Optimisation Algorithms for Data Analysis Final Assignment

Jiaming Deng 22302794

In this assignment, the first 50000 samples are used for training the model.

(a) Algorithm selection

In this section I focus on two gradient-free algorithms that I have chosen, including the global search random algorithm and the bayesian optimization algorithm. These two algorithms have different search algorithms, the global search random algorithm is a global search methods and the bayesian optimization algorithm is a global model-based methods.

The global search random algorithm is a class of global search methods that explore the entire search space by randomly sampling points. The algorithm generates random solutions in the search space and evaluates them to determine if they are better than the current best solution. The process is repeated until the algorithm converges to the optimal solution.

The bayesian optimisation algorithms, on the other hand, are model-based methods that use a probabilistic model to approximate an unknown function. The algorithm iteratively updates the model based on the observed data and selects the next point to be evaluated based on a collection function that balances exploration and exploitation trade-offs. The collection function takes into account the uncertainty of the model and the potential improvement of the solution.

(b) Algorithm implementation

1. Global search random algorithm

Firstly, the input parameters of the global search random algorithm are the function, the amount of data, the dataset, and the number of iterations. The algorithm is implemented by first reading the data of the dataset into the l and u arrays, where l stores the minimum value and u stores the maximum value. The best x and f values are initialised as X, and F as 0 and the largest int type value respectively. Next it iterate through itr_times several times, each time taking a random number from the range of l and u and putting it into the array, and calculating the function value, updating the best x and f values if the function value is smaller than the best value.

```
def random_search(f, n, data_range, N):
    l = []
    u = []
    for Range in data_range:
        l.append(Range[0])
    for Range in data_range:
        u.append(Range[1])
    X = 0
    F = sys.maxsize
    x_list = []
    f_list = []
    for i in range(N):
        data_x = []
        for j in range(n):
            x = uniform(l[j], u[j])
            data_x.append(x)
        data_f = f(data_x[0], data_x[1])
        if data_f < F:
            X = deepcopy(data_x)
            F = data_f
        x_list.append(deepcopy(X))
        f_list.append(F)
    return x_list, f_list
```

2. Bayesian optimization algorithm

Next I will describe how to start using the Bayesian optimization algorithm. First, I define the pbounds, which include the hyperparameters SGD mini-batch size, Adam α, β1, β2 and constant step size α, and specify their range of values based on the input x_rng. Next, I initialise the Bayesian optimiser using the BayesianOptimization object and specify the function to be optimised, the range of hyperparameters, the random state and specify the data to be printed during training. Finally, I call the maxmize function to specify some parameters. xi specifies the degree of exploration, which can be reduced if you want the function to converge quickly; alpha is used to smooth the variance when calculating the variance; and kappa is 2.576, as this corresponds to the 95% confidence interval and is used to strike a balance between exploration and exploitation. In addition to kappa as an empirical value, the values of xi, acq and alpha will be explored in the next experiments.

```
def bayesian_optimisation(f, x_rng):
    pbounds = {
        'batch_size': (x_rng[0][0], x_rng[0][1]),
        'alpha': (x_rng[1][0], x_rng[1][1]),
        'beta1': (x_rng[2][0], x_rng[2][1]),
        'beta2': (x_rng[3][0], x_rng[3][1]),
        'epochs': (x_rng[4][0], x_rng[4][1])
    }
    optimizer = BayesianOptimization(
        f=f,
        pbounds=pbounds,
        random_state=42,
        verbose=1
    )

    optimizer.maximize(
        init_points=5,
        n_iter=20,
        acq='ei',
        xi=0.01,
        kappa=2.576,
        alpha=1e-5,
        n_restarts_optimizer=5,
        normalize_y=True
    )
    xs = []
    fs = []
    for i, res in enumerate(optimizer.res):
        print("Iteration {}: \n\t{}".format(i, res))
        xs.append([res['params'][key] for key in pbounds.keys()])
        fs.append(res['target'])
    return xs, fs
```

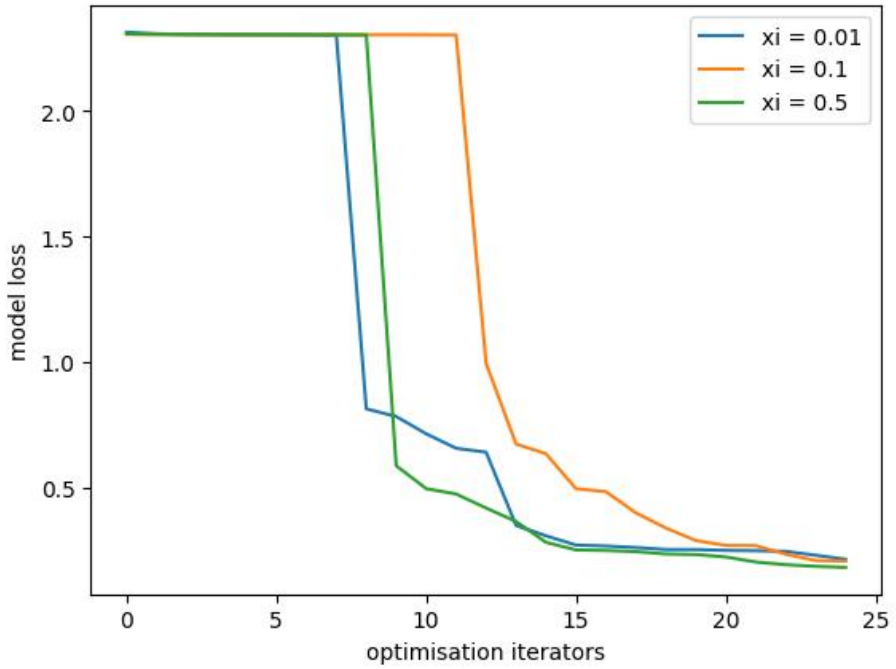
3. Grid search (baseline)

The grid search algorithm is implemented by setting all hyperparameters to a reasonable range, trying all combinations of hyperparameters and calculating the loss value of the convolution function.

