm\_param)

model.run()

The best solution found:

[ 0.00413617 -0.00091753]

Objective function:

1.7949764084734117e-05

**2. Алгоритм роя частиц**

(https://pypi.org/project/pyswarm/).

Алгоритм роя частиц (Particle Swarm Optimization - PSO) - это метод глобальной оптимизации на основе популяции, предложенный Кеннеди и Эберхартом в 1995 году. PSO моделирует поведение поиска пищи стаи птиц или рыб.

\*\*Преимущества PSO:

1. PSO прост, понятен и легко реализуется.

2. У PSO мало параметров для настройки.

3. PSO способен искать в больших и сложных пространствах.

\*\*Недостатки PSO:

1. PSO может застрять в локальных оптимумах.

2. PSO может быть неэффективен для сложных задач оптимизации с множеством экстремумов.

3. PSO может занять много времени для поиска наилучшего решения, особенно в задачах с большим пространством поиска.

from pyswarm *import* pso

*import* numpy as np

def **rosenbrock**(*x*):

*return* np.sum(100.0\*(*x*[1:]-*x*[:-1]\*\*2.0)\*\*2.0 + (1-*x*[:-1])\*\*2.0, *axis*=0)

lb = [-3, -3]

ub = [3, 3]

xopt, fopt = pso(rosenbrock, lb, ub)

print("Оптимальное значение x: ", xopt)

print("Оптимальное значение функции: ", fopt)

Stopping search: Swarm best objective change less than 1e-08

Оптимальное значение x: [1.00004167 1.00009592]

Оптимальное значение функции: 1.7570323164860688e-08

**3. Алгоритм муравья**

(<https://pypi.org/project/PyACO/>).

Алгоритм муравья (Ant Colony Optimization - ACO) - это метод глобальной оптимизации, предложенный Марко Дориго в 1992 году. ACO моделирует поведение поиска оптимального пути колонии муравьев.

\*\*Преимущества ACO:

1. ACO очень эффективен в решении задач поиска оптимального пути, таких как задача коммивояжера (Travelling Salesman Problem - TSP).

2. ACO способен искать в больших и сложных пространствах.

3. ACO способен адаптироваться к изменениям в процессе поиска, таким как изменение окружающей среды или изменение цели.

\*\*Недостатки ACO:

1. ACO может занять много времени для поиска наилучшего решения, особенно в задачах с большим пространством поиска.

2. ACO может быть неэффективен для сложных задач оптимизации с множеством экстремумов.

3. Выбор параметров ACO (таких как скорость испарения феромона, коэффициент феромона) может влиять на эффективность алгоритма.

*import* pants

*import* random

*import* math

nodes = []

*for* \_ in range(20):

    x = random.uniform(-10, 10)

    y = random.uniform(-10, 10)

    nodes.append((x, y))

def **euclidean**(*a*, *b*):

*return* math.sqrt(pow(*a*[1] - *b*[1], 2) + pow(*a*[0] - *b*[0], 2))

world = pants.World(nodes, euclidean)

solver = pants.Solver()

*# Giải quyết bài toán*

solution = solver.solve(world)

print("Оптимальный путь: ", solution.tour)

print("Общее расстояние: ", solution.distance)

Оптимальный путь: [(2.056018774978094, 5.9311022541442675), (0.16847223563071267, 5.190939202299088), (-1.7492104350753497, 6.25646893379319), (-2.016711966077674, 7.904318615623154), (-4.613282199568333, 9.193675867540751), (-5.938124646944325, 7.707592638424096), (-6.347317636634527, 8.258806676246486), (-9.944895828422789, 9.299004643989985), (-9.782940230633198, 5.95583466780303), (-5.660889235865509, 2.562382207053073), (-3.5676218309555408, 3.3960190162213717), (-7.694160000298567, -2.901421456999307), (-2.214553546562099, -6.705286160099613), (-0.9918786527914758, -7.823159024559516), (-5.800236611366545, -8.485697747303734), (-6.0070447667476445, -7.326102524344112), (-0.11199524090734769, -1.6838568300392147), (3.7965852239908813, -2.900934195387328), (7.927935715247905, 3.660194975769395), (7.875431881843944, 8.302285153667675)]

Общее расстояние: 78.97298260979755

**4. Пчелиный алгоритм**

(<https://pypi.org/project/bees-algorithm/>).

