Optimisation Algorithms - Week 8 Assignment

Ernests Kuznecovs - 17332791 - kuznecoe@tcd.ie

21st March

Contents

1 (a) Global Random Search

1.1 (i) Global Random Search

Global random search is defined with input arguments:

- 'intervals' has the type [(float, float)],
  - ith element of the list corresponds to the ith parameter of the function we are optimising.
  - First value of tuple is the minimum value the parameter can take.
  - Second value of tuple is the maximum value the parameter can take.
- 'N' has the type int
  - It's number of times to sample the parameters and run the function with those parameters.
- 'f' has the type function of arity len(intervals)
  - The function that takes in len(intervals) parameters and returns a scalar value, this is the function we are trying to find the minimum value for.

Inside our function, we keep a variable 'lowest' that keep track of what the lowest function value was and the corresponding parameters that achieved the lowest value. Each iteration (N max iterations) we randomly sample parameters for our function within the intervals we specified, then apply those parameters to the function and see if we get a new lowest value.

```python
def global_random_search(intervals, N, f):

    # lowest :: (val, [float])
    # fst is the lowest function value achieved
    # snd is the list of parameter values
    lowest = None

    # unzip list of tuples
    l = [l for l, u in intervals]
    u = [u for l, u in intervals]

    # sample and run N times
    for s in range(N):
        r = np.random.uniform(l, u)
        v = f(r)
        if (not lowest) or lowest[0] > v:
            lowest = (v.copy(), r.copy())
    return lowest
```

1.2 (ii) Global Random Search on f_1 and f_2

- $f_1(x_1, x_2) = 3\,(x_1 - 9)^4 + 5\,(x_2 - 9)^2$
- $f_2(x, y) = 5\,|y - 9| + \max\,(0, x - 9)$

For evaluating function value vs execution time it will be difficult to measure as GRS has quite a lot of randomness. The result changes from run to run on GRS, while for GD it doesn't. It's hard to measure them together because performance is not even comparable for different values of x_0, α, intervals, the hyperparameters are completely different. The nature of the algorithm is completely different.

α and x_0 will be kept the same for both function on GD because the intervals will also be kept the same for GRS.

The number of evaluations on GD is $j * i$, where j is the number of parameters the function to optimise takes, and i is the number of iterations. The number of evaluations for GRS is N, the number of times to sample and evaluate the function.

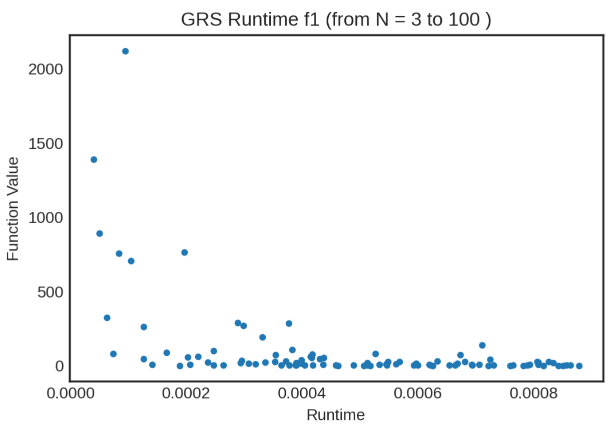

Figure 1

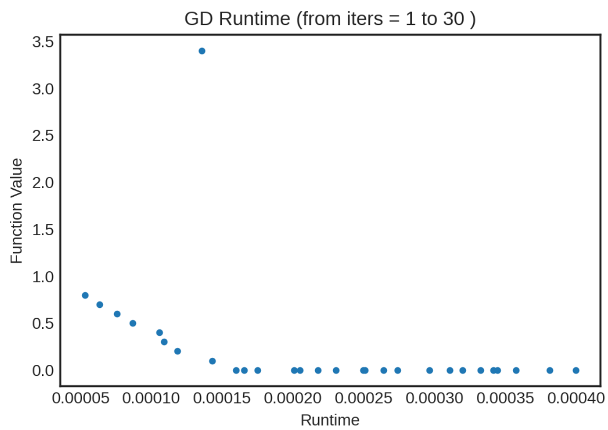

Figure 2

1.2.1 f_1

On f_1 global random search can handle the steep nature of it, as the slope has no effect algorithm. Wheras on GD, it is very sensitive to inital x, and cause numerical errors. Though the chance of GRS landing on the minimum can be slim if intervals are too large, f_1 has a relatively small area where the minimum lies at a scale of -10 to 10.

We can see that GD, fig 2 follows a curve, while GRS 1 can get a low number with lower runtime, but there is more variance the less iterations.

1.2.2 f_2

On f_2, because there the minimum is not so narrow on a -10,10 scale, GRS can land at function value of 0.5 with low amount of evaluatoins. GD is very well behaved on this function since it is concave-like, and reaches the minmum a lot in a lot faster than 100 iterations.

2 (b) Population Based Sampling

2.1 (i) Population Based Samping

Two version are implemented, the first one with that doesn't grow eponentially in runtime, and one that does. The non exponential one will be presented. Exponential one can be found in the appendex of code under the function name grs2.

2.1.1 Population Based Sampling

Population bases sampling chooses N random points, takes the top M of them, and then for each of those M points, N points are sampled within the neighbourhood, and the top points are taken out of those, and the process is repeated. N random points are chosen in a region of $(\frac{1}{M*\epsilon})^c$ of the original interval, where c is the depth of iteration, ϵ is a hyperparameter to scale size of the neighbourhood, and M is the number of top points selected.

