Xiangyu Zheng 21331025 CS7CS2 Optimisation for Machine Learning week8 assignment

(a)(i)

```
def globalRandomSearch(x, y, iters, f):
```

First of all, the algorithm has four parameters,
the array x stores the maximum and minimum values of x,
the array y is the maximum and minimum values of y,
iters is the number of iterations,
f is the incoming equation.

```
for k in range(iters):
    cx = random.uniform(x[0], x[1])
    cy = random.uniform(y[0], y[1])
    value = f(cx, cy)
```

In the number of iterations, randomly select points in the range of x and y, and calculate the function value

```
if minValue > value:
    minValue = value
```

record the smallest function value

(a)(ii)

Here are the two functions from my week4:
Function1: 1 * (x1 - 5) ** 4 + 3 * (y1 - 0) ** 2
Function2: Max(x1 - 5, 0) + 3 * abs(y1 - 0)

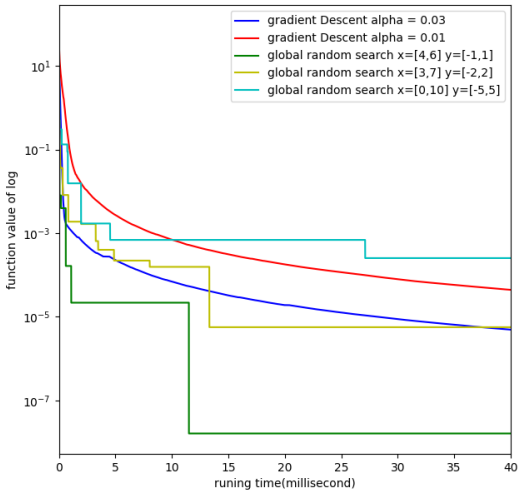

Figure 1 The comparison of global random search and gradient descent with different parameter ranges is in function1

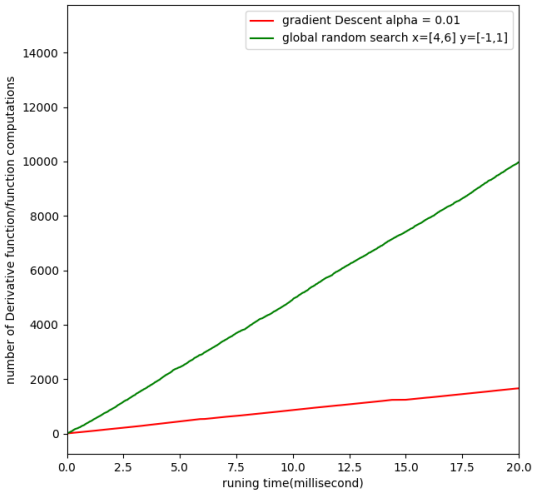

Figure 2 Global random search and gradient descent function calculation comparison in function1

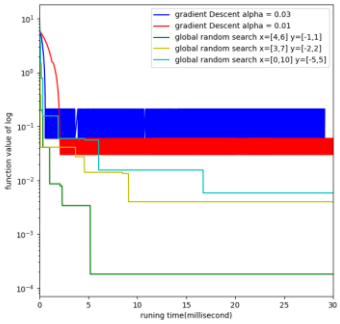

Figure 3 The comparison of global random search and gradient descent with different parameter ranges is in function2

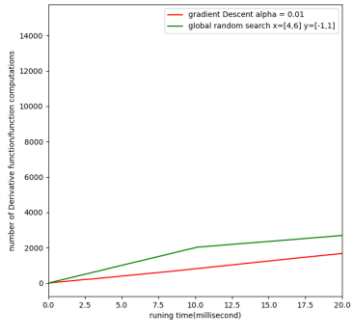

Figure 4 Global random search and gradient descent function calculation comparison in function2

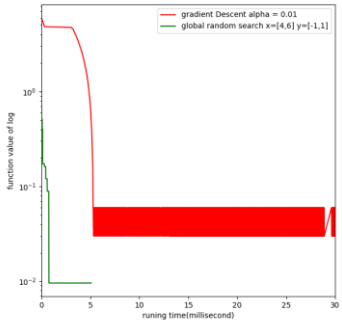

Figure 5 Time required for global random search and gradient descent under 1000 iterations.

