Function for the Assignment:

```
def generate_trainingdata(m=25):
    return np.array([0,0])+0.25*np.random.randn(m,2)

def f(x, minibatch):
    # loss function sum_{w in training data} f(x,w)
    y=0; count=0
    for w in minibatch:
        z=x-w-1
        y=y+min(16*(z[0]**2+z[1]**2), (z[0]+5)**2+(z[1]+10)**2)
        count=count+1
    return y/count
```

A)

(1) The Stochastic_Gradient_Descent class implements the mini-batch Stochastic Gradient Descent (SGD) algorithm. The class uses variables to store the function to be minimized (f), its derivatives (df), the current parameter values (x), and additional algorithm-specific parameters (params). It also initializes a records dictionary to keep track of the parameter values, the function value, and the step size after each update.The minibatch method is the most important part mini-batch SGD implementation. This method shuffles the entire training dataset and then creates mini-batches by slicing the data. For each mini-batch, it calls the selected algorithm to update the model parameters.

```
    def minibatch(self):
        np.random.shuffle(self.training_data)
        n = len(self.training_data)
        for i in range(0, n, self.batch_size):
            if i + self.batch_size > n:
                continue
            data = self.training_data[i:(i + self.batch_size)]
            self.algorithm(data)
        self.records['x'].append(deepcopy(self.x))
        self.records['f'].append(self.f(self.x, self.training_data))
```

A derivative function is used to calculate derivative value.

```
    def derivative(self, i, data):
        Sum = 0
        for j in range(self.batch_size):
            Sum = Sum + self.df[i](*self.x, *data[j])
        return Sum / self.batch_size
```

(2) Figure 1 contains wireframe and contour plot for f(x,T) where x=[x1,x2] and T is training dataset. The loss values are calculated across the complete set of training data. The shape of the surface plotted indicates that the loss function is likely convex, with the lowest point representing the optimal

parameters that we're want to find. The surface dips towards this lowest point, suggesting there is a single point where the loss is at its minimum.

The range for the plots is -20 to 20. This range is chosen to ensure we capture the full breadth of the function's behavior and include the area where we expect to find the optimal parameters.

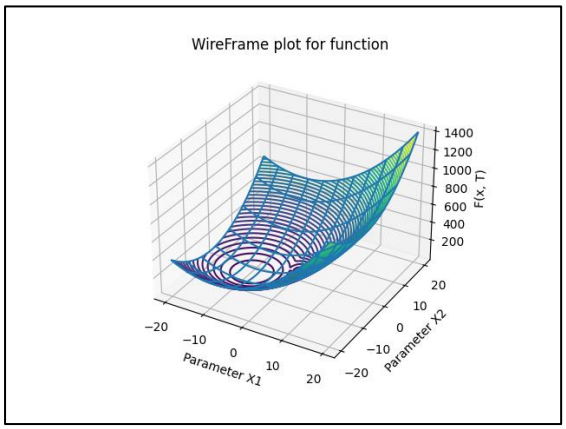


Figure 1

(3)

For the function, f, I have used SymPy library to calculate the derivative. Using the four symbols, x0,x1,w0,and w1 I have differentiated f once with respect to x0 and then with respect to x1.

```
f = sp.Min(16 * ((x0 - w0)**2 + (x1 - w1)**2), ((x0 - w0 + 5)**2 + (x1 - w1 + 10)**2))
```

The results obtained are as follows:

```
(-32*w0 + 32*x0)*Heaviside(-16*(-w0 + x0)**2 - 16*(-w1 + x1)**2 + (-w0 + x0 + 5)**2 + (-w1 + x1 + 10)**2) + (-2*w0 + 2*x0 + 10)*Heaviside(1
6*(-w0 + x0)**2 + 16*(-w1 + x1)**2 - (-w0 + x0 + 5)**2 - (-w1 + x1 + 10)**2)
(-32*w1 + 32*x1)*Heaviside(-16*(-w0 + x0)**2 - 16*(-w1 + x1)**2 + (-w0 + x0 + 5)**2 + (-w1 + x1 + 10)**2) + (-2*w1 + 2*x1 + 20)*Heaviside(1
6*(-w0 + x0)**2 + 16*(-w1 + x1)**2 - (-w0 + x0 + 5)**2 - (-w1 + x1 + 10)**2)
```

B)

(1) The gradient descent algorithm iteratively changes the parameters of a function to minimize its output. The step size, controls the magnitude of each update and is crucial for the convergence and speed of the optimisation process. Also, to determine an appropriate step size for the gradient descent, 100 iterations were performed with different step sizes. The chosen step sizes were 0.1, 0.01, 0.001, and 0.0001. Each of these was tested to observe their impact on the convergence of the function value f over the number of iterations.