- If we assume, that the best M points are evenly placed across the interval, then having 1/M of the size of the interval will mean that the sum of the sub intervals will span the range of the whole interval.

In the code, the algorithm first samples N points, top M are taken and this initiates the loop for a depth of c. From the parameter values calculated for an M, new intervals are centered around the parameter value, decreased and scaled with original range the parameter was initially, and each iteration reduces the neighbourhood. This algorithm does not throw away the M points when the new N are computed, the M points are included in picking the next top M.

```
def take_top(M, Nresults):
    # Nresults :: [(float, [float])]
    Nresults.sort(key=(lambda x : x[0]))
    return Nresults[0:M].copy()

# each param has a new interval centered around param value
# interval are centered around params
def get_new_intervals_2(params, intervals, M, c):
    # new_intervals :: [(float, float)]
    new_intervals = []
    for i, param_val in enumerate(params):
        l, u = intervals[i]
        interval_range = (u - l)
        offset = ((1/(M**c)) * interval_range) / 2
        new_l = np.clip(param_val-offset, l, u)
        new_h = np.clip(param_val+offset, l, u)
        new_intervals += [(new_l, new_h)]
    return new_intervals

def unzip_intervals(intervals):
    l = [l for l, u in intervals]
    u = [u for l, u in intervals]
```

```python
    return l,u

def grs3(intervals, N, M, f, c, eps=1):
    # intervals :: [(l, u)]

    # Nresults :: [(float, [float])]
    # fst is the lowest function value achieved
    # snd is the list of parameter values
    Nresults = []
    l,u = unzip_intervals(intervals)
    for s in range(N):
        r = np.random.uniform(l, u) ; v = f(r)
        Nresults += [(v.copy(), r.copy())]
    # topM :: [(float, [float])]
    topM = take_top(M, Nresults)

    for i in range(c):
        Nresults = []
        for _, param_values in topM:
            l,u = unzip_intervals(get_new_intervals_2(param_values, intervals, M*eps,
     i+1))
            for _ in range(N):
                r = np.random.uniform(l, u) ; v = f(r)
                Nresults += [(v.copy(), r.copy())]
        Nresults += topM
        topM = take_top(M, Nresults)
    return take_top(1, topM)[0]
```

2.2 (ii) Population Based Sampling Search on f_1 and f_2

The number of evaluations of PBS is N + (N * M * c). N samples on before the loop to get initial M, then inner loop does c iterations where N number of points for each M is taken and evaluated.

GRS and Population Based Sampling (referred in code as GRS3) are tested against each other, parameters are picked such that their evaluations are the same.

2.2.1 f_1

With unsepcialised parameters on PBS, it does not perform any better than GRS.

```python
intervals = [(-10, 10), (-10, 10)]
testGRS3(intervals, N=25, M=2, f=f1, c=2, eps=1, runs=1000)
```

```
1000 runs of GRS3
Number of f evals: 125
Standard deviation on final function values:  37.52454415835108
Mean on final function values:  5.636444145103109
```

```python
intervals = [(-10, 10), (-10, 10)]
testGRS(intervals, N=125, f=f1, runs=1000)
```

```
Number of f evals: 125
1000 runs of GRS
Standard deviation on final function values:  15.119353680012136
Mean on final function values:  7.6971497673503855
```

PBS can be made to pull ahead of GRS significantly at 100 evaluations by choosing M low and N=1 with c=3 and by bumping up the rate at which region narrows (eps=1.5). These parameters give the PBS behaviour of rapidly narrowing into a single minimum.

```python
intervals = [(-10, 10), (-10, 10)]
testGRS3(intervals, N=25, M=1, f=f1, c=3, eps=1.5, runs=1000)
```

```
1000 runs of GRS3
Number of f evals: 100
Standard deviation on final function values:  2.949378279695238
Mean on final function values:  0.545379791915719
```

```
intervals = [(-10, 10), (-10, 10)]
testGRS(intervals, N=100, f=f1, runs=1000)
```

```
Number of f evals: 100
1000 runs of GRS
Standard deviation on final function values:  24.52660098010104
Mean on final function values:  11.890304478212249
```

2.2.2 f_2

PBS can be made to pull ahead of GRS at 40 evaluations quite significanlty, using a similar rapid narrowing configuraiton.

```
intervals = [(-10, 10), (-10, 10)]
testGRS3(intervals, N=10, M=1, f=f2, c=3, eps=2, runs=1000)
```

```
1000 runs of GRS3
Number of f evals: 40
Standard deviation on final function values:  0.5220113548425356
Mean on final function values:  0.30859770356078464
```

```
intervals = [(-10, 10), (-10, 10)]
testGRS(intervals, N=40, f=f2, runs=1000)
```

```
Number of f evals: 40
1000 runs of GRS
Standard deviation on final function values:  1.1866724491447593
Mean on final function values:  1.2623655712980348
```

3 (c) Global Random Search to Choose Hyperparameters on Conv Net

Applying random search to choose hyperparameters for conv net. Hyperparams are:

- Mini-batch size: b
- Adam parameters: α, β_1, β_2
- Number of epochs: epochs

Would be good to discretise the ranges so that there is a smaller space to search.

```
def testParams(alpha, beta1, beta2, batch_size, epochs):
    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)))

    adam = tf.keras.optimizers.Adam(learning_rate=alpha, beta_1=beta1, beta_2=beta2,
   name='Adam')
```

```
    model.compile(loss="categorical_crossentropy", optimizer=adam, metrics=["accuracy"])

    model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)
    preds = model.predict(x_test)
    cce = tf.keras.losses.CategoricalCrossentropy()
    val = cce(y_test, preds).numpy()
    return val
t = lambda x: testParams(alpha=x[0], beta1=x[1], beta2=x[2], batch_size=x[3], epochs=x[4])
```

```
intervals = [
    (0.001, 0.01),              # alpha
    (0.5, 0.999),               # beta1
    (0.5, 0.999),               # beta2
    (4, 256),               # batch_size
    (5, 30),               # epochs
]
```

```
v = grs3(intervals, 20, 1, t, c=3, eps=1.5)
```

```
print(v)
```

```
(1.348901, array([6.09204202e-03, 7.55942386e-01, 7.73003179e-01, 2.49500024e+02,
       2.07511524e+01]))
```

With 80 evaluations (just under 1 hour of training on CPU), PBS picked: alpha=0.006, beta1=0.75, beta2=0.77, batchsize=250, epochs=21

```
v = global_random_search(intervals, 80, t)
```

```
print(v)
```

```
(1.3314114, array([3.76845908e-03, 8.97153808e-01, 7.73088738e-01, 1.25796853e+02,
       2.81799326e+01]))
```

With 80 evaluations GRS picked: alpha=0.003, beta1=0.89, beta2=0.77, batchsize=125, epochs=28.

GRS achieved slighly lower cross entorpy loss. The random nature and temporally expensive procedures are an unfortunate combination for picking hyperparameters.