We can see from Figure 1 and Figure 3. With the passage of time, the changes of the optimal function values of the two algorithms, global random search and gradient descent, will be inaccurate when measured by time. This is because the content calculated by each iteration of the two algorithms is inconsistent, resulting in the required iterations. time is different.

We can see that the performance of global random search seems to be better than gradient descent, no matter which function. Global random search can always find better function values, and gradient descent slows down to a certain extent in function1, while it keeps oscillating in function2, and global random search does not have this problem. Global random search also performs well in iteration speed. But global random search seems to suffer from the range problem, the closer the range is to the optimal point, the better the algorithm performs.

We can see from Figures 2 and 4 that no matter which function it is, global random search computes the function value more times in the same time. This is because although the global random search only calculates the function value once in one iteration, and the gradient descent calculates the partial derivative value twice, the global random search iteration takes less time, so the number of times the function value is calculated per unit time is more. Figure 5 shows the time required for each of the two algorithms at 1000 iterations

(b)(i)

population based search has a total of 7 parameters

```
def populationBasedSearch(x, y, iters, Nsample, bestP, range, exploitP,):
```

x represents the range of x.

y represents the range of y.

iters indicates the number of times the function needs to be iterated.

Nsample represents how many random samples the algorithm needs to take at the beginning.

bestP indicates how many optimal points are left.

range indicates how many points are randomly selected in the range of the best point.

exploitP indicates how many points to take near the best point.

First create a small top heap for accessing the best function value and xy

```
stp = BtmkHeap(bestP)
```

random sample:

```
while i < Nsample:
    cx = random.uniform(x[0], x[1])
```

```
    cy = random.uniform(y[0], y[1])
    value = f1(cx, cy)
    stp.Push((value, [cx, cy]))
    i = i + 1
```

start iterating:

```
i = 0
while i < iters:
```

Obtain the optimal point, start to randomly obtain points near the optimal point within the specified range, push the function to the small top heap, and automatically calculate the best and latest points:

```
datas = stp.BtmK()
for data in datas:
        while j < exploitP:
                cx = random.uniform(xy[0] - range, xy[0] + range)
                cy = random.uniform(xy[1] - range, xy[1] + range)
                value = f1(cx, cy)
                stp.Push((value, [cx, cy]))
```

(b)(ii)

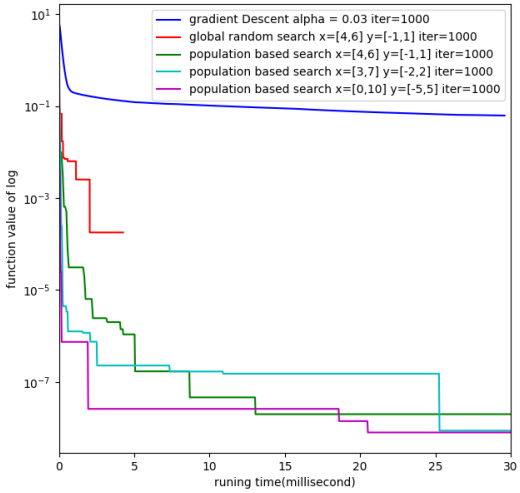

Figure 6 Gradient descent comparison of global random search and population iters = 1000 function1

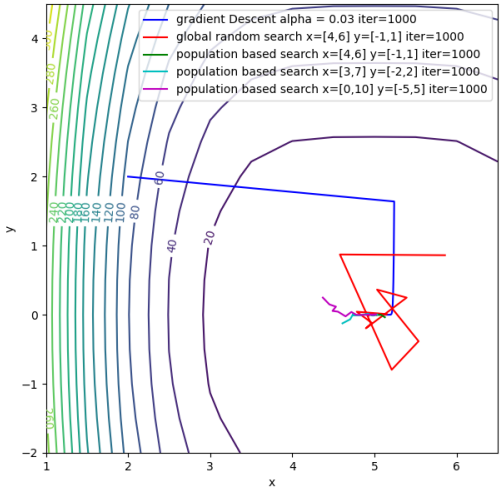

Figure 7 Gradient descent compares global random search and population x and y in function1

