Optimisation Algorithms Final

1 Design

This paper aims to contrast the performance of two gradient-free optimisation algorithms for selecting the hyperparameters of a convolution network. The performance of each algorithm will be measured by comparing them to each other as well as against a grid search baseline. This will enable conclusions not only on which algorithm is better, or how they differ from each other, but also on whether the computation expense is worth forgoing a simpler approach.

The experiment is run for a SGD with constant step size as well as an Adam optimiser. For the SGD, the choice of batch size and constant step size alpha is investigated. For Adam, the batch size, alpha, beta1 and beta2 values are investigated. The first algorithm used is a population search algorithm which generates values for the hyperparameters, and then iteratively verifies neighbours of the best parameters encountered so far. The second algorithm is a bayesian optimiser, which used a probabilistic model to choose the next set of hyperparameters investigated, based on the best parameters encountered so far. Thus, both algorithms iteratively improve upon a list of best parameters and attempt to converge, but the mechanism by which they select new values are different, making them interetsing to compare.

The population search algorithm was written by me based on indications received in class, and is similar to the version used in previous Optimisation Algorithms coursework. The Bayesian optimiser algorithm was written by Nogueira (14) and is available as a python package.

First, each optimiser is tested with varying sets of parameters to select the most competitive version. To keep a steady base for comparison, each algorithm will be allowed to make approximately 100 calls to the ML model. For policy evaluation, the numbers of neighbours and points kept on each iteration are tested, with the number of iterations adjusted to keep the same number of total calls to the model. For bayesian optimisation, the number of iterations and the number of initial random points are tested.

The performance of each algorithm is measured by comparing the test and training accuracy and cross-entropy loss of the best performing set of hyperparameters at each iteration of the algorithms. The data is balanced, with 10 classes of 6000 elements each (Angelov, Gegov, Jayne, and Shen, 2016). Therefore, the use of accuracy as a performance measure is sound. Cross-entropy loss is a common choice for classification tasks, measuring the difference between the predicted class probabilities and the real expected values.

Experiments run for the Week 8 Assignment of the ML course showed that performance can be improved quite a bit simply by increasing the size of the training set. However, this also increases the time and resources used on each iteration. Thus, it is interesting to see to what extent the optimisation algorithms are able to improve the performance of the ML model with a small, 5000 point training set, simply by identifying optimal hyperparameters. The number of of epochs will be kept to 20 throughout the experiment. This is because, following the experiments of the ML course, it became apparent that when running the model with 5000 data points, the increase in performance after 20 epochs is minimal and instead, over-fitting begins to set in.

2 Methodology

The population search algorithm was tuned by running a small number of iterations with various sets of parameters M and N corresponding to the sample size and neighbourhood size. Due to time and computation power constraints, the values tested are few and very small - $(M, N) \in 2, 3 * 1, 3$, and only 5 iterations are run for each pair. However, this experiment could generalise to testing out a larger variety of values. For example, trying different pairs of M and N between 1 and 20, as well as running at least 20 could give a better idea of the capabilities of population search. The experiment is run for both constant-step SGD and Adam

Figure 1 shows the results of the experiments on Adam. With the exception of the (3,1) pair, all other pairs have very similar performance both in terms of accuracy and loss. I will pick the pair (2,3) to continue the experiments, as it offers a good balance between computation cost and time, and performance. Its performance is very close to that of the highest accuracy and lowest loss pair - (3,3), but it requires fewer iterations and it seems so result in a slightly smaller gap between training performance and test performance - which is important in order to avoid over-fitting (or under-fitting).