4 Appendix

4.1 Code Listing

```
import matplotlib as mpl
mpl.rcParams['figure.dpi'] = 200
mpl.rcParams['figure.facecolor'] = '1'
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')
import copy
import numpy as np
from sklearn import metrics

from OptimisationAlgorithmToolkit import Algorithms
from OptimisationAlgorithmToolkit import DataType
from OptimisationAlgorithmToolkit import Plotting
from OptimisationAlgorithmToolkit import Function
import importlib
importlib.reload(Function)
importlib.reload(Algorithms)
importlib.reload(DataType)
importlib.reload(Plotting)
from OptimisationAlgorithmToolkit.Function import BatchedFunction, SymbolicFunction
```

```python
from OptimisationAlgorithmToolkit.Algorithms import ConstantStep, Polyak, RMSProp,
    HeavyBall, Adam
from OptimisationAlgorithmToolkit.DataType import create_labels, get_titles
from OptimisationAlgorithmToolkit.Plotting import ploty, plot_contour, plot_path,
    plot_step_size

from time import perf_counter

import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, regularizers
from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
from keras.layers import Conv2D, MaxPooling2D, LeakyReLU
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.utils import shuffle
import matplotlib.pyplot as plt
plt.rc('font', size=18)
plt.rcParams['figure.constrained_layout.use'] = True
import sys

# 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]
#x_test=x_test[1:500]; y_test=y_test[1:500]

# 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
    "])
  model.summary()

  batch_size = 128
  epochs = 20
  history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
    validation_split=0.1)
  model.save("cifar.model")
  plt.subplot(211)
  plt.plot(history.history['accuracy'])
  plt.plot(history.history['val_accuracy'])
  plt.title('model accuracy')
  plt.ylabel('accuracy')
```

```
    plt.xlabel('epoch')
    plt.legend(['train', 'val'], loc='upper left')
    plt.subplot(212)
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('model loss')
    plt.ylabel('loss'); plt.xlabel('epoch')
    plt.legend(['train', 'val'], loc='upper left')
    plt.show()

preds = model.predict(x_train)
y_pred = np.argmax(preds, axis=1)
y_train1 = np.argmax(y_train, axis=1)
print(classification_report(y_train1, y_pred))
print(confusion_matrix(y_train1,y_pred))

preds = model.predict(x_test)
y_pred = np.argmax(preds, axis=1)
y_test1 = np.argmax(y_test, axis=1)
print(classification_report(y_test1, y_pred))
print(confusion_matrix(y_test1,y_pred))

from sympy import symbols, Max, Abs
x1, x2 = symbols('x1 x2', real=True)

sym_f1 = 3 * (x1-9)**4 + 5 * (x2-9)**2
f1 = SymbolicFunction(sym_f1, [x1, x2], "f_1").function_list_arg
f1o = SymbolicFunction(sym_f1, [x1, x2], "f_1")

sym_f2 = Max(x1-9 ,0) + 5 * Abs(x2-9)
f2 = SymbolicFunction(sym_f2, [x1, x2], "f_2").function_list_arg
f2o = SymbolicFunction(sym_f2, [x1, x2], "f_2")

def munzip(ll):
    l = [l for l, u in ll]
    u = [u for l, u in ll]
    return l,u

def myt(lam):
    ts = []
    r1 = lam()
    for i in range(50):
        t1 = perf_counter(); lam(); t2 = perf_counter()
        ts += [t2-t1]
    return (sum(ts)/len(ts), r1)

x0 = np.array([10, 10])
alpha = 0.1
f = f1o
iters=100

# print("Final f:", o[0]['Y'][-1])
# print("Final xs:", o[0]['X'][-1])

gdif = lambda i, f: ConstantStep.set_parameters(x0=x0, alpha=alpha, f=f, iters=i).run
    ()[0]['Y'][-1]

i = list(range(1,30))
p1 = []
for ii in i:
    p1 += [myt(lambda: gdif(ii, f2o))]

x,y = munzip(p1)
plt.scatter(x,y, s=10)

plt.xlabel("Runtime")
plt.ylabel("Function Value")
plt.title("GD Runtime (from iters = 1 to 30 )")
```

```
intervals = [(-10, 10), (-10, 10)]

grs = lambda N, f: global_random_search(intervals, N, f)[0]

n = list(range(3,100))
p2 = []
for N in n:
    p2 += [myt(lambda: grs(N, f2))]

# x,y = munzip(p1)
# plt.scatter(x,y, s=10, marker='^')
x,y = munzip(p2)
plt.scatter(x,y, s=10, marker='o')

plt.xlabel("Runtime")
plt.ylabel("Function Value")
plt.title("GRS Runtime f1 (from N = 3 to 100 )")

# x,y = munzip(p1)
# plt.scatter(x,y, s=10, marker='^')
x,y = munzip(p2)
plt.scatter(x,y, s=10, marker='o')

plt.xlabel("Runtime")
plt.ylabel("Function Value")
plt.title("GRS Runtime f2 (from N = 3 to 100 )")

gdif = lambda i, f: ConstantStep.set_parameters(x0=x0, alpha=alpha, f=f, iters=i).run
    ()[0]['Y'][-1]

def global_random_search(intervals, N, f):

    # lowest :: (val, [float])
    # fst is the lowest function value achieved
    # snd is the list of parameter values
    lowest = None

    # unzip list of tuples
    l = [l for l, u in intervals]
    u = [u for l, u in intervals]

    # sample and run N times
    for s in range(N):
        r = np.random.uniform(l, u)
        v = f(r)
        if (not lowest) or lowest[0] > v:
            lowest = (v.copy(), r.copy())
    return lowest

a = [1, 2, 3]
b = [4, 5, 6]
c = np.random.uniform(a, b)
print(c)

def testGRS(intervals, N, f, runs):
    r = []
    for i in range(runs):
        r += [global_random_search(intervals, N, f)[0]]

    print("Number of f evals:", N)
    print(runs, "runs of GRS")
    print("Standard deviation on final function values: ", np.std(r))
    print("Mean on final function values: ", np.mean(r))

def take_top(M, Nresults):
    # Nresults :: [(float, [float])]
    Nresults.sort(key=(lambda x : x[0]))
```

```
    return Nresults[0:M].copy()

# each param has a new interval centered around param value
# interval are centered around params
def get_new_intervals_2(params, intervals, M, c):
    # new_intervals :: [(float, float)]
    new_intervals = []
    for i, param_val in enumerate(params):
        l, u = intervals[i]
        interval_range = (u - l)
        offset = ((1/(M**c)) * interval_range) / 2
        new_l = np.clip(param_val-offset, l, u)
        new_h = np.clip(param_val+offset, l, u)
        new_intervals += [(new_l, new_h)]
    return new_intervals

def unzip_intervals(intervals):
    l = [l for l, u in intervals]
    u = [u for l, u in intervals]
    return l,u

def grs3(intervals, N, M, f, c, eps=1):
    # intervals :: [(l, u)]

    # Nresults :: [(float, [float])]
    # fst is the lowest function value achieved
    # snd is the list of parameter values
    Nresults = []
    l,u = unzip_intervals(intervals)
    for s in range(N):
        r = np.random.uniform(l, u) ; v = f(r)
        Nresults += [(v.copy(), r.copy())]
    # topM :: [(float, [float])]
    topM = take_top(M, Nresults)

    for i in range(c):
        Nresults = []
        for _, param_values in topM:
            l,u = unzip_intervals(get_new_intervals_2(param_values, intervals, M*eps, i+1))
            for _ in range(N):
                r = np.random.uniform(l, u) ; v = f(r)
                Nresults += [(v.copy(), r.copy())]
        Nresults += topM
        topM = take_top(M, Nresults)
    return take_top(1, topM)[0]

# each param has a new interval centered around param value
# interval will be at a range of 1/M
def get_new_intervals(params, intervals, M):
    # new_intervals :: [(float, float)]
    new_intervals = []
    for i, param_val in enumerate(params):
        l, u = intervals[i]
        interval_range = (u - l)
        offset = ((1/M) * interval_range) / 2
        new_l = np.clip(param_val-offset, l, u)
        new_h = np.clip(param_val+offset, l, u)
        new_intervals += [(new_l, new_h)]
    return new_intervals

# global_random_search_2 returns (float, [float])
# fst is the lowest function value achieved
# snd is the list of parameter values
def grs2(intervals, N, M, f, c, eps):
    # intervals :: [(l, u)]

    # Nresults :: [(float, [float])]
```

```python
    # fst is the lowest function value achieved
    # snd is the list of parameter values
    Nresults = []

    # unzip list of tuples
    l = [l for l, u in intervals]
    u = [u for l, u in intervals]

    # sample and run N times
    for s in range(N):
        r = np.random.uniform(l, u)
        v = f(r)
        Nresults += [(v.copy(), r.copy())]

    # topM :: [(float, [float])]
    topM = take_top(M, Nresults)
    if (c-1 == 0):
        return topM[0]

    # when c = 0 do the pulse

    # collect the top results from applying grs to topM
    # top_params :: [(float, [float])]
    top_params = []
    for (_, params) in topM:
        new_intervals = get_new_intervals(params, intervals, M/eps)
        top_params += [global_random_search_2(new_intervals, N, M, f, c-1, eps)]

    return take_top(1, top_params)[0]

intervals = [(-10, 10), (-10, 10)]
testGRS3(intervals, N=25, M=2, f=f1, c=2, eps=1, runs=1000)

intervals = [(-10, 10), (-10, 10)]
testGRS(intervals, N=125, f=f1, runs=1000)

intervals = [(-10, 10), (-10, 10)]
testGRS3(intervals, N=25, M=1, f=f1, c=3, eps=1.5, runs=1000)

intervals = [(-10, 10), (-10, 10)]
testGRS(intervals, N=100, f=f1, runs=1000)

intervals = [(-10, 10), (-10, 10)]
testGRS3(intervals, N=10, M=1, f=f2, c=3, eps=2, runs=1000)

intervals = [(-10, 10), (-10, 10)]