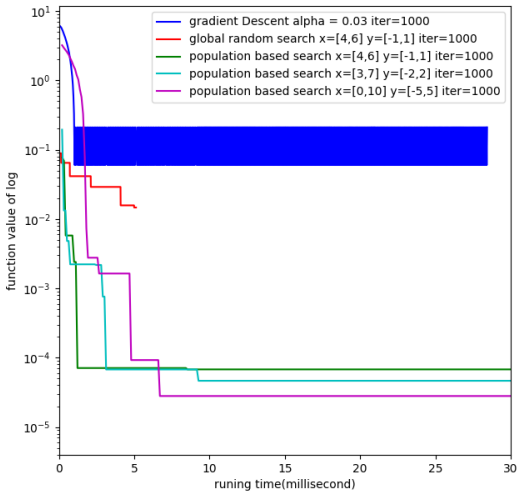

Figure 8 Gradient descent comparison of global random search and population iters = 1000 function2

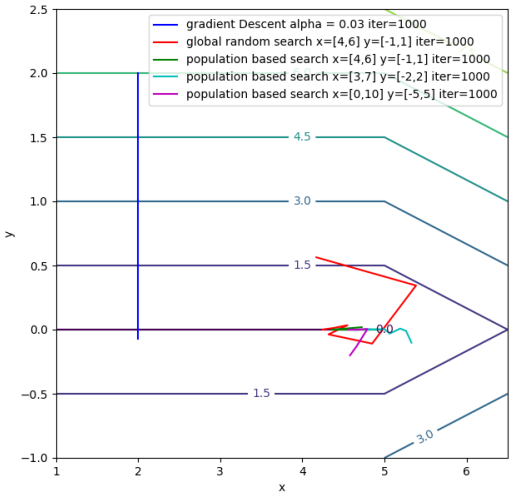

Figure 9 Gradient descent compares global random search and population x and y in function2

It can be seen from Figure 6 and Figure 8 that no matter which function it is, the performance of population based search is the best because it records several optimal points and continues to search for points near the optimal point. But its overhead is relatively high.

As can be seen from Figures 7 and 9, the iterative paths of various algorithms, gradient descent is at a fixed starting point, global random has strong randomness, the starting point is far away, and the fluctuation is also large. Population based search selects the optimal points, the starting point is close, and the iteration is stable.

(c)

Global random search :

Pass the range of alpha, beta1, beta2, batchsize, epoches, into the algorithm

```
def globalRandomSearch(alpha, beta1, beta2, batchsize, epoches, iters):
```

Enter the iteration to randomly select 5 parameters, if the loss value obtained is smaller, record it

```
while i < iters:
    calpha = random.uniform(alpha[0], alpha[1])
    cbeta1 = random.uniform(beta1[0], beta1[1])
    cbeta2 = random.uniform(beta2[0], beta2[1])
    cbatchsize = random.randint(batchsize[0], batchsize[1])
    cepoches = random.randint(epoches[0], epoches[1])

if loss < minloss:
    minloss = loss
    bestPara = (calpha, cbeta1, cbeta2, cbatchsize, cepoches)
```

population based search

Pass parameters

```
def populationBasedSearch(alpha, beta1, beta2, batchsize, epoches, iters,
Nsample, bestP, exploitP):
```

select sample

```
while i < Nsample:
    calpha = random.uniform(alpha[0], alpha[1])
    cbeta1 = random.uniform(beta1[0], beta1[1])
    cbeta2 = random.uniform(beta2[0], beta2[1])
    cbatchsize = random.randint(batchsize[0], batchsize[1])
    cepoches = random.randint(epoches[0], epoches[1])
```

Start iteration, control parameter range

```
while i < iters + Nsample:
    for data in datas:
        paras = data[1]
        j = 0
        while j < exploitP:
            minAlpha = paras[0] - 0.05
            if minAlpha < alpha[0]:
                minAlpha = alpha[0]
```