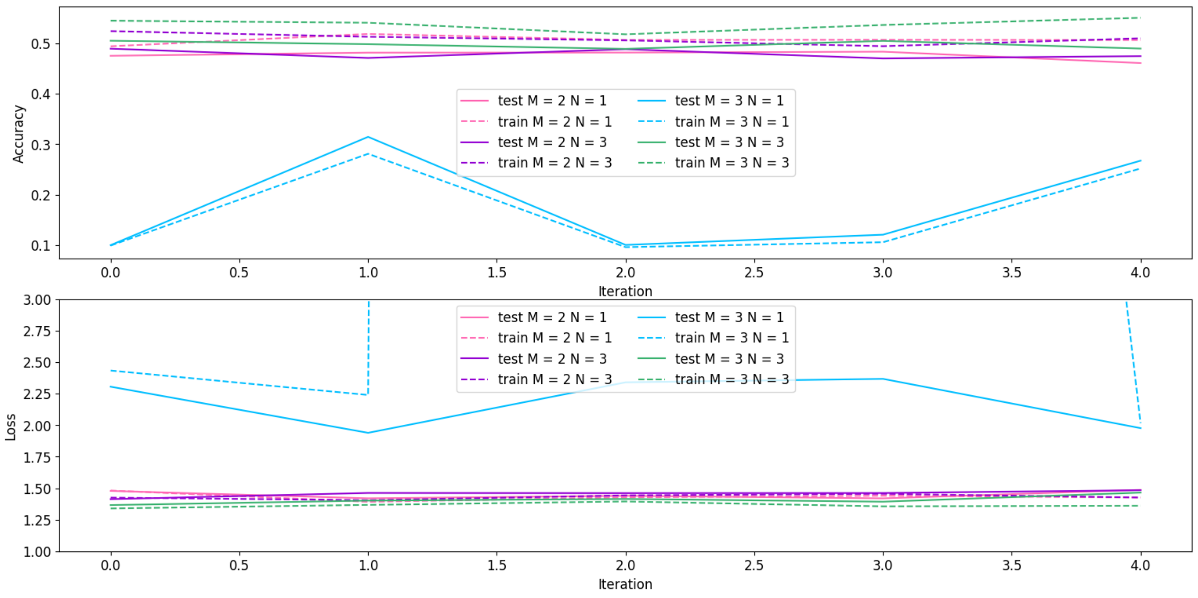

Figure 1: Tuning the population search algorithm for the Adam optimiser

Figure 2 shows the results of the tuning process on the SGD optimiser. Once again, I will select the pair (2,3) to continue the experiments, as it offers the best performance and satisfactory computation cost and time.

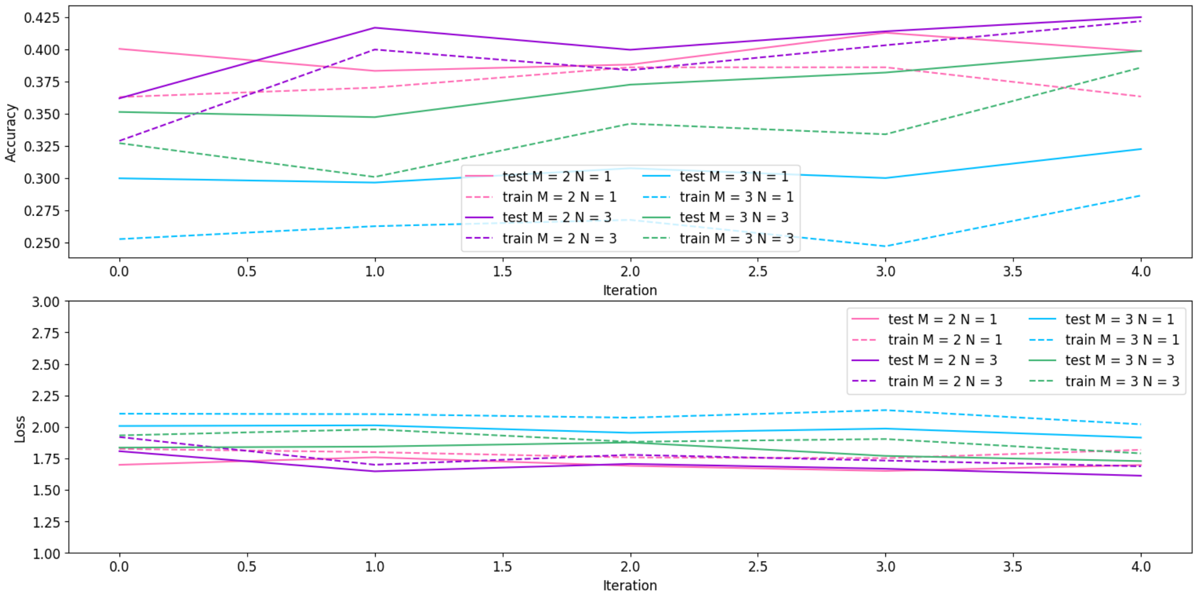

Figure 2: Tuning the population search algorithm for the SGD optimiser

The bayesian optimisation algorithm was tuned by running the algorithm with different pairs of values for the parameters B and R, corresponding to the number of bayesian optimisation steps B and the number of random exploration steps R. This experiment looks into the balance between exploitation (bayesian optimisation steps) and exploration (random steps). The algorithm was run for pairs $(B, R) \in 10, 20 * 5, 10$.

Figure 3 shows the results of tuning on Adam. On each iteration, the value corresponding to the set of parameters with the highest accuracy is given. The (B,R) = (20,10) run finds a good solution quickly which achieves high accuracy and low loss. The (B,R) = (20,5) run manages to eventually find a set of parameters which result in a slightly higher accuracy. However, the difference is larger for the training sets, with the test sets being almost equal. Moreover, The distance between the training and test accuracy of the blue run is larger, pointing to the possibility of more overfitting compared to the green run. Finally, in spite of blue achieving a slightly higher accuracy, green maintains the lowest loss. Thus, I will continue the Adam experiments with the Bayesian optimiser with 10 random steps and 20 Bayesian optimisation steps.