testGRS(intervals, N=40, f=f2, runs=1000)

def testGRS3(intervals, N, M, f, c, eps, runs):
    r = []
    for i in range(runs):
        r += [grs3(intervals, N, M, f, c, eps)[0]]

    print(runs, "runs of GRS3")
    print("Number of f evals:", (N + (N*M*c)))
    print("Standard deviation on final function values: ", np.std(r))
    print("Mean on final function values: ", np.mean(r))

intervals = [(-100, 100), (-100, 100)]
f = global_random_search_2(intervals, 100, 3, f1, 4, 1)
print(f)

intervals = [(-100, 100), (-100, 100)]
%timeit f = global_random_search_2(intervals, 100, 3, f2, 4, 1)
print(f)

intervals = [(-100, 100), (-100, 100)]
f = grs3(intervals, 100, 3, f2, 4, 1)
```

```
print(f)

intervals = [(-100, 100), (-100, 100)]
%timeit f = grs3(intervals, 100, 3, f1, 4, 1)
print(f)

def testParams(alpha, beta1, beta2, batch_size, epochs):
    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)))

    adam = tf.keras.optimizers.Adam(learning_rate=alpha, beta_1=beta1, beta_2=beta2, name='Adam')
    model.compile(loss="categorical_crossentropy", optimizer=adam, metrics=["accuracy"])

    model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)
    preds = model.predict(x_test)
    cce = tf.keras.losses.CategoricalCrossentropy()
    val = cce(y_test, preds).numpy()
    return val
t = lambda x: testParams(alpha=x[0], beta1=x[1], beta2=x[2], batch_size=x[3], epochs=x[4])

intervals = [
    (0.001, 0.01),              # alpha
    (0.5, 0.999),               # beta1
    (0.5, 0.999),               # beta2
    (4, 256),               # batch_size
    (5, 30),               # epochs
]

v = grs3(intervals, 20, 1, t, c=3, eps=1.5)

print(v)

v = global_random_search(intervals, 80, t)

print(v)

import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, regularizers
from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
from keras.layers import Conv2D, MaxPooling2D, LeakyReLU
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.utils import shuffle
import matplotlib.pyplot as plt
plt.rc('font', size=18)
plt.rcParams['figure.constrained_layout.use'] = True
import sys

# 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]
```

```
#x_test=x_test[1:500]; y_test=y_test[1:500]

# 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)
```

```
# Algorithms.py

# Algorithms implement a similar inteface:
# - specific names on input arguments
# - accesses function related things through the OptimisableFunction class
# - needs to return X, Y

import numpy as np


from OptimisationAlgorithmToolkit.Function import FunctionIterator

class OptimisationAlgorithm:
    def __init__(self, algorithm, algorithm_name):
        self.algorithm = algorithm
        self.algorithm_name = algorithm_name

        arguments = algorithm.__code__.co_varnames[:algorithm.__code__.co_argcount]
        self.mini_batch_parameters = ('b')
        self.all_parameters = arguments
        self.standard_parameters = ("x0", "f", "iters")
        self.hyperparameters = list(filter(lambda arg: arg not in self.standard_parameters, arguments))

    def __type_check_parameters(self, input_record):
        for key in input_record.keys():
            if key not in self.all_parameters:
                raise NameError(key + " is not one of: " + str(self.all_parameters))
        for key in self.all_parameters:
            if key not in input_record:
                if key is not "b":
                    raise NameError(key + " is missing from input: " + str(list(input_record.keys())))

    def set_parameters(self, **input_record):
        self.__type_check_parameters(input_record)
        self.parameter_values = input_record
        return self

    def run(self):
        inputs = self.__make_input()
        for input in inputs:
            input["X"], input["Y"] = self.algorithm(**input)
            input["X"] = np.array(input["X"])
            input["Y"] = np.array(input["Y"])
            input["algorithm"] = self
        return inputs

    def __make_input(self):
        kwargs = self.parameter_values.copy()
        expected_vector = { "x0" }
        for key, value in kwargs.items():
            if key in expected_vector:
                value = np.array(value)
                if value.ndim == 1:
                    kwargs[key] = [value]
            else:
                if type(value) is not list:
```

```
                    kwargs[key] = [value]

        keys = kwargs.keys()
        partial_dicts = [{}]
        for key in keys:
            partial_dicts_new = []
            for partial_dict in partial_dicts:
                for value in kwargs[key]: # making a new partial dict for each value
                    partial_dict_new = partial_dict.copy()
                    partial_dict_new[key] = value
                    partial_dicts_new += [partial_dict_new]
                    partial_dicts = partial_dicts_new
        return partial_dicts



def polyak(x0, f, f_star, eps, iters, b=None):
    fi = FunctionIterator(f, b, iters) ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]

    for fN, dfs in fi:
        fdif = f(*x) - f_star
        df_squared_sum = np.sum(np.array([df(*x)**2 for df in dfs]))
        alpha = fdif / (df_squared_sum + eps)
        x = x - alpha * np.array([df(*x) for df in dfs])

        X += [x] ; Y += [f(*x)]
    return X, Y

Polyak = OptimisationAlgorithm(algorithm=polyak,
                               algorithm_name="Polyak")

def constant_step(x0, alpha, f, iters, b=None):
    fi = FunctionIterator(f, b, iters) ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]

    for fN, dfs in fi:
        step = alpha * np.array([df(*x) for df in dfs])
        x = x - step

        X += [x] ; Y += [f(*x)]
    return X, Y

ConstantStep = OptimisationAlgorithm(algorithm=constant_step,
                                     algorithm_name="Constant")

def adagrad(x0, f, alpha0, eps, iters, b=None):
    fi = FunctionIterator(f, b, iters) ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]

    df_vector_sum = np.zeros(len(dfs))
    for fN, dfs in fi:
        df_vec = np.array([df(*x) for df in dfs])
        df_vector_sum += df_vec**2
        alphas = alpha0 / (np.sqrt(df_vector_sum) + eps)
        x = x  - (alphas * df_vec)

        X += [x] ; Y += [f(*x)]
    return X, Y

Adagrad = OptimisationAlgorithm(algorithm=adagrad,
                                algorithm_name="Adagrad")

def rmsprop(x0, f, alpha0, beta, eps, iters, b=None):
    fi = FunctionIterator(f, b, iters) ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]

    sum = np.zeros(len(x0)) ; alpha = alpha0
```

```
    for fN, dfs in fi:
      x = x - (alpha * np.array([df(*x) for df in dfs]))
      sum = beta * sum + (1 - beta) * np.array([df(*x)**2 for df in dfs])
      alpha = alpha0 / (np.sqrt(sum) + eps)

      X += [x] ; Y += [f(*x)]
    return X, Y

RMSProp = OptimisationAlgorithm(algorithm=rmsprop,
                                algorithm_name="RMSProp")


def heavy_ball(x0, f, alpha, beta, iters, b=None):
    fi = FunctionIterator(f, b, iters) ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]

    z = np.zeros(len(x0))
    for fN, dfs in fi:
        z = beta * z + alpha * np.array([df(*x) for df in dfs])
        x = x - z

        X += [x] ; Y += [f(*x)]
    return X, Y

HeavyBall = OptimisationAlgorithm(algorithm=heavy_ball,
                                  algorithm_name="Heavy Ball")

def adam(x0, f, eps, beta1, beta2, alpha, iters, b=None):
    fi = FunctionIterator(f, b, iters) ; f = f.function ; x = x0 ; X = [x] ; Y = [f(*x)]

    m = np.zeros(len(x0)) ; v = np.zeros(len(x0)) ; k = 1
    for fN, dfs in fi:
        m = beta1 * m + (1 - beta1) * np.array([df(*x) for df in dfs])
        v = beta2 * v + (1 - beta2) * np.array([(df(*x)**2) for df in dfs])
        mhat = (m / (1 - beta1**k))
        vhat = (v / (1 - beta2**k))
        x = x - alpha * (mhat / (np.sqrt(vhat) + eps))
        k = k + 1

        X += [x] ; Y += [f(*x)]
    return X,Y

Adam = OptimisationAlgorithm(algorithm=adam,
                             algorithm_name="Adam")
```

```
# Each record should contain its label depending on what are the other records in the list.

# The user semi-mannually inputs what the title should be.
# - Have utility functions to extract pieces of the title from the list of records.

# Function that takes in a list of records.
#  - For each record determines the label based on what is in the list of records.

# Perhaps there should be a function that calculatesthe meta information that is used by both
# - utility functions that extract peieces of title
# - function that assigns the labels to each individual record


# MetaInfo: extracts:
# - Which optimisaiton functions there area
# - For each optimisation function
#   - What are the parameters that are not varying and what values do they have
#   - What are the parameters that are varying and what values do they have