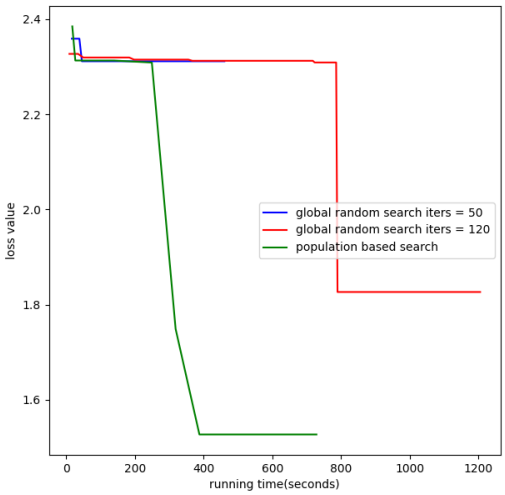

Figure 10 loss function value varies with different algorithms Population based search iters=8, select 10 starting samples, 3 optimal points, 3 exploration points

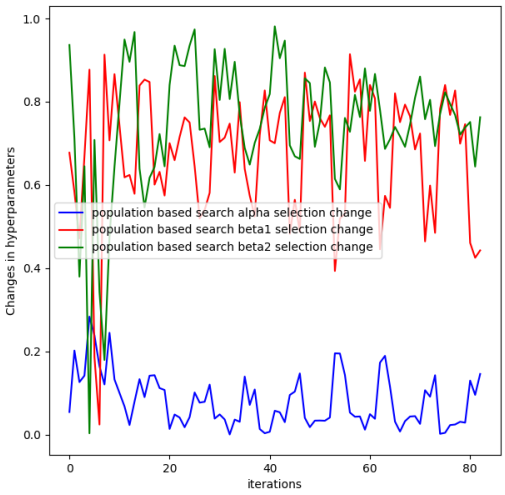

Figure 11 Changes in population based search hyperparameters

We can see from Figure 10. The performance of population based search is significantly better than global random search. It can get a smaller loss value. In this experiment, the range of alpha, beta1, and beta2 is set to 0-1, the batch size is 16 to 128, and the epoches is 10 to 50.

The optimal hyperparameters of global random search for 50 iterations are alpha=0.0073, beta1= 0.40 beta2 = 0.02 batch size = 102, epoches=34
The optimal hyperparameters of global random search for 120 iterations are alpha=0.0078, beta1= 0.50 beta2 = 0.52 batch size =127, epoches=20
The best hyperparameters for population based search are alpha=0.0037, beta1= 0.70 beta2 = 0.5 batch size =84, epoches=18

We can also see from Figure 11 that when selecting hyperparameters, population based search is always within a fixed range, while global random search is random.