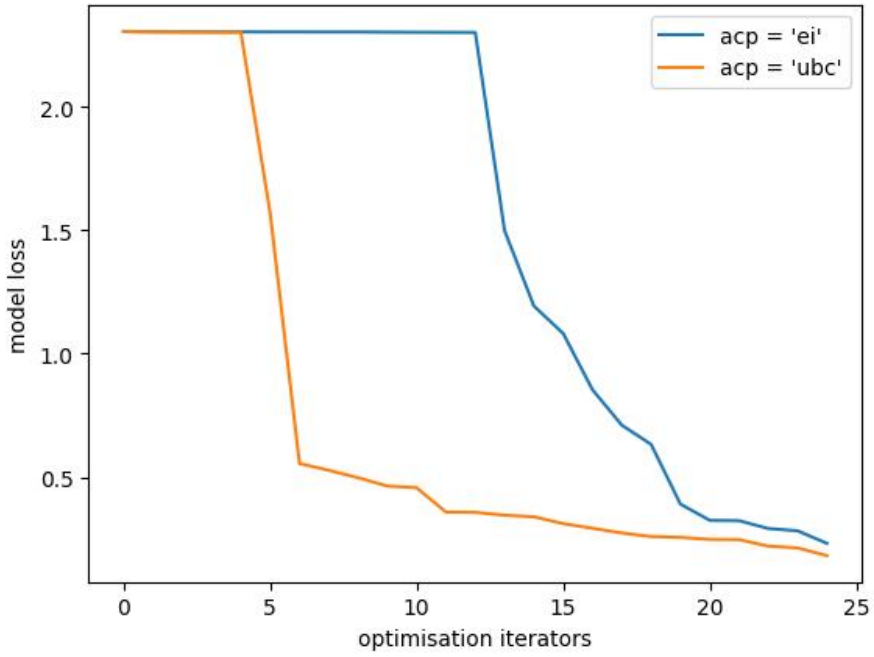
```
for batch_size in batches_list:
    for alpha in alpha_list:
        for beta1 in beta1_list:
            for beta2 in beta2_list:
                for epochs in epochs_list:
                    x = [batch_size, alpha, beta1, beta2, epochs]
                    loss = f(*x)
                    xs.append(x)
                    losses.append(loss)
                    if loss < best_loss:
                        best_loss = loss
                        best_x = x
print("Best loss:", best_loss)
print("Best x:", best_x)
```

(c) Algorithm parameter selection

In the search for the best parameters for Bayesian optimization, I set the hyperparameters for Bayesian optimization to range from batch size [30, 60], alpha [0.001, 0.1], beta1 [0.25, 0.99], beta2 [0.99, 0.999] and epoch [10, 30], with some randomness in the choice of parameters to be more accurate understanding of how the chosen parameters behave in different combinations of hyperparameters.
First I chose xi from the three values [0.01,0.1,0.5] to obtain the model loss for the different combinations of hyperparameters shown below. xi=0.1 performed relatively worst and xi=0.5 performed relatively best because xi is the degree of exploration and xi=0.1 is not high enough to make the function converge slowly.



Secondly, acp refers to Active Covariance Matrix Adaptation - Population Size and has two options, one is EI, or Expected Improvement, which is used to select the next most likely sample point to be evaluated by calculating the expected value of the boost between the currently obtained sample point and the likely sample point. UCB, which stands for Upper Confidence Bound, is used to select the next parameter point to be evaluated by calculating the upper confidence interval of the observed sample. By looking at the graph below it can be seen that acp='ucb' performs better than acp='ei' in most iterations, as the UCB algorithm aims to minimise the upper confidence interval and is therefore more advantageous when searching for more complex objective functions.



(d) Algorithm performance comparison

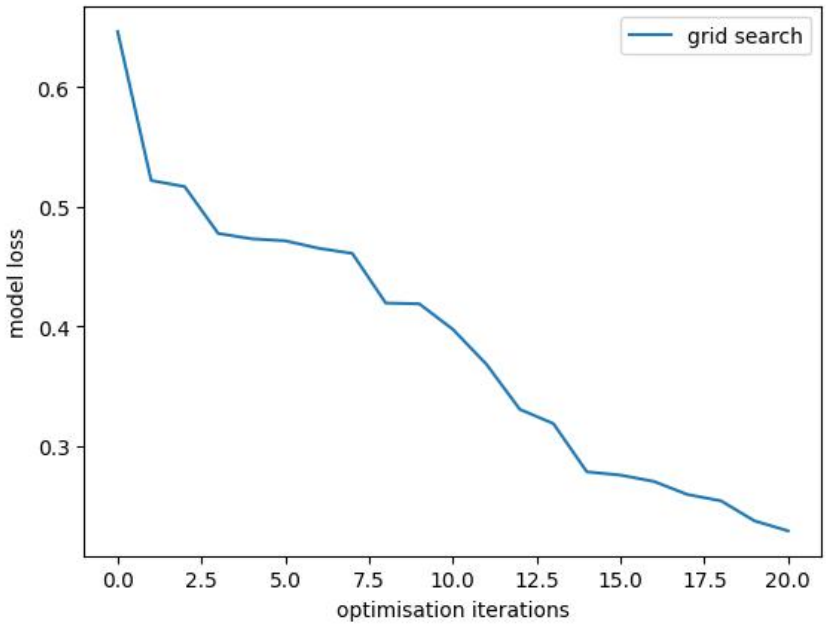
1. Grid search (Baseline)