```

```

# {
#   ...
#   ...
#   label:
# }
# label made up from what uniquely identifies it
# - first is optimisation algorithm itself
# - second are the hyperparmeters that uniqely identifies the cluster of algorithms
#   - RMSProp alpha0=0.4
#   - RMSProp alpha0=0.5
#   - Adam    beta1=0.2  beta2=0.4
#   - Adam    beta1=0.3  beta2=0.5

# - Then would like to extract the common descriptive pieces
#   - Different common pieces per algorithm used
#     - Records -> AlgorihtmName -> CommonThingsString
#       - Adam: beta1=0.1 eps=0.0001 iters=50 x0=[1, 1]
#       - RMSProp:  eps=0.0001 iters=50 x0=[1, 1]


# MetaRecord extracts
# - Algorithms and their corresponding Varying fields
# {
#   "Adam"    : ["eps", "beta1"]
#   "RMSProp" : ["eps", "alpha0"]
# }


# meta_record = meta(inputs)
# inputs = create_labels(meta_record, inputs)
# inputs = get_title(meta_record, inputs)

# get_titles returns
# {
#   "Adam" : "Adam: beta1=0.1 eps=0.0001 iters=50 x0=[1, 1]",
#   "RMSProp" : "RMSProp:  eps=0.0001 iters=50 x0=[1, 1]"
# }

import numpy as np

def get_titles(records):
    m = meta(records)
    t = {}
    for alg_name in m.keys():
        t[alg_name] = get_title(alg_name, records, m)
    return t

def get_title(alg_name, records, meta):
    title = f'{alg_name}:'
    algs = alg(records, alg_name)

    r = algs[0]
    params = set(r["algorithm"].all_parameters)
    varied = meta[alg_name]
    params.remove('f')
    params = params - varied

    for p in params:
        if p in r:
            title += f' {p}={r[p]}'
    return title

def create_labels(records):
    m = meta(records)
    for r in records:
        r['label'] = create_label(r, m)