Appendix

Code

a-i – b-ii

```
import random
import time

import numpy as np

from minheap import BtmkHeap
import sympy
from matplotlib import pyplot as plt
from sympy import Max

x1, y1 = sympy.symbols('x,y', real=True)
xa = sympy.Array([x1, y1])
f1 = 1 * (x1 - 5) ** 4 + 3 * (y1 - 0) ** 2
f2 = Max(x1 - 5, 0) + 3 * abs(y1 - 0)

dfdx1 = sympy.diff(f1, xa)
dfdx1 = sympy.lambdify(xa, dfdx1)

f1 = sympy.lambdify(xa, f1)
f2 = sympy.lambdify(xa, f2)

x2 = sympy.symbols('x', real=True)
f2p2 = 3 * abs(x2 - 0)
dfdx2 = sympy.diff(f2p2, x2)
dfdx2 = sympy.lambdify(x2, dfdx2)


def getF1(x, y):
    return f1(x, y)


def getF2(x, y):
    return f2(x, y)


def getDff2p1(x):
    if x > 5:
        return 1
    else:
        return 0


def getDff2(x, y):
    dx = getDff2p1(x)
    dy = dfdx2(y)
    return dx, dy

def getPF1(x, y):
    x1 = x.copy()
    for i in range(0, x.shape[0]):
        for j in range(0, x.shape[1]):
            if x[i][j] <= 5:
                x1[i][j] = 0
            else:
```

```
                x1[i][j] = x[i][j] - 5


    part2 = 3 * abs(y)

    return x1 + part2


def gradientDescent(x, y, alpha, iters, ii, ss, tt, xx, yy, ff):
    start_time = time.time() * 1000
    i = 0
    s = 0
    while i < iters:
        # dx, dy = dfdx1(x, y)
        dx, dy = getDff2(x, y)
        stepx = alpha * dx
        stepy = alpha * dy
        ii.append(i)
        s = s + 2
        ss.append(s)
        xx.append(x)
        yy.append(y)
        value = f2(x, y)
        ff.append(value)
        used = (time.time() * 1000 - start_time)
        print(used)
        tt.append(used)
        x = x - stepx
        y = y - stepy
        i = i + 1
    return ii, ss, tt, xx, yy, ff


def globalRandomSearch(x, y, iters, ii, ss, tt, xx, yy, ff):
    start_time = time.time() * 1000
    minValue = float('inf')
    minX = random.uniform(x[0], x[1])
    minY = random.uniform(y[0], y[1])
    i = 0
    s = 0
    while i < iters:
        cx = random.uniform(x[0], x[1])
        cy = random.uniform(y[0], y[1])
        value = f2(cx, cy)
        if minValue > value:
            minValue = value
            minX = cx
            minY = cy
        ii.append(i)
        s = s + 1
        ss.append(s)
        xx.append(minX)
        yy.append(minY)
        ff.append(minValue)
        used = (time.time() * 1000 - start_time)
        # print(used)
        tt.append(used)
        i = i + 1
    return ii, ss, tt, xx, yy, ff


def populationBasedSearch(x, y, iters, Nsample, bestP, range, exploitP, ii,
    ss, tt, xx, yy, ff):
    # choose Nsample points
    start_time = time.time() * 1000
    stp = BtmkHeap(bestP)
```

```
    i = 0
    s = 0
    while i < Nsample:
        cx = random.uniform(x[0], x[1])
        cy = random.uniform(y[0], y[1])
        value = f2(cx, cy)
        stp.Push((value, [cx, cy]))
        i = i + 1

    i = 0
    while i < iters:
        datas = stp.BtmK()
        ii.append(i)
        used = (time.time() * 1000 - start_time)
        # print(used)
        tt.append(used)
        smallestXY = stp.getSmallestV()
        xx.append(smallestXY[0])
        yy.append(smallestXY[1])