I implemented a grid of hyperparameter values as the baseline model as following. Firstly I specify that the range of batch_size is [30, 60], alpha is [0.0001, 0.1], beta1 is [0.25, 0.99], beta2 is [0.9, 0.9999] and epoch is [5, 30]. The arguments are all integers in the convenient range of the integer range, and the arguments are 11 numbers equidistant in the specified range of the fractional numbers. I used the number of iterations and the loss value as criteria for evaluating the optimisation algorithm, as it is clear that the effect of each iteration on the loss value can be observed to objectively evaluate the speed and results of the algorithm.

The results of the calculation are as follows, and it can be seen that the best performing combination of parameters within the hyperparameters I specified is mini-batch size=59, alpha=0.5, beta1=0.25, beta2=0.9999, epoch=34. The minimum

loss value in 20 iterations=0.2289.

```
Best loss: 0.22895396
Best x: [59, 0.050050000000000004, 0.25, 0.9999, 34]
```

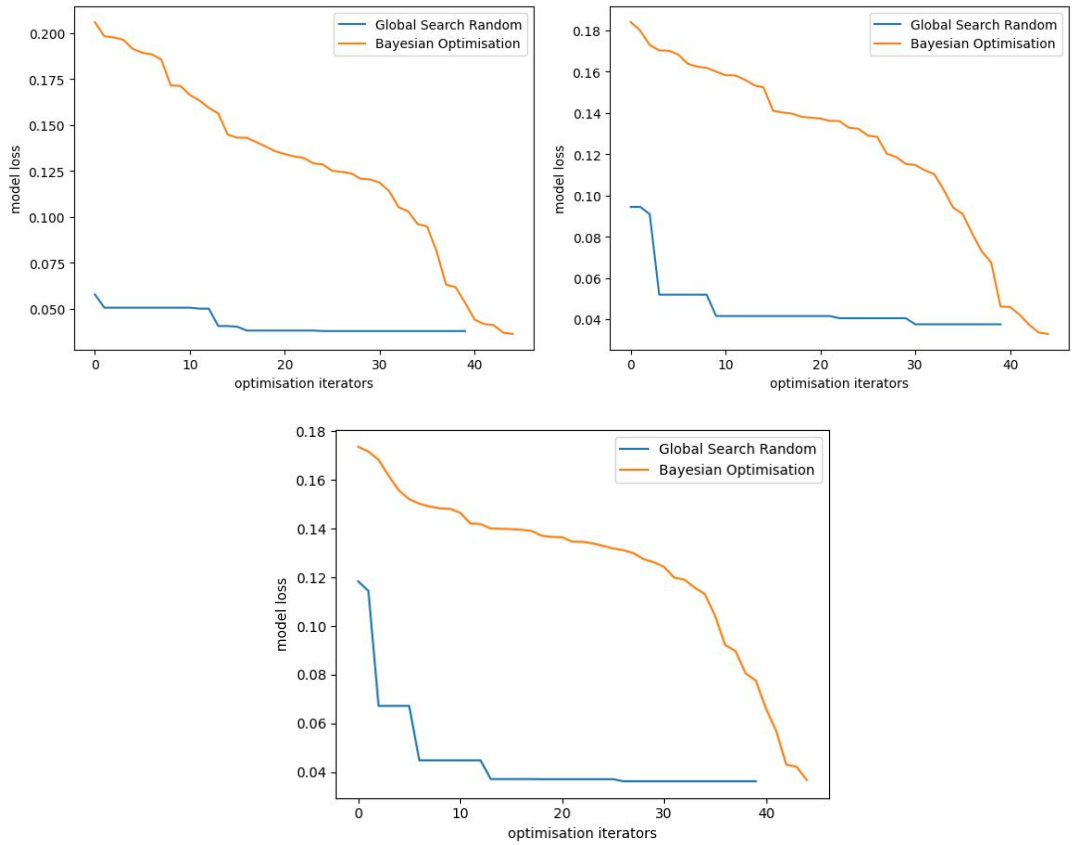


2. In this section, I use the model prediction training dataset to calculate the loss values of the model, plot the ML loss function against the optimisation iterations and use the lowest value as the performance metric.
First I choose to fix the values of alpha, beta1, beta2 and epoch and only change the value of mini-batch size to compare the performance of the two optimisation functions. Therefore, I did not plot the results of the baseline model in each comparison between the global search randomisation algorithm and the Bayesian optimisation algorithm. Instead, I plotted a separate graph of the baseline results and calculated the minimum loss values to compare with the results of the two algorithms.

I fixed alpha=0.001, beta1=0.9, beta2=0.999, epoch=20, and the range of the mini-batch was 1 to 128. The parameters for Bayesian optimization were chosen from the best parameters obtained in the previous section, acq= 'ucb', xi=0.5. Because SGD involves randomization, to reduce the results' volatility, I ran three rounds to obtain three results for a combined comparison.