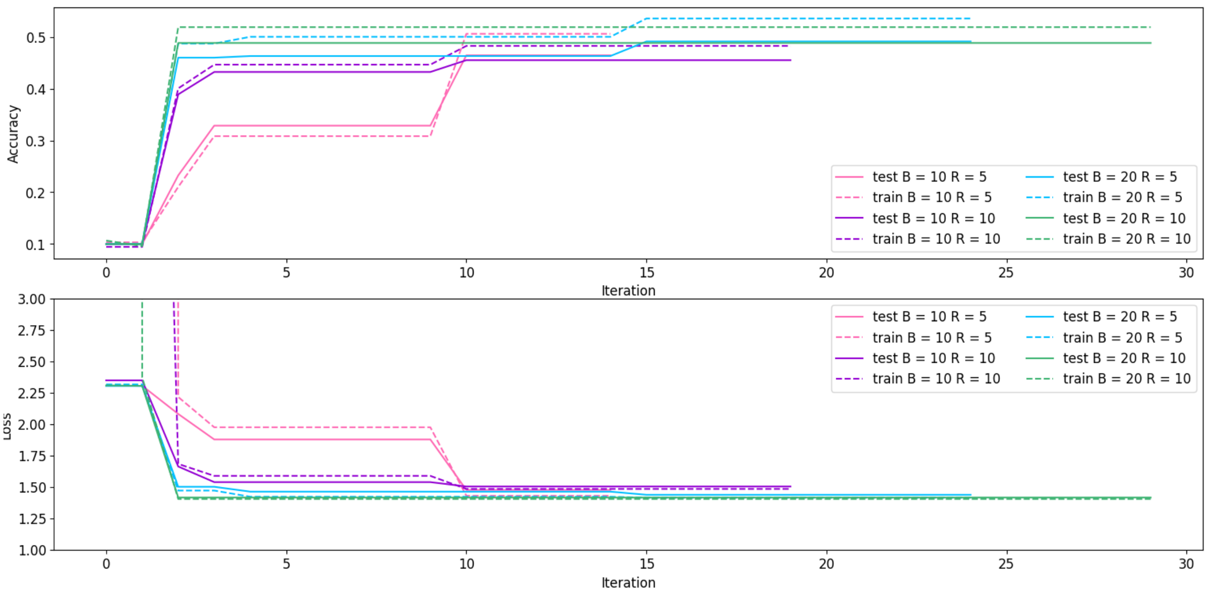

Figure 3: Tuning the Bayesian optimisation algorithm for the Adam optimiser

Figure 4 shows the same experiment as in figure 3, but this time for the constant step SGD. The run with (B,R) = (20,10) finds a set of parameters with a high accuracy quickly, and none of the other runs manage to outperform it. Thus, I will run the remaining SGD experiments using the Bayesian optimiser with 10 random steps and 20 Bayesian optimisation steps.