Пчелиный алгоритм - это метод глобальной оптимизации на основе популяции, предложенный Фамом, Ганбарзаде и соавторами в 2005 году. Пчелиный алгоритм моделирует поведение поиска меда пчелами.

\*\*Преимущества Пчелиного алгоритма:

1. Пчелиный алгоритм очень эффективен в решении задач поиска оптимального пути, таких как задача коммивояжера (Travelling Salesman Problem - TSP).

2. Пчелиный алгоритм способен искать в больших и сложных пространствах.

3. Пчелиный алгоритм способен адаптироваться к изменениям в процессе поиска, таким как изменение окружающей среды или изменение цели.

\*\*Недостатки Пчелиного алгоритма:

1. Пчелиный алгоритм может занять много времени для поиска наилучшего решения, особенно в задачах с большим пространством поиска.

2. Пчелиный алгоритм может быть неэффективен для сложных задач оптимизации с множеством экстремумов.

3. Выбор параметров Пчелиного алгоритма (таких как количество исследовательских пчел, количество следящих пчел) может влиять на эффективность алгоритма.

*import* random

*import* math

def **f**(*x*):

*return* *x*\*\*2 - 3\**x* + 1

def **bee\_algorithm**(*f*, *a*, *b*, *n\_bees*, *n\_iter*):

    bees = [random.uniform(*a*, *b*) *for* \_ in range(*n\_bees*)]

*for* \_ in range(*n\_iter*):

        quality = [*f*(bee) *for* bee in bees]

        best\_bee = bees[quality.index(min(quality))]

        bees = [random.uniform(best\_bee - 1, best\_bee + 1) *for* \_ in range(*n\_bees*)]

*return* best\_bee

x\_opt = bee\_algorithm(f, -10, 10, 100, 1000)

print("Giá trị tối ưu của x: ", x\_opt)

print("Giá trị tối ưu của hàm: ", f(x\_opt))

Оптимальное значение x: 1.4696019711682324

Оптимальное значение функции: -1.2490759598431427

**5. Алгоритм дифференциальной эволюции**

([https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.d ifferential\_evolution.html](https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.d%20ifferential_evolution.html)).

Алгоритм дифференциальной эволюции (Differential Evolution - DE) - это метод глобальной оптимизации на основе популяции, предложенный Сторном и Прайсом в 1997 году. DE использует механизм естественной эволюции для поиска оптимального решения.

\*\*Преимущества DE:

1. DE прост, понятен и легко реализуется.

2. DE эффективен в решении задач непрерывной оптимизации и задач с большим пространством поиска.

3. У DE меньше параметров для настройки, чем у других алгоритмов оптимизации.

\*\*Недостатки DE:

1. DE может занять много времени для поиска наилучшего решения, особенно в задачах с большим пространством поиска.

2. DE может быть неэффективен для дискретных или сложных задач оптимизации с множеством экстремумов.

3. DE может застрять в локальных оптимумах, если его не тщательно настроить.

from scipy.optimize *import* differential\_evolution

*import* numpy as np

def **rosenbrock**(*x*):

*return* np.sum(100.0\*(*x*[1:]-*x*[:-1]\*\*2.0)\*\*2.0 + (1-*x*[:-1])\*\*2.0, *axis*=0)

bounds = [(-3, 3), (-3, 3)]

result = differential\_evolution(rosenbrock, bounds)

print("Оптимальное значение x: ", result.x)

print("Оптимальное значение функции: ", result.fun)

Оптимальное значение x: [1. 1.]