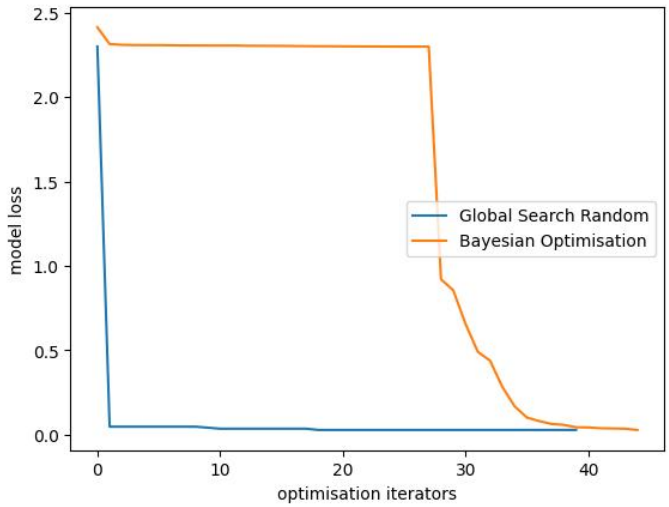
As can be seen from the three graphs below, the minimum loss value of the model obtained by Bayesian Optimisation is smaller than that obtained by the Global Search Random algorithm. Starting somewhere between the 10th and 20th iterations, there is no suitable mini-batch size to reduce the loss value of the Global Search Random algorithm as the number of iterations increases. Although the loss value of the Bayesian Optimisation algorithm is larger at the beginning of the iteration, it

continues to decrease as the number of iterations increases. At the 40th iteration is the loss value of Bayesian Optimisation is smaller than Global Search Random, therefore, Bayesian Optimisation performs better than Global Search and both algorithms outperform the baseline.

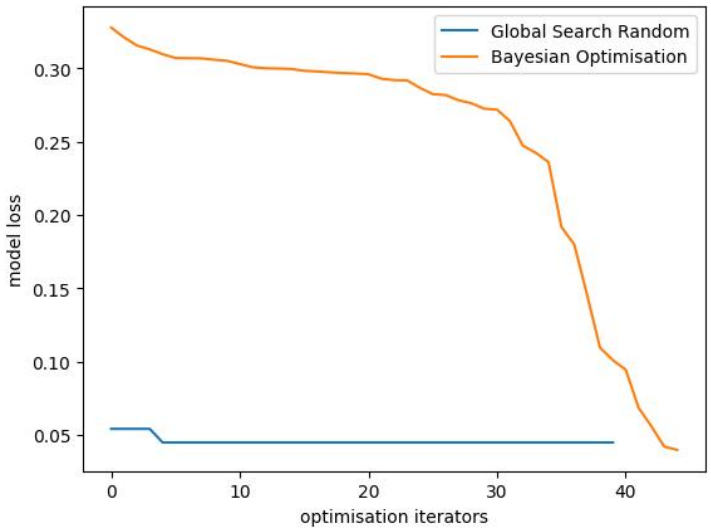


Second, I fixed mini-batch size=30, epoch=20, alpha ranged from 0.001 to 0.1, beta1 ranged from 0.25 to 0.99, beta2 ranged from 0.9 to 0.999. The result is as follows.

Looking at the image, it can be seen that Bayesian optimization has a much higher model loss value at the beginning of the iteration than the global search random algorithm is basically consistent. However, as the number of iterations increases, the Global Search Random algorithm converges to a very low level in the first few iterations, but the Bayesian optimization makes the loss value of the model slowly decrease, and it is not the same as the Global Search Random algorithm until the 40th iteration. So the global search random algorithm performs better. Both of thses algorithms are better than baseline.



Finally, I fixed mini-batch size=40, alpha=0.001, beta1=0.9, beta2=0.999, and designed the range of epoch to be 5 to 30.
Observing the image, it can be seen that the model loss value of Bayesian optimization at the beginning of the iteration is much higher than that of the global search random algorithm. same. The model loss values are basically the same. So, in general, the Global Search Random algorithm performs better. Both of thses algorithms are better than baseline.