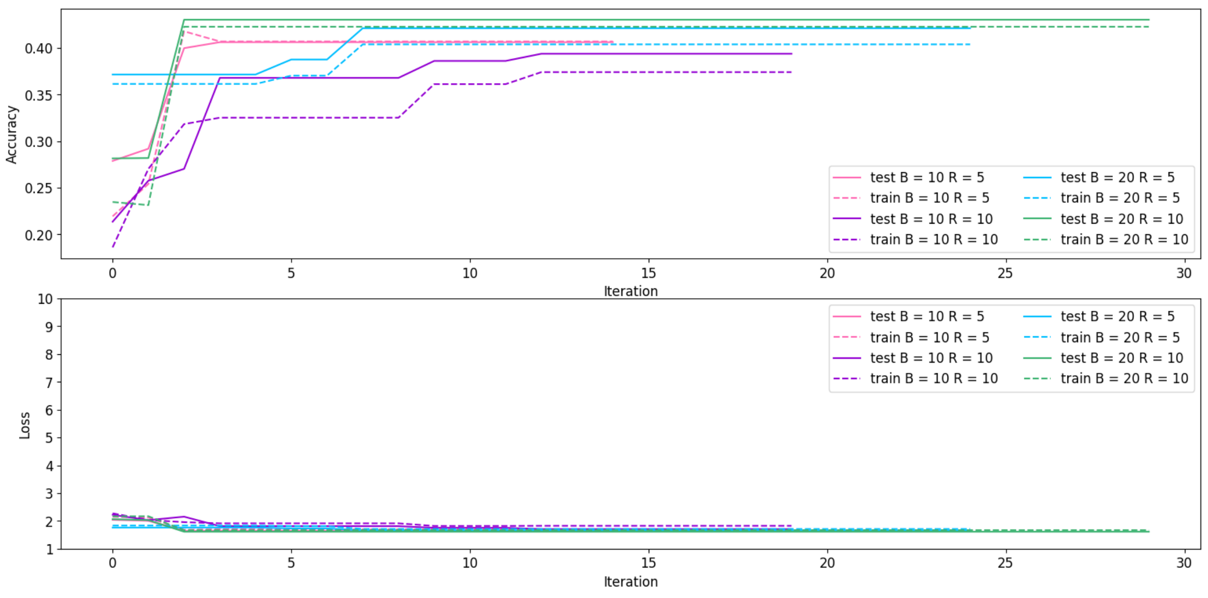

Figure 4: Tuning the Bayesian optimisation algorithm for the SGD optimiser

The population search algorithm will be run for 20 iterations, with M = 2 and N = 3. The Bayesian optimisation algorithm will be run for 30 iterations in total, 10 random points and 20 Bayesian optimisation steps. Both of the algorithms are given the same bounds for the generation of points. The batch size must be between 32 and 256, alpha must be between 0.001 and 0.05. For Adam, the same constraints hold and, in addition, beta1 and beta2 must be between 0.7 and 1. This gives a large enough pool to find optimal values, but also to see whether the algorithms are able to ignore non-optimal values and converge. The two algorithms are then tested against a baseline. The baseline is a grid-search which looks at 64 combinations of parameters for Adam and 4 combinations for SGD. The batch size is taken as 64 or 128, alpha is 0.001 or 0.01, beta1 is 0.8 or 0.9, and beta2 is 0.9 or 0.999. These values are common values used with SGD or Adam, and some are the default values of these algorithms. Thus, the two optimisers need to outperform the approach of using common or default values for hyperparameter selection. The ML model runs 20 epochs each time, as stated in the Design section. Throughout the experiment, the seed is set at 42 for replicability. This affects the randomised neighbours selected by population search, but not the mini-batch selection of the ML model. Thus, an element of variability exists which is best seen in the population search performance measurements.

3 Evaluation

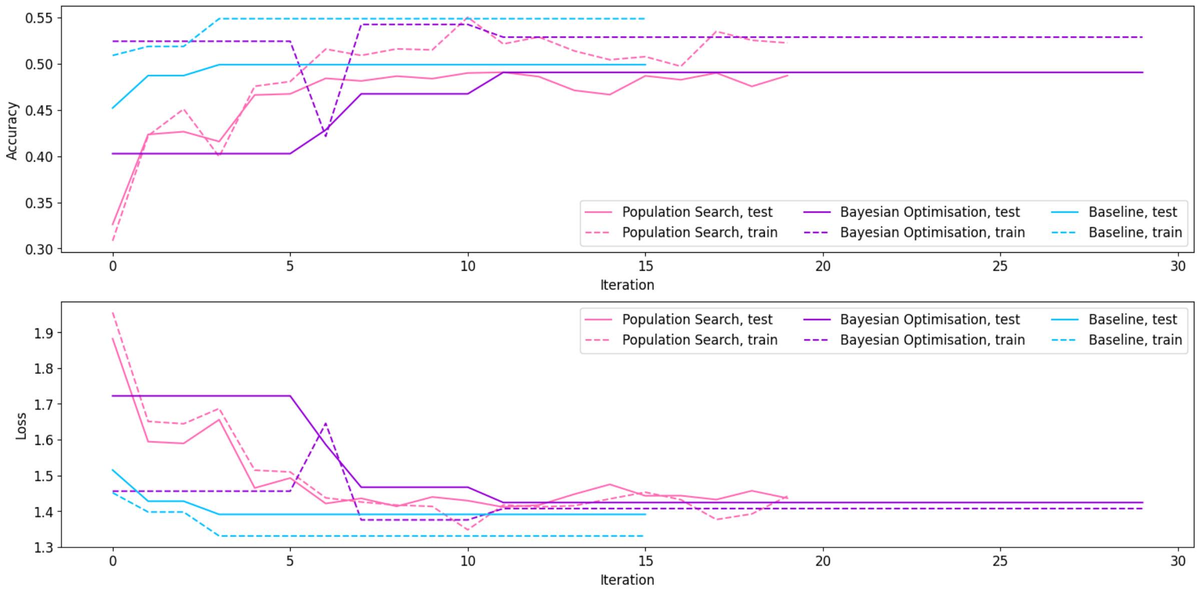

Figure 5: Comparison of the optimisers vs the baseline on Adam

Figure 5 shows and compares the population search optimiser, the Bayesian optimiser, and the grid search baseline. The solid lines correspond to the performance measured using the test data, and the dashed lines correspond to the performance measured on the test data. Each colour corresponds to an algorithm. By examining each pair of train and test data measures, some of the algorithms might be overfitting the data. In particular, the baseline achieves much better accuracy and loss on the training set than on the test set. Bayesian optimisation starts with a large gap, but actually ends up closing it somewhat as it begins to fit the model better. Population search begins with accuracy values that are very close, but they begin to diverge around the fourth iteration. However, their loss values do not diverge.

The baseline values are computed by taking pairs of parameters and computing the performance measures after fitting an ML model using the pair. Each time a pair of parameters achieves a new high in accuracy, the best performing pair is updated. By taking common values for an Adam optimiser, the baseline already begins high at 0.45 accuracy, and climbs up to a test accuracy of 0.499 corresponding to batch size 64, alpha 0.001, beta1 0.9, and beta2 0.999. Coincidentally, these values are the default values for the Adam optimiser. More interestingly, none of the optimisers manage to outperform this performance.