The convergence of f as observed in the performance plots indicates how each step size behaves over iterations. The convergence plot with respect to function values f against the iterations shows that the step size of 0.1 converges quickly but appears to be unstable, as indicated by the oscillations in f. As the step size decreases, the convergence becomes more stable, and the oscillations reduce. However, the smaller step sizes, particularly 0.001 and 0.0001, show a very slow convergence.

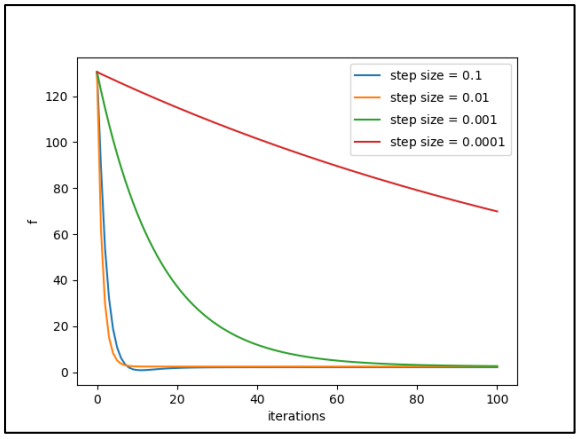


Figure 2

The contour plot shows the trajectory of the algorithm's path through the function's domain. Each line represents a level curve where the function has the same value, and the color gradient represents the height of the function, with darker colors indicating lower values. The goal of the algorithm is to reach the darkest area, which corresponds to the function's minimum. The selected step size of 0.01 offers a trade-off between a rapid descent and precise navigation towards the minimum, as seen by the trajectory on the contour plot. This plot is particularly informative as it illustrates not only the endpoint but also the path taken, thereby providing insight into the algorithm's behavior during optimization. Also, we can conclude that 0,01 is appropriate choice for the function.

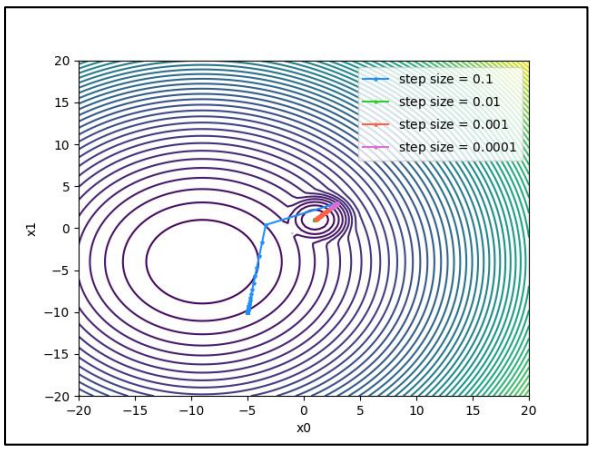


Figure 3

(2) The SGD optimization was performed using a step size α=0.1 that was previously found as optimal through question 1. We ran the optimization for 80 iterations, with a mini-batch size of 5, to minimize the loss function starting from the initial point x=[3,3]. The trials was repeated five times to assess the changes in convergence.

As shown in Figure 4, the loss function f decreases sharply within the first few iterations before stabilizing across all trials, showing rapid convergence towards the minimum. The variability from run to run is very small, suggesting that the introduced randomness through shuffling the mini-batches does not significantly affect the outcome of the optimization.