        ff.append(stp.getSmallest())
        for data in datas:
            xy = data[1]
            j = 0
            while j < exploitP:
                cx = random.uniform(xy[0] - range, xy[0] + range)
                cy = random.uniform(xy[1] - range, xy[1] + range)
                value = f2(cx, cy)
                stp.Push((value, [cx, cy]))
                j = j + 1
        i = i + 1

    return ii, ss, tt, xx, yy, ff


ii, ss, tt, xx, yy, ff = gradientDescent(2, 2, 0.03, 1000, [], [], [], [], [
], [])
# ii1, ss1, tt1, xx1, yy1, ff1 = gradientDescent(2, 2, 0.01, 1000, [], [], [
], [], [], [])
ii2, ss2, tt2, xx2, yy2, ff2 = globalRandomSearch([4, 6], [-
1, 1], 1000, [], [], [], [], [], [])
# ii3, ss3, tt3, xx3, yy3, ff3 = globalRandomSearch([3, 7], [-
2, 2], 15000, [], [], [], [], [], [])
# ii4, ss4, tt4, xx4, yy4, ff4 = globalRandomSearch([0, 10], [-
5, 5], 15000, [], [], [], [], [], [])
ii3, ss3, tt3, xx3, yy3, ff3 = populationBasedSearch([4, 6], [-
1, 1], 1000, 50, 5, 0.1, 5, [], [], [], [], [], [])
ii4, ss4, tt4, xx4, yy4, ff4 = populationBasedSearch([3, 7], [-
2, 2], 1000, 50, 5, 0.1, 5, [], [], [], [], [], [])
ii5, ss5, tt5, xx5, yy5, ff5 = populationBasedSearch([0, 10], [-
5, 5], 1000, 50, 5, 0.1, 5, [], [], [], [], [], [])


plt.figure(figsize=(7, 7))

x = np.arange(1, 7, 0.5)
y = np.arange(-1, 3, 0.5)
X, Y = np.meshgrid(x, y)

contours = plt.contour(X, Y, getPF1(X, Y), 5);
plt.clabel(contours, inline=True, fontsize=10)
# plt.plot(tt, ff, color='b', label='gradient Descent alpha = 0.03')
# plt.plot(tt1, ff1, color='r', label='gradient Descent alpha = 0.01')
# plt.plot(tt2, ff2, color='g', label='global random search x=[4,6] y=[-
1,1]')
# plt.plot(tt3, ff3, color='y', label='global random search x=[3,7] y=[-
2,2]')
```

```
# plt.plot(tt4, ff4, color='c', label='global random search x=[0,10] y=[-5,5]')
# plt.plot(tt1, ss1, color='r', label='gradient Descent alpha = 0.01')
# plt.plot(tt2, ss2, color='g', label='global random search x=[4,6] y=[-1,1]')
# plt.plot(tt, ff, color='b', label='gradient Descent alpha = 0.03 iter=1000')
# plt.plot(tt2, ff2, color='r', label='global random search x=[4,6] y=[-1,1] iter=1000')
# plt.plot(tt3, ff3, color='g', label='population based search x=[4,6] y=[-1,1] iter=1000')
# plt.plot(tt4, ff4, color='c', label='population based search x=[3,7] y=[-2,2] iter=1000')
# plt.plot(tt5, ff5, color='m', label='population based search x=[0,10] y=[-5,5] iter=1000')
plt.plot(xx, yy, color='b', label='gradient Descent alpha = 0.03 iter=1000')

plt.plot(xx2, yy2, color='r', label='global random search x=[4,6] y=[-1,1] iter=1000')
plt.plot(xx3, yy3, color='g', label='population based search x=[4,6] y=[-1,1] iter=1000')
plt.plot(xx4, yy4, color='c', label='population based search x=[3,7] y=[-2,2] iter=1000')
plt.plot(xx5, yy5, color='m', label='population based search x=[0,10] y=[-5,5] iter=1000')
plt.xlabel('x')
plt.ylabel('y')
# plt.xlabel('runing time(millisecond)')
# plt.ylabel('number of Derivative function/function computations')
# plt.xlim((0, 30))
# plt.yscale("log")
plt.legend()
plt.show()
```