Bayesian optimisation is updated similarly to the baseline. Each time a new high in accuracy is found, all 4 performance measures are computed for that set of parameters. This means that the test accuracy will always be increasing, but the training set accuracy and the loss can fluctuate. In the end, the best set of parameters explored by the Bayesian optimiser is batch size = 104, alpha = 0.0049, beta1 = 0.76, beta2 =

0.9. This achieves an accuracy of 0.49.

Population search maintains a list of the best sets of parameters encountered so far. On each iteration, the performance value depicted in figure 5 corresponds to the best set of parameter available at that iteration. While the baseline and Bayesian optimiser do not recompute the function value of the same set of parameters multiple times, the population search algorithm computes every function value of the parameters in the neighbour set, including the current best. When a value is recomputed, the stochastic nature of the ML model causes slight variation on the performance of the same set of parameters tested multiple times. Because of this, the test accuracy of the population search algorithm can fluctuate slightly, showing the variance in performance of the ML model. Overall, the best accuracy achieved by population search is 0.491 using batch size 104, alpha 0.0042, beta1 0.86 and beta2 0.84. When selecting hyperparameters for Adam, population search seems to start off better than Bayesian optimisation, as it consistently outperforms it during the first 10 iterations. However, two factors make this comparison less impressive; first, the iteratons depicted in the figure to one call to the ML function, but instead to $M * (N + 1)$ calls made during that iteration while scoring each element in the sample. Thus, Bayesian optimisation likely begins to rival population search much sooner than it seems. Second, the first 10 iterations of Bayesian optimisation are random points sampled to explore the state space. As soon as the optimiser begins taking Bayesian steps, it immediately matches the population search performance both in terms of accuracy and loss.

It is noteworthy that both the Bayesian optimiser and the population search ended up with the same value for the optimal batch size - 104, and very similar alpha values around 0.0045. However, the population search is significantly more expensive to run than the Bayesian optimisation, making it an inferior choice for the Adam optimiser. However, none of the two optimisers managed to beat a baseline. This is especially surprising as not only is the basline very computationally affordable, but the values identified by it as optimal are the default Adam hyperparameter values. It is very likely that given more power, the optimisers might manage to outperform the default values. That being said, it is questionable whether such an improvement would be worth the computational expense, as the underwhelming results identified so far are already much more expensive than the baseline.

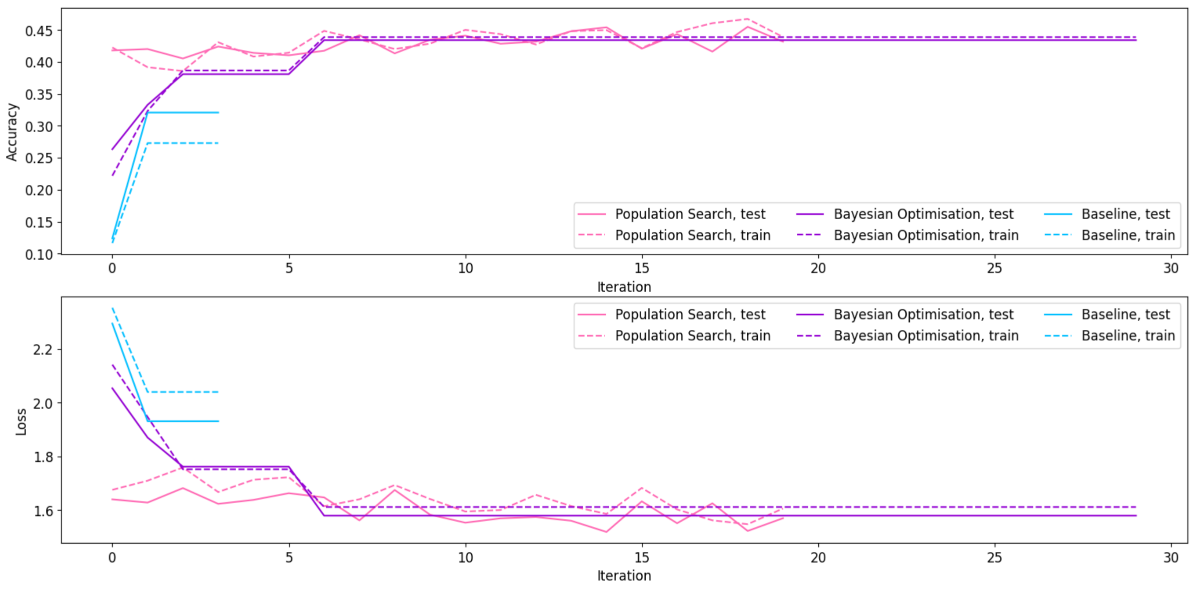

Figure 6: Comparison of the optimisers vs the baseline on the constant step size SGD

Figure 6 shows the performance of population search and the Bayesian optimiser against the baseline for the contant step size SGD. By examining each set of two lines of the same colour, none of the algorithms seem to overfit the data.

The baseline initially achieves low accuracy (0.1), and then the second pair achieves an improved accuracy of 0.32 that will remain the best result identified by the baseline. This result corresponds to a batch size of 64 and a step size alpha of 0.01.

The best accuracy achieved by the Bayesian optimiser is 0.434, corresponding to a batch size of 35 and a step size of 0.048. This is quite an improvement on the baseline, but a pattern begins to emerge - The batch size of 35 is close to the lower bound 32 of allowed batch sizes, while alpha = 0.048 is close to the alpha upper bound 0.05. Likewise, the best results identified by the baseline correspond to the lower option of batch size (64 out of 64 or 128) and the higher option of alpha (0.01 out of 0.001 or 0.01).

Population search achieves very similar performance to the Bayesian optimiser, with its maximum values fluctuating around those of the latter. The highest accuracy it ever achieves is 0.455 using a batch size of 33 and an alpha of 0.048. This is very close to the values identified by the Bayesian optimiser as well.