Оптимальное значение функции: 4.979684464207637e-30

import numpy as np

from scipy.optimize import newton, differential\_evolution

from scipy.optimize import minimize

import math

def func(x):

    return (math.sin(3 \* math.pi \* x[0]))\*\*2 + ((x[0]-1)\*\*2) \* (1 + (math.sin(3 \* math.pi \* x[1]))\*\*2) + ((x[1]-1)\*\*2) \* (1 + (math.sin(2 \* math.pi \* x[1]))\*\*2)

newton\_results = []

for \_ in range(100):

    x0 = np.random.uniform(-10, 10, 2)

    result = minimize(func, x0, method='BFGS')

    if result.success:

        newton\_results.append(result.fun)

newton\_mean = np.mean(newton\_results)

newton\_var = np.var(newton\_results)

bounds = [(-10, 10), (-10, 10)]

de\_results = [differential\_evolution(func, bounds).fun for \_ in range(100)]

de\_mean = np.mean(de\_results)

de\_var = np.var(de\_results)

print(f"Newton mean: {newton\_mean}, Newton variance: {newton\_var}")

print(f"Differential Evolution mean: {de\_mean}, Differential Evolution variance: {de\_var}")

Newton mean: 30.65498166717154, Newton variance: 1140.5764895078999

Differential Evolution mean: 7.407256156043771e-29, Differential Evolution variance: 1.256727927116218e-88

На основе результатов мы видим, что алгоритм дифференциальной эволюции (DE) работает намного лучше, чем алгоритм Ньютона (используя метод BFGS) в оптимизации функции Леви N.13.

1. Ньютон (BFGS): Среднее конечное значение целевой функции после 100 запусков составляет около 30,65, с дисперсией около 1140,58. Это показывает, что алгоритм Ньютона не может эффективно найти глобальный минимум функции Леви N.13, возможно, из-за того, что эта функция имеет множество локальных минимумов.

2. Дифференциальная эволюция (DE): Среднее конечное значение целевой функции после 100 запусков практически равно 0 (около 7,41e-29), с очень маленьким разбросом (около 1,26e-88). Это показывает, что DE очень эффективен в поиске глобального минимума функции Леви N.13

import numpy as np

from scipy.optimize import differential\_evolution

import math

import time

def func(x):

    return (math.sin(3 \* math.pi \* x[0]))\*\*2 + ((x[0]-1)\*\*2) \* (1 + (math.sin(3 \* math.pi \* x[1]))\*\*2) + ((x[1]-1)\*\*2) \* (1 + (math.sin(2 \* math.pi \* x[1]))\*\*2)

global\_minima = 0

bounds = [(-10, 10), (-10, 10)]

start\_time = time.time()

result = differential\_evolution(func, bounds)

end\_time = time.time()

if abs(result.fun - global\_minima) < 1e-6:

    print(f"Время, необходимое для достижения глобального минимума: {end\_time - start\_time} секунд")

else:

    print("Алгоритм не достиг глобального минимума за отведенное время")

Время, необходимое для достижения глобального минимума: 0.22446012496948242 секунд

import numpy as np

from scipy.optimize import differential\_evolution

import math

import time

def func(x):

    return (math.sin(3 \* math.pi \* x[0]))\*\*2 + ((x[0]-1)\*\*2) \* (1 + (math.sin(3 \* math.pi \* x[1]))\*\*2) + ((x[1]-1)\*\*2) \* (1 + (math.sin(2 \* math.pi \* x[1]))\*\*2)

max\_gen = 1000

bounds = [(-10, 10), (-10, 10)]

times = []

results = []

for \_ in range(100):

    start\_time = time.time()

    result = differential\_evolution(func, bounds, maxiter=max\_gen)

    end\_time = time.time()

    times.append(end\_time - start\_time)

    results.append(result.fun)

time\_mean = np.mean(times)

time\_var = np.var(times)

result\_mean = np.mean(results)

result\_var = np.var(results)

print(f"Thời gian trung bình/Среднее время: {time\_mean}, Phương sai thời gian/Дисперсия времени: {time\_var}")

print(f"Kết quả trung bình/Средний результат: {result\_mean}, Phương sai kết quả/Дисперсия результата: {result\_var}")

Thời gian trung bình/Среднее время: 0.164423451423645, Phương sai thời gian/Дисперсия времени: 0.00047658129697631464

Kết quả trung bình/Средний результат: 7.407256156043771e-29, Phương sai kết quả/Дисперсия результата: 1.256727927116218e-88

На основе результатов мы видим, что алгоритм дифференциальной эволюции (DE) работает очень быстро и эффективно при оптимизации функции Леви N.13.