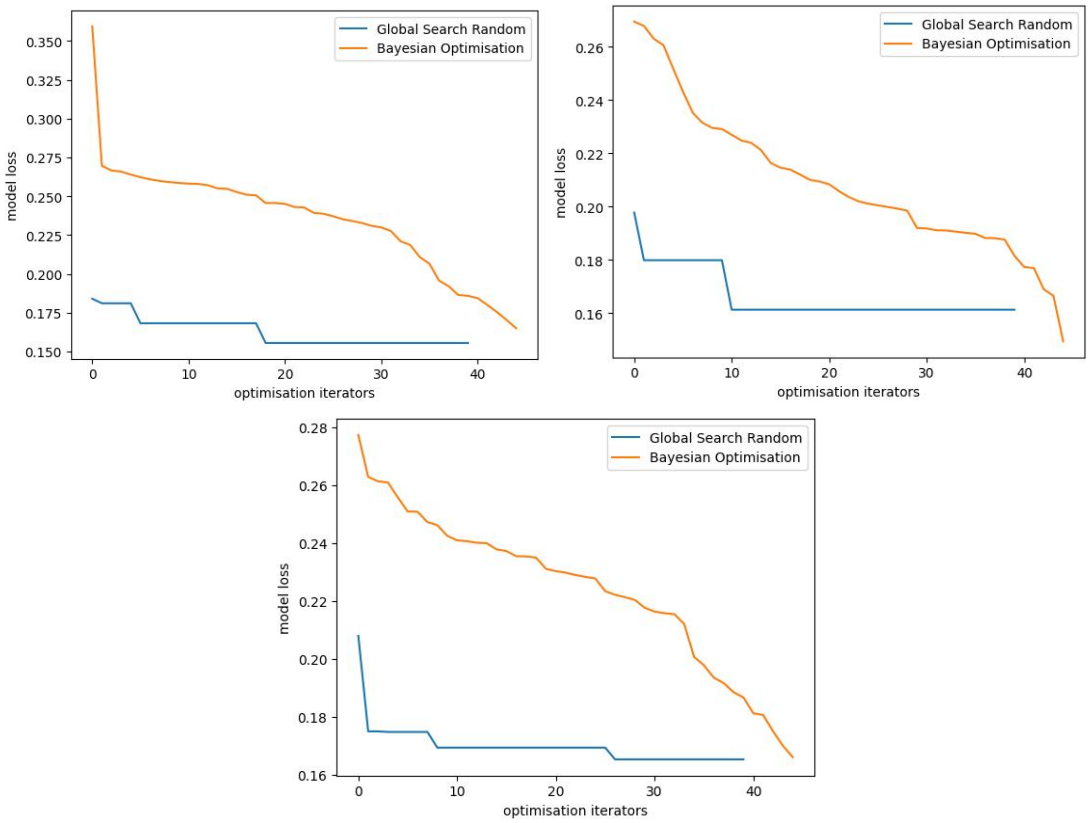


3. In this section I use the model to predict the test data set to calculate the loss value of the model, plot the ML loss function vs optimization iterations and use the lowest value as a performance measure.

Firstly I choose to fix the values of alpha, beta1, beta2 and epoch and only change the value of the mini-batch size to compare the performance of the two optimization functions. Therefore, I do not plot the results of the baseline model in each comparison between the Global Search Random algorithm and the Bayesian Optimisation algorithm. Instead, I plot a separate graph of the baseline results and calculate the minimum loss value to compare with the results of the two algorithms.

I specified alpha value of 0.001, beta1 of 0.9, beta2 of 0.999, epoch of 20, and mini-batch size ranging from 1 to 128. The parameters for Bayesian optimization were chosen from the best parameters obtained in the previous section, acq='ucb', xi=0.5. Because SGD involves randomisation, to reduce the volatility of the results, I ran three runs to obtain three results for a combined comparison.

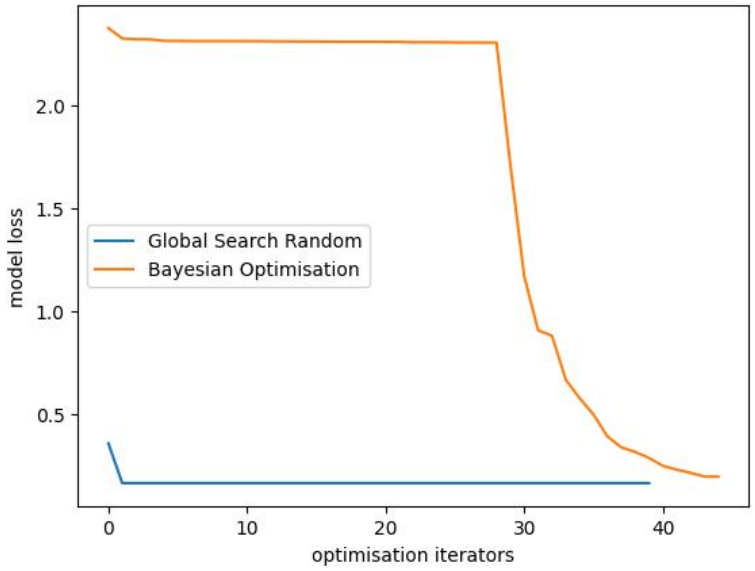
As can be seen from the three graphs below, the minimum loss value of the model obtained by Bayesian Optimization is smaller than the minimum loss value of the model obtained by the Global Search Random algorithm in all but the first graph. It can also be seen that the Global Search Random algorithm basically fails to find a suitable mini-batch size to reduce the model loss value after a certain iteration between the 10th and 20th iteration as the number of iterations increases. Although Bayesian Optimization has a large model loss at the beginning of the iteration, it keeps decreasing as the number of iterations increases, until the number of iterations of 40 that I set, the loss value still keeps decreasing. Therefore, Bayesian optimization performs better than global search, and both algorithms are better than baseline. Both of thses algorithms are better than baseline.



Second, I fix mini-batch size=30, epoch=20, alpha range from 0.001 to 0.1, beta1 range from 0.25 to 0.99 and beta2 range from 0.9 to 0.999. The result is as follows.