For the SGD model, both the Bayesian optimiser and the population search achieve not only the same performance, but find the same optimal hyperparameters. Both of them manage to outperform the baseline which was constrained to more average hyperparameter values, while the optimal values seem to be on the lower end for the batch size, and the higher end for the step size. However, in spite of ending up with very similar results, population search makes over 5 times as many calls to the ML function as the Bayesian optimiser. This could potentially be improved, for example by not making repeated calls for a set of parameters previously investigated and kept in the sample set, but this would only decrease the calls by an expected of M calls per iteration, as each new neighbour must be investigated anyway. Thus, given that it achieves the same performance and gives the same values at a fraction of the cost, the Bayesian optimiser outperforms both the baseline and the population search on the SGD model.

References

Angelov, P., A. Gegov, C. Jayne, and Q. Shen (2016). *Advances in Computational Intelligence Systems: Contributions Presented at the 16th UK Workshop on Computational Intelligence, September 7–9, 2016, Lancaster, UK*. Springer.

Nogueira, F. (2014–). Bayesian Optimization: Open source constrained global optimization tool for Python.

Appendix A - Code

```
import time
from math import *
import random
import numpy as np

import tensorflow as tf
from tensorflow import keras
from keras import layers, regularizers
from keras.optimizers import Adam, SGD
from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
from keras.layers import Conv2D, MaxPooling2D, LeakyReLU
from sklearn.metrics import accuracy_score, log_loss
from sklearn.utils import shuffle
from bayes_opt import BayesianOptimization
import matplotlib.pyplot as plt
from matplotlib import cm

import warnings

warnings.filterwarnings("ignore")

rng = random.Random()
# rng.seed(42)

accuracy_test_temp = []
accuracy_train_temp = []
loss_test_temp = []
loss_train_temp = []

A_test = []
A_train = []
L_test = []
L_train = []

accuracy_test = 0
loss_test = 0
accuracy_train = 0
loss_train = 0


def convolutions(x):
    global accuracy_test_temp
    global accuracy_train_temp
    global loss_test_temp
    global loss_train_temp
    if len(x) == 4:
        batch_size, alpha, beta1, beta2 = x
    if len(x) == 2:
        batch_size, alpha = x
    batch_size = int(batch_size)

    # Model / data parameters
    num_classes = 10
    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_data()
    n = 5000
    x_train = x_train[1:n];
    y_train = y_train[1:n]

    with tf.device('/device:GPU:0'):
        # Scale images to the [0, 1] range
        x_train = x_train.astype("float32") / 255
        x_test = x_test.astype("float32") / 255
        # print("orig x_train shape:", x_train.shape)

        # convert class vectors to binary class matrices
        y_train = keras.utils.to_categorical(y_train, num_classes)
        y_test = keras.utils.to_categorical(y_test, num_classes)

        use_saved_model = False
        if use_saved_model:
            model = keras.models.load_model("cifar.model")
        else:
            model = keras.Sequential()
            model.add(Conv2D(16, (3, 3), padding='same', input_shape=x_train.shape[1:],
            activation='relu'))
            model.add(Conv2D(16, (3, 3), strides=(2, 2), padding='same', activation='relu'))
            model.add(Conv2D(32, (3, 3), padding='same', activation='relu'))
            model.add(Conv2D(32, (3, 3), strides=(2, 2), padding='same', activation='relu'))
            model.add(Dropout(0.5))
            model.add(Flatten())
            model.add(Dense(num_classes, activation='softmax',
            kernel_regularizer=regularizers.l1(0.0001)))
            # model.compile(loss="categorical_crossentropy", optimizer='adam', metrics=["accuracy"])
            if len(x) == 4:
                model.compile(loss="categorical_crossentropy",
                              optimizer=Adam(learning_rate=alpha, beta_1=beta1, beta_2=beta2),
                              metrics=["accuracy"])
            if len(x) == 2:
                model.compile(loss="categorical_crossentropy", optimizer=SGD(learning_rate=alpha),
                metrics=["accuracy"])
            # model.summary()
            epochs = 20
            history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
            validation_split=0.1, verbose=0)
            model.save("cifar.model")
        preds = model.predict(x_test)
        y_pred = np.argmax(preds, axis=1)
        y_test1 = np.argmax(y_test, axis=1)
        accuracy_test = accuracy_score(y_test1, y_pred)
        loss_test = log_loss(y_test, preds)
        accuracy_train = history.history['accuracy'][-1]
        loss_train = history.history['loss'][-1]
        accuracy_test_temp.append(accuracy_test)
        accuracy_train_temp.append(accuracy_train)
        loss_test_temp.append(loss_test)