c

```
import time
import random

from matplotlib import pyplot as plt

from model import nnModel
from minheap import BtmkHeap


def populationBasedSearch(alpha, beta1, beta2, batchsize, epoches, iters, Nsample, bestP, exploitP, ii, tt, ff):
    # choose Nsample points
    global nnModel
    start_time = time.time()
    stp = BtmkHeap(bestP)
    i = 0
    z = 0
    while i < Nsample:
        calpha = random.uniform(alpha[0], alpha[1])
        cbeta1 = random.uniform(beta1[0], beta1[1])
        cbeta2 = random.uniform(beta2[0], beta2[1])
        cbatchsize = random.randint(batchsize[0], batchsize[1])
        cepoches = random.randint(epoches[0], epoches[1])
        print(str(i) + 'round: aplha=' + str(calpha) + ',beta1=' + str(cbeta1) + ',beta2=' + str(
            cbeta2) + ',batchsize=' + str(cbatchsize) + ',epoches=' + str(cepoches))
        model = nnModel(calpha, cbeta1, cbeta2, cbatchsize, cepoches)
        acc = model.getAuc()
```

```
        print(str(i) + 'round finished, get test data acc=' + str(acc))
        stp.Push((acc, [calpha, cbeta1, cbeta2, cbatchsize, cepoches]))
        i = i + 1

        used = (time.time() - start_time)
        # print(used)
        tt.append(used)
        ff.append(stp.getSmallest())

    while i < iters + Nsample:
        datas = stp.BtmK()

        for data in datas:
            paras = data[1]
            j = 0
            while j < exploitP:
                minAlpha = paras[0] - 0.05
                if minAlpha < alpha[0]:
                    minAlpha = alpha[0]
                maxAlpha = paras[0] + 0.05
                if maxAlpha > alpha[1]:
                    maxAlpha = alpha[1]
                calpha = random.uniform(minAlpha, maxAlpha)

                minBeta1 = paras[1] - 0.1
                if minBeta1 < beta1[0]:
                    minBeta1 = beta1[0]
                maxBeta1 = paras[1] + 0.1
                if maxBeta1 > beta1[1]:
                    maxBeta1 = beta1[1]
                cbeta1 = random.uniform(minBeta1, maxBeta1)

                minBeta2 = paras[2] - 0.1
                if minBeta2 < beta2[0]:
                    minBeta2 = beta2[0]
                maxBeta2 = paras[2] + 0.1
                if maxBeta2 > beta1[1]:
                    maxBeta2 = beta1[1]
                cbeta2 = random.uniform(minBeta2, maxBeta2)

                minbatchsize = paras[3] - 5
                if minbatchsize < batchsize[0]:
                    minbatchsize = batchsize[0]
                maxbatchsize = paras[3] + 5
                if maxbatchsize > batchsize[1]:
                    maxbatchsize = batchsize[1]
                cbatchsize = random.randint(minbatchsize, maxbatchsize)

                minEpoches = paras[4] - 10
                if minEpoches < epoches[0]:
                    minEpoches = epoches[0]
                maxEpoches = paras[4] + 10
                if maxEpoches > epoches[1]:
                    maxEpoches = epoches[1]
                cepoches = random.randint(minEpoches, maxEpoches)
                print(str(i) + 'round,' + str(j) + 'exploit point: aplha=' +
    str(calpha) + ',beta1=' + str(
                    cbeta1) + ',beta2=' + str(
                    cbeta2) + ',batchsize=' + str(cbatchsize) + ',epoches=' +
     str(cepoches))
                model = nnModel(calpha, cbeta1, cbeta2, cbatchsize, cepoches)

                acc = model.getAuc()
                print(
                    str(i) + 'round finished,' + str(j) + 'exploit point:  ge
    t test data acc=' + str(stp.getSmallest()))
```

```
                stp.Push((acc, [calpha, cbeta1, cbeta2, cbatchsize, cepoches]))
                j = j + 1

        ii.append(i)
        used = (time.time() - start_time)
        # print(used)
        tt.append(used)
        ff.append(stp.getSmallest())
        i = i + 1

    paras = stp.getSmallestV()
    print('the best parameters are:aplha=' + str(paras[0]) + ',beta1=' + str(
        paras[1]) + ',beta2=' + str(
        paras[2]) + ',batchsize=' + str(paras[3]) + ',epoches=' + str(paras[4]))
    return ii, tt, ff


def globalRandomSearch(alpha, beta1, beta2, batchsize, epoches, iters, ii, tt, ff):
    start_time = time.time()
    minloss = float("inf")
    bestPara = (alpha[0], beta1[0], beta2[0], batchsize[0], epoches[0])
    i = 0
    while i < iters:
        calpha = random.uniform(alpha[0], alpha[1])
        cbeta1 = random.uniform(beta1[0], beta1[1])
        cbeta2 = random.uniform(beta2[0], beta2[1])
        cbatchsize = random.randint(batchsize[0], batchsize[1])
        cepoches = random.randint(epoches[0], epoches[1])
        print(str(i) + 'round: aplha=' + str(calpha) + ',beta1=' + str(cbeta1) + ',beta2=' + str(
            cbeta2) + ',batchsize=' + str(cbatchsize) + ',epoches=' + str(cepoches))
        model = nnModel(calpha, cbeta1, cbeta2, cbatchsize, cepoches)
        loss = model.getAuc()
        print(str(i) + 'round finished, get test data acc=' + str(minloss))

        if loss < minloss:
            minloss = loss
            bestPara = (calpha, cbeta1, cbeta2, cbatchsize, cepoches)

        ii.append(i)
        used = (time.time() - start_time)
        # print(used)
        tt.append(used)
        ff.append(minloss)
        i = i + 1

    print('best acc:' + str(minloss))
    print(bestPara)
    return ii, tt, ff


ii, tt, ff = globalRandomSearch([0, 0.3], [0, 1], [0, 1], [16, 128], [10, 50], 50, [], [], [])
ii2, tt2, ff2 = globalRandomSearch([0, 0.3], [0, 1], [0, 1], [16, 128], [10, 50], 120, [], [], [])
ii4, tt4, ff4 = populationBasedSearch([0, 1], [0, 1], [0, 1], [16, 128], [10, 50], 8, 10, 3, 3, [], [], [])
plt.figure(figsize=(7, 7))
plt.plot(tt, ff, color='b', label='global random search iters = 50')
plt.plot(tt2, ff2, color='r', label='global random search iters = 120')
```

```
plt.plot(ii4, ff4, color='b', label='population based search ')
plt.xlabel('iterations')
plt.ylabel('Changes in hyperparameters')
plt.legend()
plt.show()
```