1. Время: Среднее время для поиска последнего экстремума (который может быть локальным) целевой функции составляет около 0,164 секунды, с очень маленьким разбросом (около 0,00048).

2. Результат: Среднее конечное значение целевой функции после 100 запусков практически равно 0 (около 7,41e-29), с очень маленьким разбросом (около 1,26e-88).

В целом, DE, похоже, очень подходит для задач оптимизации функций с множеством локальных минимумов, таких как Леви N.13, и он работает очень быстро и стабильно.

import numpy as np

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

def func(x, y):

    return (np.sin(3 \* np.pi \* x))\*\*2 + ((x-1)\*\*2) \* (1 + (np.sin(3 \* np.pi \* y))\*\*2) + ((y-1)\*\*2) \* (1 + (np.sin(2 \* np.pi \* y))\*\*2)

x = np.linspace(-10, 10, 100)

y = np.linspace(-10, 10, 100)

x, y = np.meshgrid(x, y)

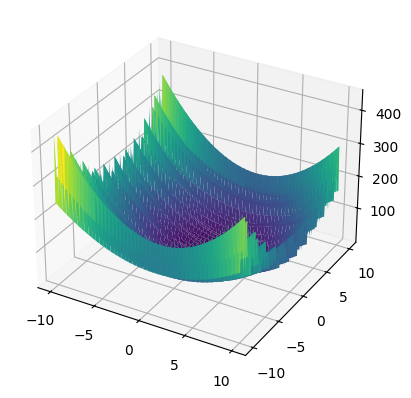
z = func(x, y)

fig = plt.figure()

ax = fig.add\_subplot(111, projection='3d')

ax.plot\_surface(x, y, z, cmap='viridis')

plt.show()



import numpy as np

from scipy.optimize import differential\_evolution

import matplotlib.pyplot as plt

def func(x):

    return (np.sin(3 \* np.pi \* x[0]))\*\*2 + ((x[0]-1)\*\*2) \* (1 + (np.sin(3 \* np.pi \* x[1]))\*\*2) + ((x[1]-1)\*\*2) \* (1 + (np.sin(2 \* np.pi \* x[1]))\*\*2)

bounds = [(-10, 10), (-10, 10)]

values = []

def callback(x, convergence):

    values.append(func(x))

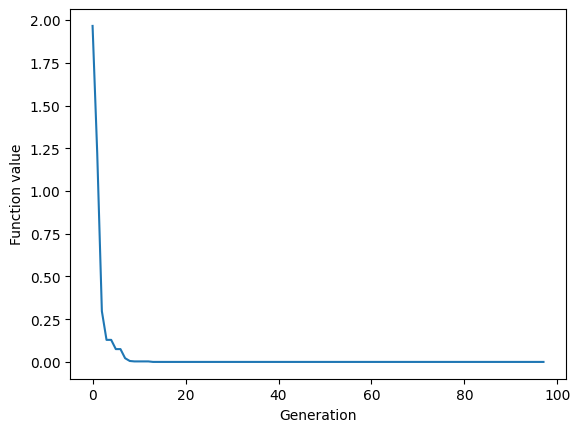
result = differential\_evolution(func, bounds, callback=callback)

plt.plot(values)

plt.xlabel('Generation')

plt.ylabel('Function value')

plt.show()



import pandas as pd

data = {

    'Algorithm': ['Newton', 'Differential Evolution'],

    'Mean Function Value': [newton\_mean, de\_mean],

    'Variance Function Value': [newton\_var, de\_var],

    'Mean Execution Time': [np.nan, time\_mean],

    'Variance Execution Time': [np.nan, time\_var]

}

df = pd.DataFrame(data)

print(df)

Algorithm Mean Function Value Variance Function Value \

0 Newton 3.065498e+01 1.140576e+03

1 Differential Evolution 7.407256e-29 1.256728e-88

Mean Execution Time Variance Execution Time

0 NaN NaN

1 0.164423 0.000477

Основываясь на результатах таблицы, можно сделать следующие выводы:

1. Ньютон: Среднее значение целевой функции после 100 запусков составляет около 30,65, с дисперсией около 1140,58. Это показывает, что алгоритм Ньютона не может эффективно найти глобальный минимум функции Леви N.13. Время выполнения для алгоритма Ньютона в этом случае не было рассчитано.