        loss_train_temp.append(loss_train)
    return accuracy_test


def population_search(l, u, iters=20, M=2, N=3, neighbourhood=10,
condition=50, function=convolutions):
    global accuracy_test_temp
    global accuracy_train_temp
    global loss_test_temp
    global loss_train_temp
    global A_test
    global A_train
    global L_test
    global L_train
    A_test = []
    A_train = []
    L_test = []
    L_train = []
    n = len(l)
    x_sample = [[rng.uniform(l[i], u[i]) for i in range(n)] for j in range(N)]
    X = []
    Y = []
    f_prev = 100
    change = 0
    for k in range(iters):
        accuracy_test_temp = []
        accuracy_train_temp = []
        loss_test_temp = []
        loss_train_temp = []
        x_neighbourhood = x_sample.copy()
        for x in x_sample:
            for m in range(N):
                neighbour = [x[i] + rng.uniform(-(u[i] - l[i]) / neighbourhood,
                (u[i] - l[i]) / neighbourhood) for i in
                             range(n)]
                if l[0] < neighbour[0] < u[0] and l[1] < neighbour[1] < u[1]:
                    x_neighbourhood.append(neighbour)
        x_dict = {}
        for x in x_neighbourhood:
            x_dict[tuple(x)] = function(x)
        positions = {}
        for k, key in enumerate(list(x_dict.keys())):
            positions[key] = k
        x_dict = {k: v for k, v in sorted(x_dict.items(), key=lambda item: item[1])}
        x_sample = list(x_dict.keys())[-M:]
        A_test.append(accuracy_test_temp[positions[x_sample[-1]]])
        A_train.append(accuracy_train_temp[positions[x_sample[-1]]])
        L_test.append(loss_test_temp[positions[x_sample[-1]]])
        L_train.append(loss_train_temp[positions[x_sample[-1]]])
        f_x = list(x_dict.values())[-1]
        X.append(x_sample[-1])
        # Y.append(f_x)
        if f_prev == f_x:
            change = change + 1
        else:
            f_prev = f_x
            change = 0
        if change > condition:
            break
    return X


def population_search_tuning():
    global A_test
    global A_train
    global L_test
    global L_train

    plt.rcParams['font.size'] = 12
    fig, ax = plt.subplots(2)
    colours = ['hotpink', 'darkviolet', 'deepskyblue', 'mediumseagreen', 'goldenrod']
    i = 0
    for M in [2, 3]:
        print('M ' + str(M))
        for N in [1, 3]:
            print('N ' + str(N))
            population_search([32, 0.001], [256, 0.05], iters=5, M=M, N=N, condition=5,
            function=convolutions)
            print(A_test)
            print(A_train)
            print(L_test)
            print(L_train)
            ax[0].plot(A_test, c=colours[i], linestyle='-', label=('test M = ' + str(M)
            + ' N = ' + str(N)))
            ax[0].plot(A_train, c=colours[i], linestyle='--', label=('train M = ' + str(M)
            + ' N = ' + str(N)))
            ax[1].plot(L_test, c=colours[i], linestyle='-', label=('test M = ' + str(M)
            + ' N = ' + str(N)))
            ax[1].plot(L_train, c=colours[i], linestyle='--', label=('train M = ' + str(M)
            + ' N = ' + str(N)))
            i += 1
    # ax.set(ylim=(0, 1))
    ax[0].legend(ncol=2)
    ax[1].legend(ncol=2)
    ax[0].set_xlabel('Iteration')
    ax[0].set_ylabel('Accuracy')
    ax[1].set_xlabel('Iteration')
    ax[1].set_ylabel('Loss')
    plt.show()


def bayesian_optimisation(l, u, iters=20, random=10, function=convolutions):
    global accuracy_test_temp
    global accuracy_train_temp
    global loss_test_temp
    global loss_train_temp
    global A_test
    global A_train
    global L_test
    global L_train
    accuracy_test_temp = []
    accuracy_train_temp = []
    loss_test_temp = []
    loss_train_temp = []

    def convolutions_bayes(batch_size, alpha, beta1=0, beta2=0):
        x = [batch_size, alpha, beta1, beta2] if beta1 > 0 else [batch_size, alpha]
        return function(x)

    bounds = {'batch_size': (l[0], u[0]), 'alpha': (l[1], u[1]), 'beta1': (l[2], u[2]),
    'beta2': (l[3], u[3])} if len(
        l) == 4 else {'batch_size': (l[0], u[0]), 'alpha': (l[1], u[1])}
    bayes = BayesianOptimization(f=convolutions_bayes, pbounds=bounds, verbose=1)
    bayes.maximize(n_iter=iters, init_points=random)
    best_params = bayes.max['params']
    best_accuracy = bayes.max['target']

    best_acc = 0
    best_a = 0
    for a, acc in enumerate(accuracy_test_temp):
        if acc > best_acc:
            best_acc = acc
            best_a = a
        A_test.append(best_acc)
        A_train.append(accuracy_train_temp[best_a])
        L_test.append(loss_test_temp[best_a])
        L_train.append(loss_train_temp[best_a])
    return best_accuracy, best_params


def bayesian_optimisation_tuning():
    global A_test
    global A_train
    global L_test
    global L_train

    plt.rcParams['font.size'] = 12
    fig, ax = plt.subplots(2)
    colours = ['hotpink', 'darkviolet', 'deepskyblue', 'mediumseagreen', 'goldenrod']
    i = 0
    for B in [20]:
        print('B ' + str(B))
        for R in [5, 10]:
            print('R ' + str(R))
            bayesian_optimisation([32, 0.001, 0.8, 0.8], [256, 0.05, 1, 1], B, R,
            function=convolutions)
            print(accuracy_test_temp)
            print(accuracy_train_temp)
            print(loss_test_temp)
            print(loss_train_temp)
            print(A_test)
            print(A_train)
            print(L_test)
            print(L_train)
            ax[0].plot(A_test, c=colours[i], linestyle='-',
            label=('test B = ' + str(B) + ' R = ' + str(R)))
            ax[0].plot(A_train, c=colours[i], linestyle='--',
            label=('train B = ' + str(B) + ' R = ' + str(R)))
            ax[1].plot(L_test, c=colours[i], linestyle='-',
            label=('test B = ' + str(B) + ' R = ' + str(R)))
            ax[1].plot(L_train, c=colours[i], linestyle='--',
            label=('train B = ' + str(B) + ' R = ' + str(R)))
            i += 1
    # ax.set(ylim=(0, 1))
    ax[0].legend(ncol=2)
    ax[1].legend(ncol=2)
    ax[0].set_xlabel('Iteration')
    ax[0].set_ylabel('Accuracy')
    ax[1].set_xlabel('Iteration')
    ax[1].set_ylabel('Loss')
    plt.show()


def baseline_Adam():
    global accuracy_test_temp
    global accuracy_train_temp
    global loss_test_temp
    global loss_train_temp
    accuracy_test_temp = []
    accuracy_train_temp = []
    loss_test_temp = []
    loss_train_temp = []
    best_params = []
    best_accuracy = 0
    accuracy_history = []
    best_history = []
    for batch_size in [64, 128]:
        for alpha in [0.001, 0.01]:
            for beta1 in [0.8, 0.9]:
                for beta2 in [0.9, 0.999]:
                    params = [batch_size, alpha, beta1, beta2]
                    accuracy = convolutions(params)
                    accuracy_history.append(accuracy)
                    if accuracy > best_accuracy:
                        best_history.append(accuracy)
                        best_accuracy = accuracy
                        best_params = params
    best_acc = 0
    best_a = 0
    for a, acc in enumerate(accuracy_test_temp):
        if acc > best_acc:
            best_acc = acc
            best_a = a
        A_test.append(best_acc)
        A_train.append(accuracy_train_temp[best_a])
        L_test.append(loss_test_temp[best_a])
        L_train.append(loss_train_temp[best_a])
    return best_accuracy, best_params, accuracy_history, best_history


def baseline_constant():
    global accuracy_test_temp
    global accuracy_train_temp
    global loss_test_temp
    global loss_train_temp
    accuracy_test_temp = []
    accuracy_train_temp = []
    loss_test_temp = []
    loss_train_temp = []
    best_params = []
    best_accuracy = 0
    accuracy_history = []
    best_history = []
    for batch_size in [64, 128]:
        for alpha in [0.001, 0.01]:
            params = [batch_size, alpha]
            accuracy = convolutions(params)
            accuracy_history.append(accuracy)
            if accuracy > best_accuracy:
                best_history.append(accuracy)
                best_accuracy = accuracy
                best_params = params
    best_acc = 0
    best_a = 0
    for a, acc in enumerate(accuracy_test_temp):
        if acc > best_acc:
            best_acc = acc
            best_a = a
        A_test.append(best_acc)
        A_train.append(accuracy_train_temp[best_a])
        L_test.append(loss_test_temp[best_a])
        L_train.append(loss_train_temp[best_a])
    return best_accuracy, best_params, accuracy_history, best_history


def baseline(l, u):
    if len(l) == 2:
        return baseline_constant()
    if len(l) == 4:
        return baseline_Adam()


def compare(l, u):
    global A_test
    global A_train
    global L_test
    global L_train
    plt.rcParams['font.size'] = 12
    fig, ax = plt.subplots(2)
    colours = ['hotpink', 'darkviolet', 'deepskyblue', 'mediumseagreen', 'goldenrod']
    labels = ['Population Search, ', 'Bayesian Optimisation, ', 'Baseline, ']
    for f, function in enumerate([population_search, bayesian_optimisation, baseline]):
        A_test = []
        A_train = []
        L_test = []
        L_train = []
        print(function(l, u))
        print(A_test)
        print(A_train)
        print(L_test)
        print(L_train)
        ax[0].plot(A_test, c=colours[f], linestyle='-', label=labels[f] + 'test')
        ax[0].plot(A_train, c=colours[f], linestyle='--', label=labels[f] + 'train')
        ax[1].plot(L_test, c=colours[f], linestyle='-', label=labels[f] + 'test')
        ax[1].plot(L_train, c=colours[f], linestyle='--', label=labels[f] + 'train')

    #ax[1].set(ylim=(1, 10))
    ax[0].legend(ncol=3)
    ax[1].legend(ncol=3)
    ax[0].set_xlabel('Iteration')
    ax[0].set_ylabel('Accuracy')
    ax[1].set_xlabel('Iteration')
    ax[1].set_ylabel('Loss')
    plt.show()
```

```
#bayesian_optimisation_tuning()
# population_search_tuning()
#compare([32, 0.001],[256, 0.05])
#compare([32, 0.001,0.7,0.8],[256, 0.05,1,1])
```