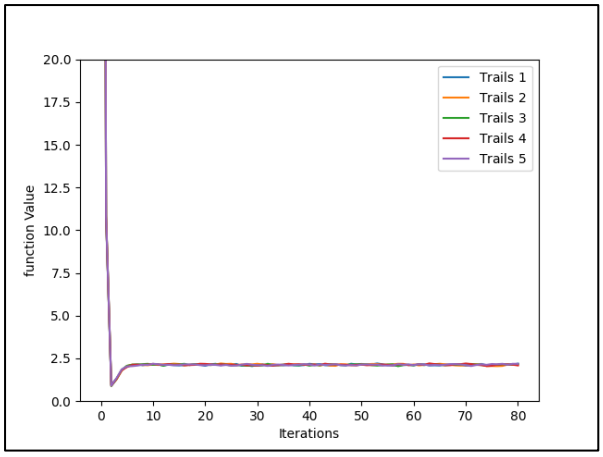


Figure 4

Also, it can be seen in Figure 5 that the trajectories on the contour plot of the function, where the paths taken by the optimizer in all trials appear to be overlapping. This shows the algorithm's stability and robustness against the inherent randomness in the mini-batch selection.

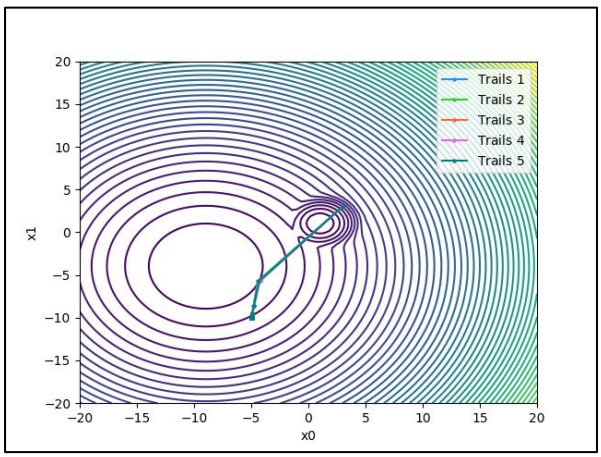


Figure 5

The steadiness of results in multiple runs shows that Mini-batch SGD with a constant step size of $\alpha=0.1$ is not only effective but also reliable. When contrasted with the CGD approach from question (i), Mini-batch SGD shows a faster convergence rate, reaching the vicinity of the global minimum within fewer iterations. The convergence to the global minimum, despite the noise in the training data, shows the effectiveness of the chosen step size.

(3)

The trails were done using the SGD algorithm with a constant step size of $\alpha=0.1$. The mini-batch sizes chosen for comparison were 1, 5, 10, 25, and 30. The optimization process ran for 50 iterations for each batch size. The results can be seen through the plots.

Figure 6 shows the convergence patterns of the loss function f over iterations for different batch sizes. It can be seen that for the smaller batch sizes there is fluctuation in f, indicating higher variance in the gradient estimates. Conversely, larger batch sizes show a smoother convergence, yet potentially slower.

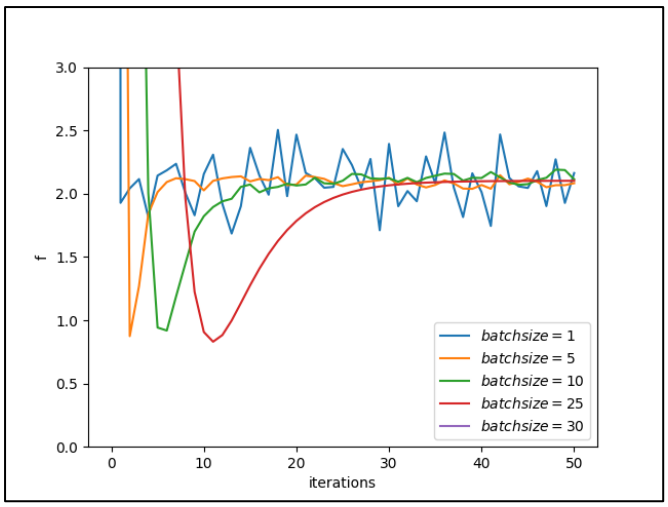


Figure 6

Also it can be seen in Figure 8 that the paths of smaller batch sizes appear more erratic compared to the more direct trajectories of larger batch sizes towards the minimum.