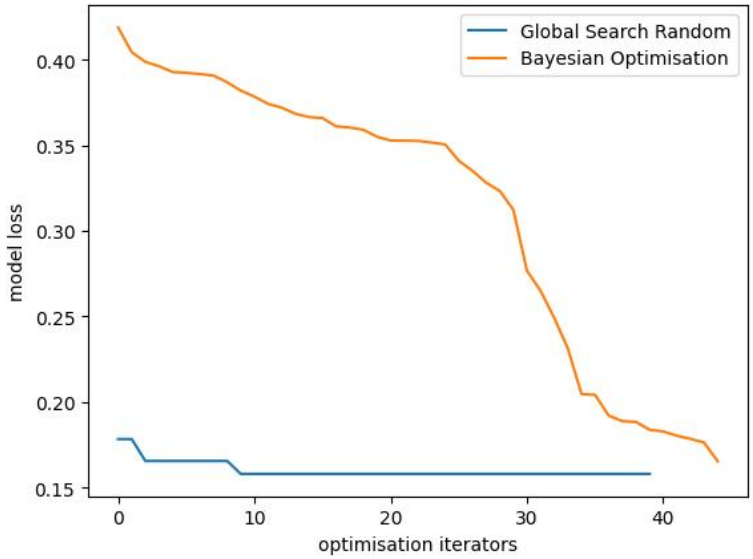
Looking at the images it can be seen that Bayesian Optimisation has a much higher

model loss value at the beginning of the iteration than the Global Search Random algorithm, as Bayesian Optimisation typically uses Gaussian Process Regression to model the relationship between hyperparameters and model loss. At the beginning of the iteration, due to the small number of samples, the Gaussian process model may have a large variance, resulting in less accurate predictions for the next hyperparameter combination. However, as the number of iterations increases, Bayesian optimisation at the 40th iteration and the Global Search Random algorithm obtain essentially the same value of model loss. However, overall the Global Search Random algorithm performed better. Both of thses algorithms are better than baseline.



Finally, I fixed mini-batch size=40, alpha=0.001, beta1=0.9, beta2=0.999, and designed the range of epoch to be 5 to 30.

Observing the image, it can be seen that the model loss value of Bayesian optimization at the beginning of the iteration is much higher than that of the Global Search Random algorithm, but as the number of iterations increases, the Bayesian optimization at the 40th iteration is the same as the Global Search Random algorithm. The model loss values are basically the same. Therefore, in general, the performance of the Global Search Random algorithm is better. Both of thses algorithms are better than baseline.