```

```
# e.g: Adam    beta1=0.2  beta2=0.4
def create_label(record, meta):
    alg_name = record['algorithm'].algorithm_name
    differing_fields = meta[alg_name]
    label = f'{alg_name}'
    for f in differing_fields:
        label += f' {f}={record[f]}'
    return label

# {
#   "Adam"    : ["eps", "beta1"]
#   "RMSProp" : ["eps", "alpha0"]
# }
def meta(records):
    mr = {}
    algs = get_algs(records)
    for a in algs:
        a_records = alg(records, a)
        mr[a] = differing_fields(a_records)
    return mr

def differing_fields(records):
    diff_fields = set({})
    t = records[0]
    for r in records:
        for key, value in r.items():
            # print("a")
            # print(t[key])
            # print(type(value))
            # print(isinstance(value, list))

            if isinstance(value, list):
                value = np.array(value)
            if isinstance(t[key], list):
                t[key] = np.array(t[key])

            b = t[key] == value
            # print(b)
            # print(type(b))
            if type(b) == np.ndarray:
                b = b.all()
            if not (b):
                diff_fields.add(key)


    diff_fields.discard('X')
    diff_fields.discard('Y')
    return diff_fields

# extract one algorithm type, filter out the rest
def alg(records, algorithm_name):
    return list(filter(lambda r: r['algorithm'].algorithm_name == algorithm_name,
    records))

# gets algorithms names in the records
def get_algs(records):
    algs = set({})
    for r in records:
        algs.add(r['algorithm'].algorithm_name)
    return algs


# wonder how this would look in haskell
# funcitonal operators and stuff, would it make it easier.
```

```
# Functions that will be optimised:
# - Allows access to
#   - Parital Derivatives
#   - String representation of the function (latex)
```

```python
# - Constructor uses sympy to obtain the above

from sympy import simplify, latex, lambdify
import numpy as np

class BatchedFunction:
    def __init__(self, f, M, name="f"):
        self.f = f
        self.function = lambda  x1, x2 : f(np.array([x1,x2]), minibatch=M)
        self.M = M
        self.function_name = name

class FunctionIterator:
    # b = len(M) will behave like normal gradient descent
    def __init__(self, f, b, i):
        self.i = i
        self.f = f
        self.function = f.function
        if type(f) is SymbolicFunction:
            self.batch = False
        else:
            self.batch = True
            self.M = f.M
            self.m = len(self.M)
            if b is None:
                self.b = len(self.M) # act as non stochastic
            else:
                self.b = b
            if self.b == len(self.M):
                self.shuffle = True
            else:
                self.shuffle = True

    def __iter__(self):
        self.epoch = -1
        self.batch_start_indices = iter(())
        return self

    def __next__(self):
        if (self.i <= 0):
            raise StopIteration
        self.i -= 1
        if not self.batch:
            return self.function, self.f.partial_derivatives

        self.batch_index = next(self.batch_start_indices, None)
        if self.batch_index == None:
            self.epoch += 1
            if self.shuffle:
                np.random.shuffle(self.M)
            self.batch_start_indices = iter(np.arange(0, (self.m-self.b)+1, self.b))
            self.batch_index = next(self.batch_start_indices, None)

        N = np.arange(self.batch_index, self.batch_index + self.b)
        fN = lambda x: self.f.f(x, minibatch=self.M[N])
        dfs = [(lambda x1, x2, xi=i : finite_diff(fN, np.array([x1, x2]), xi)) for i
   in range(2)]
        return fN, dfs

class SymbolicFunction:
    def __init__(self, sympy_function, sympy_symbols, function_name):
        self.sympy_symbols = sympy_symbols
        self.function_name = function_name

        self.sympy_function = sympy_function
        self.function = lambdify(sympy_symbols, sympy_function, modules="numpy")
        self.function_list_arg = lambda x: self.function(x[0], x[1])