2. Дифференциальная эволюция (DE): Среднее значение целевой функции после 100 запусков практически равно 0 (около 7,41e-29), с очень маленьким разбросом (около 1,26e-88). Это показывает, что DE очень эффективен в поиске глобального минимума функции Леви N.13. Среднее время выполнения составляет около 0,164 секунды, с дисперсией около 0,000477.

В целом, DE, похоже, очень подходит для задач оптимизации функций с множеством локальных минимумов, таких как Леви N.13, и он работает очень быстро и стабильно.

from hyperopt import hp, fmin, tpe, STATUS\_OK, Trials

from sklearn.model\_selection import cross\_val\_score

from sklearn.svm import SVC

from sklearn.neighbors import KNeighborsClassifier

from sklearn.ensemble import RandomForestClassifier

from sklearn.datasets import load\_iris

iris = load\_iris()

X = iris.data

y = iris.target

def objective\_svm(params):

    clf = SVC(\*\*params)

    score = cross\_val\_score(clf, X, y, cv=3).mean()

    return {'loss': -score, 'status': STATUS\_OK}

space\_svm = {

    'C': hp.loguniform('C', -5, 2),

    'gamma': hp.loguniform('gamma', -5, 2),

    'kernel': hp.choice('kernel', ['linear', 'poly', 'rbf'])

}

best\_svm = fmin(fn=objective\_svm, space=space\_svm, algo=tpe.suggest, max\_evals=100)

def objective\_knn(params):

    clf = KNeighborsClassifier(\*\*params)

    score = cross\_val\_score(clf, X, y, cv=3).mean()

    return {'loss': -score, 'status': STATUS\_OK}

space\_knn = {

    'n\_neighbors': hp.choice('n\_neighbors', range(1, 50)),

    'weights': hp.choice('weights', ['uniform', 'distance']),

    'p': hp.choice('p', [1, 2])

}

best\_knn = fmin(fn=objective\_knn, space=space\_knn, algo=tpe.suggest, max\_evals=100)

def objective\_rf(params):

    clf = RandomForestClassifier(\*\*params)

    score = cross\_val\_score(clf, X, y, cv=3).mean()

    return {'loss': -score, 'status': STATUS\_OK}

space\_rf = {

    'n\_estimators': hp.choice('n\_estimators', range(10, 200)),

    'max\_depth': hp.choice('max\_depth', range(1, 20)),

    'criterion': hp.choice('criterion', ['gini', 'entropy'])

}

best\_rf = fmin(fn=objective\_rf, space=space\_rf, algo=tpe.suggest, max\_evals=100)

print("Best parameters for SVM:", best\_svm)

print("Best parameters for KNN:", best\_knn)

print("Best parameters for RF:", best\_rf)

100%|██████████| 100/100 [00:01<00:00, 67.87trial/s, best loss: -0.9933333333333333]

100%|██████████| 100/100 [00:01<00:00, 62.16trial/s, best loss: -0.9866666666666667]

100%|██████████| 100/100 [00:36<00:00, 2.75trial/s, best loss: -0.9666666666666667]

Best parameters for SVM: {'C': 1.7736699764148685, 'gamma': 0.16645213045659762, 'kernel': 0}

Best parameters for KNN: {'n\_neighbors': 5, 'p': 1, 'weights': 1}

Best parameters for RF: {'criterion': 1, 'max\_depth': 6, 'n\_estimators': 160}

\*\*Алгоритм Ньютона:

Для многократных функций алгоритм Ньютона может не работать хорошо. Ньютон - это алгоритм локальной оптимизации, то есть он начинает с начальной точки и ищет ближайший минимум. Таким образом, если целевая функция имеет множество локальных минимумов, алгоритм Ньютона может застрять в локальном минимуме и не сможет найти глобальный минимум.

\*\*Алгоритм Differential Evolution (DE):

Для многократных функций DE обычно работает намного лучше, чем Ньютон. DE - это алгоритм глобальной оптимизации, то есть он ищет лучшее решение во всем пространстве поиска, а не только около начальной точки. Таким образом, DE способен найти глобальный минимум, даже если есть множество локальных минимумов. Однако