Appendix

```
import matplotlib.pyplot as plt
from copy import deepcopy
from random import uniform
from tensorflow import keras
from keras import regularizers, layers
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D
from keras.losses import CategoricalCrossentropy
from keras.optimizers import Adam
from bayes_opt import BayesianOptimization
import numpy as np


def get_model_loss(batch_size, alpha, beta1, beta2, epochs):
    batch_size = int(batch_size)  # ensure random values are ints
    epochs = int(epochs)  # ensure random values are ints
    num_classes = 10
    (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
    n = 50000
    x_train = x_train[1:n]
    y_train = y_train[1:n]
    x_train = x_train.astype("float32") / 255
    x_test = x_test.astype("float32") / 255
```

```
    y_train = keras.utils.to_categorical(y_train, num_classes)
    y_test = keras.utils.to_categorical(y_test, num_classes)
    model = keras.Sequential(
        [
            keras.Input(shape=(28, 28, 1)),
            layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
            layers.MaxPooling2D(pool_size=(2, 2)),
            layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
            layers.MaxPooling2D(pool_size=(2, 2)),
            layers.Flatten(),
            layers.Dropout(0.5),
            layers.Dense(num_classes, activation="softmax"),
        ]
    )
    optimizer = Adam(learning_rate=alpha, beta_1=beta1, beta_2=beta2)
    model.compile(loss="categorical_crossentropy", optimizer=optimizer,
                  metrics=["accuracy"])
    model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
              validation_split=0.1, verbose=0)
    y_predicts_test = model.predict(x_test)
    loss = CategoricalCrossentropy()
    return loss(y_test, y_predicts_test).numpy()


def get_model_loss_train(batch_size, alpha, beta1, beta2, epochs):
    batch_size = int(batch_size)  # ensure random values are ints
    epochs = int(epochs)  # ensure random values are ints
    num_classes = 10
    (x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
    n = 5000
```

```
    x_train = x_train[1:n]
    y_train = y_train[1:n]
    x_train = x_train.astype("float32") / 255
    x_test = x_test.astype("float32") / 255
    y_train = keras.utils.to_categorical(y_train, num_classes)
    y_test = keras.utils.to_categorical(y_test, num_classes)
    model = keras.Sequential(
        [
            keras.Input(shape=(28, 28, 1)),
            layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
            layers.MaxPooling2D(pool_size=(2, 2)),
            layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
            layers.MaxPooling2D(pool_size=(2, 2)),
            layers.Flatten(),
            layers.Dropout(0.5),
            layers.Dense(num_classes, activation="softmax"),
        ]
    )
    optimizer = Adam(learning_rate=alpha, beta_1=beta1, beta_2=beta2)
    model.compile(loss="categorical_crossentropy", optimizer=optimizer,
                  metrics=["accuracy"])
    model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
              validation_split=0.1, verbose=0)
    y_predicts_train = model.predict(x_train)
    loss = CategoricalCrossentropy()
    return loss(y_test, y_predicts_train).numpy()


def global_search_random(f, n, x_rng, N):
    l = [r[0] for r in x_rng]
    u = [r[1] for r in x_rng]
    best_x = None
    best_f = 1e20
    xs, fs = [], []
```

```
    vv = 0
    for _ in range(N):
        this_x = [uniform(l[i], u[i]) for i in rang
e(n)]
        this_f = f(*this_x)
        if this_f < best_f:
            best_x = deepcopy(this_x)
            best_f = this_f
        xs.append(deepcopy(best_x))
        fs.append(best_f)
        print(vv, best_f, best_x)
        vv += 1
    return xs, fs


def bayesian_optimisation(f, x_rng):
    pbounds = {
        'batch_size': (x_rng[0][0], x_rng[0][1]),

        'alpha': (x_rng[1][0], x_rng[1][1]),
        'beta1': (x_rng[2][0], x_rng[2][1]),
        'beta2': (x_rng[3][0], x_rng[3][1]),
        'epochs': (x_rng[4][0], x_rng[4][1])
    }
    optimizer = BayesianOptimization(
        f=f,
        pbounds=pbounds,
        random_state=42,
        verbose=1
    )

    optimizer.maximize(
        init_points=5,
        n_iter=20,
        acq= 'ucb',
        xi=0.01,
        kappa=2.576,
        alpha=1e-5,
        n_restarts_optimizer=5,
        normalize_y=True
    )
    xs = []
    fs = []
    for i, res in enumerate(optimizer.res):
```

```
            print("Iteration {}: \n\t{}".format(i, res))
            xs.append([res['params'][key] for key in pbounds.keys()])
            fs.append(res['target'])
    return xs, fs


def mini_batch():
    n = 5
    x_rng = [
        [1, 128],
        [0.001, 0.001],
        [0.9, 0.9],
        [0.999, 0.999],
        [20, 20]
    ]
    print('====================== mini batch ======================')
    print('global search random')
    gsr_xs, gsr_fs = global_search_random(get_model_loss, n, x_rng, N=50)
    print('bayesian optimisation')
    bay_xs, bay_fs = bayesian_optimisation(get_model_loss, x_rng)
    return gsr_xs, gsr_fs, bay_xs, bay_fs


def mini_batch_train():
    n = 5
    x_rng = [
        [1, 128],
        [0.001, 0.001],
        [0.9, 0.9],
        [0.999, 0.999],
        [20, 20]
    ]
    print('====================== mini batch ======================')
    print('global search random')
    gsr_xs, gsr_fs = global_search_random(get_model_loss_train, n, x_rng, N=50)
    print('bayesian optimisation')
```

```
166.       bay_xs, bay_fs = bayesian_optimisation(get_m
   odel_loss_train, x_rng)
167.       return gsr_xs, gsr_fs, bay_xs, bay_fs
168.
169.
170.   def adam_params():
171.       n = 5
172.       x_rng = [
173.           [30, 30],
174.           [0.001, 0.1],
175.           [0.25, 0.99],
176.           [0.9, 0.9999],
177.           [20, 20]
178.       ]
179.       print('====================== adam params ==
   ====================')
180.       print('global search random')
181.       gsr_xs, gsr_fs = global_search_random(get_mo
   del_loss, n, x_rng, N=50)
182.       print('bayesian optimisation')
183.       bay_xs, bay_fs = bayesian_optimisation(get_m
   odel_loss, x_rng)
184.       return gsr_xs, gsr_fs, bay_xs, bay_fs
185.
186.
187.   def adam_params_train():
188.       n = 5
189.       x_rng = [
190.           [45, 45],
191.           [0.1, 0.0001],
192.           [0.25, 0.99],
193.           [0.9, 0.9999],
194.           [20, 20]
195.       ]
196.       print('====================== adam params ==
   ====================')