```

```
        self.sympy_partial_derivatives = [sympy_function.diff(symbol) for symbol in
    sympy_symbols]
        self.partial_derivatives = [lambdify(sympy_symbols, p, modules="numpy") for p
     in self.sympy_partial_derivatives]

    def __iter__(self):
        return self

    def __next__(self):
        return self.function, self.partial_derivatives

    def __parameters_string(self):
        s = map(latex, self.sympy_symbols)
        return ",".join(s)

    def latex(self):
        return self.function_name + "(" + self.__parameters_string() + ") = " + latex
    (simplify(self.sympy_function))

    def partials_latex(self):
        s = map(latex, self.sympy_symbols)
        z = zip(self.sympy_partial_derivatives, s)
        return [ "\\frac{\\partial " +  self.function_name + "}{\\partial " +
    partial_wrt_name + "}" "=" + latex(simplify(partial))
                for (partial, partial_wrt_name) in z]

    def print_partials_latex(self):
        for p in self.partials_latex():
            print(p)


def finite_diff(f, x, i, delta=0.0001):
    d = np.zeros(len(x)) ; d[i] = delta
    return (f(x) - f(x - d)) / delta
```

```
import matplotlib as mpl
mpl.rcParams['figure.dpi'] = 200
mpl.rcParams['figure.facecolor'] = '1'
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')

from OptimisationAlgorithmToolkit.DataType import create_labels, get_titles

from matplotlib.ticker import LogLocator

import numpy as np


def plot_contour(records, x1r, x2r, log=False, sym=False):
    create_labels(records)
    t = get_titles(records)

    f = records[0]['f']

    X1, X2 = np.meshgrid(x1r, x2r)
    Z = np.vectorize(f.function)(X1, X2)
    if log:
        plt.contourf(X1, X2, Z, locator=LogLocator(), cmap=plt.get_cmap('gist_earth')
    )
    else:
        plt.contourf(X1, X2, Z, cmap=plt.get_cmap('gist_earth'))
    xlim = plt.xlim()
    ylim = plt.ylim()
    for (X, label) in dicts_collect(("X", "label"), records):
        plt.plot(X.T[0], X.T[1], linewidth=2.0, label=label)

    f = records[0]['f']
    function_name = f.function_name
    if sym:
```

```
        f_latex = f.latex()
        title = rf'${f_latex}$' + " \n " + title_string(records)
    else:
        title = title_string(records)
    plt.xlabel(r'$x_1$')
    plt.ylabel(r'$x_2$')
    plt.title(title)


    plt.xlim(xlim)
    plt.ylim(ylim)
    plt.legend()
    plt.colorbar()

def plot_path(records, xr):
    create_labels(records)
    f = records[0]['f'].function;
    function_name = records[0]['f'].function_name
    f_latex = records[0]['f'].latex()

    yr = [f(x) for x in xr]
    plt.plot(xr, yr)
    xlim = plt.xlim()
    ylim = plt.ylim()

    for (X, label) in dicts_collect(("X", "label"), records):
        xs = X.flatten()
        ys = [f(x) for x in xs]
        plt.plot(xs, ys, linewidth=2.0, label=label)

    plt.xlim(xlim)
    plt.ylim(ylim)
    plt.legend()
    title = rf'${f_latex}$' + "\n" + title_string(records)
    plt.title(title)
    plt.ylabel(f'${function_name}$')
    plt.xlabel(r'$x$')

def plot_step_size(records, mean=True):
    create_labels(records)
    fig, ax = plt.subplots()
    f_latex = records[0]['f'].latex()
    for (X, label) in dicts_collect(("X", "label"), records):
        if mean:
            s = np.array([np.mean(x) for x in  step_sizes(X).T])
            ax.plot(np.arange(1, len(s)+1), s, linewidth=2.0, label=label)
        else:
            sX = step_sizes(X)
            for i in range(len(sX)):
                x = i + 1
                s = sX[i]
                ax.plot(np.arange(1, len(s)+1), s, linewidth=2.0, label=label + f'
$x_{x} step$')
    ax.legend()

    title = rf'${f_latex}$' + " \n " + title_string(records)
    if mean:
        ax.set_title("Mean Step Across x's \n" + title)
    else:
        ax.set_title("Mean Step Across x's \n" + title)
    ax.set_ylabel(f'Step Size')
    ax.set_xlabel(r'$i$')


def title_string(records):
    title = ""
    t = get_titles(records)
    for _, v in t.items():
```

```
        title += v + '\n'
    return title

# [[x11 x21 x31 ...] [x12 x22 x32 ...] ...]  -> [[x12-x11 x13-x12 ...] [x22-x21 x23-x22 ...] ...]
def step_sizes(X):
    return np.array([(x[1:] - x[:-1]) for x in X.T])



def ploty(records, sym=False):
    create_labels(records)
    t = get_titles(records)

    fig, ax = plt.subplots()
    for (X, Y, label) in dicts_collect(("X", "Y", "label"), records):
        ax.plot(range(len(Y)), Y, linewidth=2.0, label=label)


    f = records[0]['f']
    function_name = f.function_name

    if sym:
        f_latex = f.latex()
        title = rf'${f_latex}$' + " \n " + title_string(records)
    else:
        title = title_string(records)

    ax.set_title(title)
    ax.set_ylabel(f'${function_name}$')
    ax.set_xlabel(r'$i$')

    ax.legend()
    return ax

def dicts_collect(keys, dicts):
    values = []
    for dict in dicts:
        values += [[dict[key] for key in keys]]
    return values
```