minheap.py

```
import heapq


class BtmkHeap(object):
    def __init__(self, k):
        self.k = k
        self.data = []

    def Push(self, elem):
        num1 = elem[0]
        num1 = -num1
        num2 = elem[1]

        if len(self.data) < self.k:
            heapq.heappush(self.data, (num1, num2))
        else:
            topk_small = self.data[0][0]
            if elem[0] > topk_small:
                heapq.heapreplace(self.data, (num1, num2))

    def BtmK(self):
        return sorted([(-x[0], x[1]) for x in self.data])

    def getSmallest(self):
        return -heapq.nlargest(1, self.data)[0][0]

    def getSmallestV(self):
        return heapq.nlargest(1, self.data)[0][1]
```

model.py

```
from keras import regularizers
from keras.layers import Conv2D, Dropout, Flatten, Dense
from tensorflow import keras


class nnModel(object):
    def __init__(self, alpha, beta1, beta2, batchsize, epoches):
        self.alpha = alpha
        self.beta1 = beta1
        self.beta2 = beta2
        self.batchsize = batchsize
        self.epoches = epoches

        self.num_classes = 10
        self.input_shape = (32, 32, 3)

        # the data, split between train and test sets
        (x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_da
ta()
        n = 5000
        x_train = x_train[1:n];
        y_train = y_train[1:n]
        # x_test=x_test[1:500]; y_test=y_test[1:500]

        # Scale images to the [0, 1] range
        self.x_train = x_train.astype("float32") / 255
        self.x_test = x_test.astype("float32") / 255
        # print("orig x_train shape:", x_train.shape)
```

```
28.
29.         # convert class vectors to binary class matrices
30.         self.y_train = keras.utils.to_categorical(y_train, self.num_classes)
31.         self.y_test = keras.utils.to_categorical(y_test, self.num_classes)
32.
33.     def getAuc(self):
34.         model = keras.Sequential()
35.         model.add(Conv2D(16, (3, 3), padding='same', input_shape=self.x_train
    .shape[1:], activation='relu'))
36.         model.add(Conv2D(16, (3, 3), strides=(2, 2), padding='same', activati
    on='relu'))
37.         model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
38.         model.add(Conv2D(32, (3, 3), strides=(2, 2), padding='same', activati
    on='relu'))
39.         model.add(Dropout(0.5))
40.         model.add(Flatten())
41.         model.add(Dense(self.num_classes, activation='softmax', kernel_regula
    rizer=regularizers.l1(0.0001)))
42.
43.         adam = keras.optimizers.Adam(learning_rate=self.alpha, beta_1=self.be
    ta1, beta_2=self.beta2)
44.         model.compile(loss="categorical_crossentropy", optimizer=adam, metric
    s=["accuracy"])
45.
46.
47.         batch_size = self.batchsize
48.         epochs = self.epoches
49.         history = model.fit(self.x_train, self.y_train, batch_size=batch_size
    , epochs=epochs, validation_split=0.1,
50.                             verbose=0)
51.
52.         loss = model.evaluate(self.x_test, self.y_test)[0]
53.         if loss > 5:
54.             loss = 5
55.         return loss
```