197.       print('global search random')
198.       gsr_xs, gsr_fs = global_search_random(get_mo
   del_loss_train, n, x_rng, N=50)
199.       print('bayesian optimisation')
200.       bay_xs, bay_fs = bayesian_optimisation(get_m
   odel_loss_train, x_rng)
201.       return gsr_xs, gsr_fs, bay_xs, bay_fs
202.
```

```
203.
204. def epochs():
205.     n = 5
206.     x_rng = [
207.         [45, 45],
208.         [0.001, 0.001],
209.         [0.9, 0.9],
210.         [0.999, 0.999],
211.         [5, 30]
212.     ]
213.     print('====================== epochs =======
     ==============')
214.     print('global search random')
215.     gsr_xs, gsr_fs = global_search_random(get_mo
     del_loss, n, x_rng, N=50)
216.     print('bayesian optimisation')
217.     bay_xs, bay_fs = bayesian_optimisation(get_m
     odel_loss, x_rng)
218.     return gsr_xs, gsr_fs, bay_xs, bay_fs
219.
220.
221. def epochs_train():
222.     n = 5
223.     x_rng = [
224.         [45, 45],
225.         [0.001, 0.001],
226.         [0.9, 0.9],
227.         [0.999, 0.999],
228.         [5, 30]
229.     ]
230.     print('====================== epochs =======
     ==============')
231.     print('global search random')
232.     gsr_xs, gsr_fs = global_search_random(get_mo
     del_loss_train, n, x_rng, N=50)
233.     print('bayesian optimisation')
234.     bay_xs, bay_fs = bayesian_optimisation(get_m
     odel_loss_train, x_rng)
235.     return gsr_xs, gsr_fs, bay_xs, bay_fs
236.
237.
238. def plot_results(gsr_fs, bay_fs):
239.     gsr_xs, bay_xs = list(range(len(gsr_fs))), l
     ist(range(bay_fs))
```

```
240.       plt.plot(gsr_xs, gsr_fs, label='Global Searc
   h Random')
241.       plt.plot(bay_xs, bay_fs, label='Bayesian Opt
   imisation')
242.       plt.xlabel('optimisation iterators')
243.       plt.ylabel('model loss')
244.       plt.legend()
245.       plt.show()
246.
247.
248.  def grid_search(f, x_rng):
249.       batches_list = np.arange(x_rng[0][0], x_rng[
   0][1] + 1)
250.       alpha_list = np.linspace(x_rng[1][0], x_rng[
   1][1], num=11)
251.       beta1_list = np.linspace(x_rng[2][0], x_rng[
   2][1], num=11)
252.       beta2_list = np.linspace(x_rng[3][0], x_rng[
   3][1], num=11)
253.       epochs_list = np.arange(x_rng[4][0], x_rng[4]
   [1] + 1)
254.
255.       best_loss = np.inf
256.       best_x = None
257.       losses = []
258.       xs = []
259.       for batch_size in batches_list:
260.           for alpha in alpha_list:
261.               for beta1 in beta1_list:
262.                   for beta2 in beta2_list:
263.                       for epochs in epochs_list:
264.                            x = [batch_size, alpha,
   beta1, beta2, epochs]
265.                           loss = f(*x)
266.                           xs.append(x)
267.                           losses.append(loss)
268.                           if loss < best_loss:
269.                               best_loss = loss
270.                               best_x = x
271.       print("Best loss:", best_loss)
272.       print("Best x:", best_x)
273.
274.       # plot search process
275.       N = len(losses)
```

```
276.       X = list(range(N))
277.       plt.plot(X, losses, label='grid search')
278.       plt.xlabel('optimisation iterations')
279.       plt.ylabel('model loss')
280.       plt.legend()
281.       plt.show()
282.
283.       return xs, losses
284.
285.
286.  # run this tonight
287.  def compares():
288.       print('====================== test dataset =
      ======================')
289.       gsr_xs, gsr_fs, bay_xs, bay_fs = mini_batch()

290.       plot_results(gsr_fs, bay_fs)
291.
292.       gsr_xs, gsr_fs, bay_xs, bay_fs = mini_batch()

293.       plot_results(gsr_fs, bay_fs)
294.
295.       gsr_xs, gsr_fs, bay_xs, bay_fs = mini_batch()

296.       plot_results(gsr_fs, bay_fs)
297.
298.       gsr_xs, gsr_fs, bay_xs, bay_fs = adam_params
      ()
299.       plot_results(gsr_fs, bay_fs)
300.
301.       gsr_xs, gsr_fs, bay_xs, bay_fs = epochs()
302.       plot_results(gsr_fs, bay_fs)
303.
304.       print('====================== train dataset
      ======================')
305.       gsr_xs, gsr_fs, bay_xs, bay_fs = mini_batch_
      train()
306.       plot_results(gsr_fs, bay_fs)
307.
308.       gsr_xs, gsr_fs, bay_xs, bay_fs = mini_batch_
      train()
309.       plot_results(gsr_fs, bay_fs)
310.
```

```
311.       gsr_xs, gsr_fs, bay_xs, bay_fs = mini_batch_
   train()
312.       plot_results(gsr_fs, bay_fs)
313.
314.       gsr_xs, gsr_fs, bay_xs, bay_fs = adam_params
   _train()
315.       plot_results(gsr_fs, bay_fs)
316.
317.       gsr_xs, gsr_fs, bay_xs, bay_fs = epochs_trai
   n()
318.       plot_results(gsr_fs, bay_fs)
319.
320.
321.  def chooseParameters(f):
322.       pbounds = {
323.           'batch_size': (30, 30),
324.           'alpha': (0.001, 0.001),
325.           'beta1': (0.9, 0.9),
326.           'beta2': (0.999, 0.999),
327.           'epochs': (30, 30)
328.       }
329.       optimizer_1 = BayesianOptimization(
330.           f=f,
331.           pbounds=pbounds,
332.           random_state=42,
333.           verbose=1
334.       )
335.       optimizer_1.maximize(
336.           init_points=5,
337.           n_iter=20,
338.           acq='ei',
339.           xi=0.01,
340.           kappa=2.576,
341.           alpha=1e-5,
342.           n_restarts_optimizer=5,
343.           normalize_y=True
344.       )
345.       optimizer_2 = BayesianOptimization(
346.           f=f,
347.           pbounds=pbounds,
348.           random_state=42,
349.           verbose=1
350.       )
351.       optimizer_2.maximize(
```

```
        init_points=5,
        n_iter=20,
        acq='ei',
        xi=0.1,
        kappa=2.576,
        alpha=1e-5,
        n_restarts_optimizer=5,
        normalize_y=True
    )
    optimizer_3 = BayesianOptimization(
        f=f,
        pbounds=pbounds,
        random_state=42,
        verbose=1
    )
    optimizer_3.maximize(
        init_points=5,
        n_iter=20,
        acq='ei',
        xi=0.5,
        kappa=2.576,
        alpha=1e-5,
        n_restarts_optimizer=5,
        normalize_y=True
    )
    fs_1 = []
    fs_2 = []
    fs_3 = []
    for i, res in enumerate(optimizer_1.res):
        print("Iteration {}: \n\t{}".format(i, r
es))
        fs_1.append(res['target'])
    for i, res in enumerate(optimizer_2.res):
        print("Iteration {}: \n\t{}".format(i, r
es))
        fs_2.append(res['target'])
    for i, res in enumerate(optimizer_3.res):
        print("Iteration {}: \n\t{}".format(i, r
es))
        fs_3.append(res['target'])
```