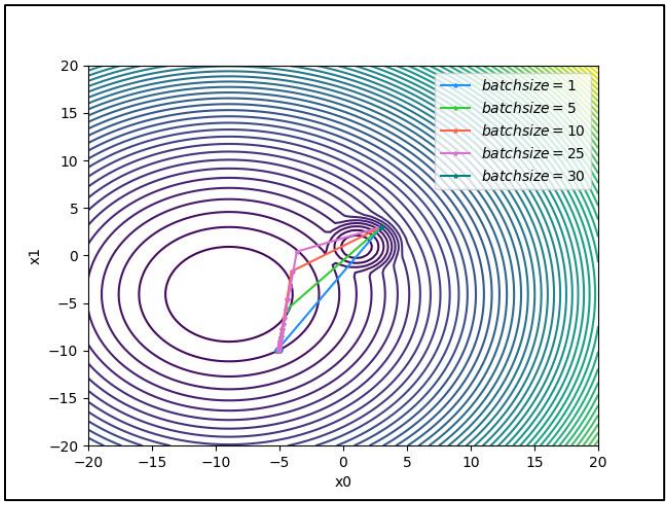


Figure 7

(4) With the mini-batch size fixed at 5, different step sizes of 0.1, 0.01, 0.001, and 0.0001 which are same as above. It can be seen that that a step size of 0.1 allows for rapid and stable convergence towards the function's minimum. This is different to the step sizes of 0.01 and 0.001, where the convergence is slower. Particularly, the step size of 0.0001 is too slow, leading to hardly any progression towards minimizing the function within the given iterations, as clearly seen in Figure 8.

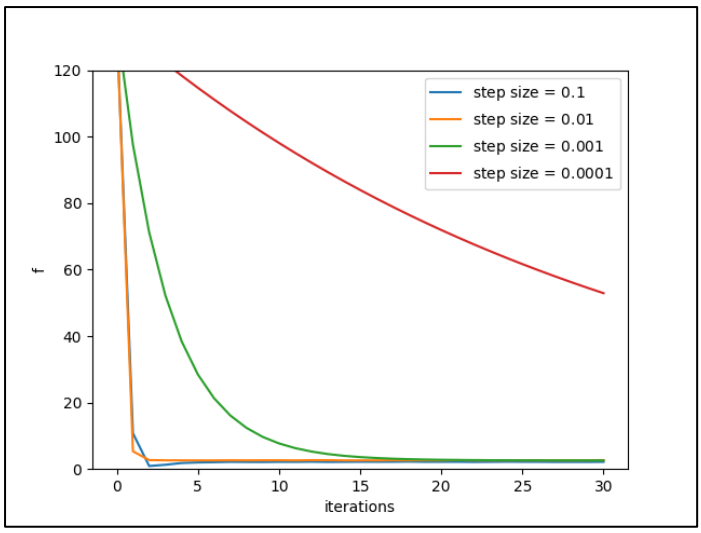


Figure 8

The contour plots in Figure 9 shows the optimization paths corresponding to these step sizes. The larger step size of 0.1 directs the optimizer in a beeline toward the function's minimum. In comparison, smaller step sizes result in a more cautious approach, taking more iterations to reach the same proximity to the minimum. It's apparent that a too-small step size hampers the optimizer ability to make significant progress per iteration, and therefore slowing down the convergence.

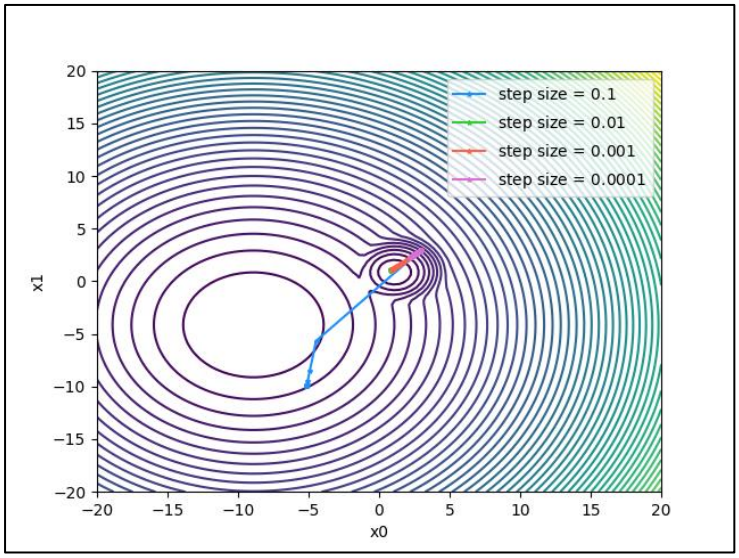
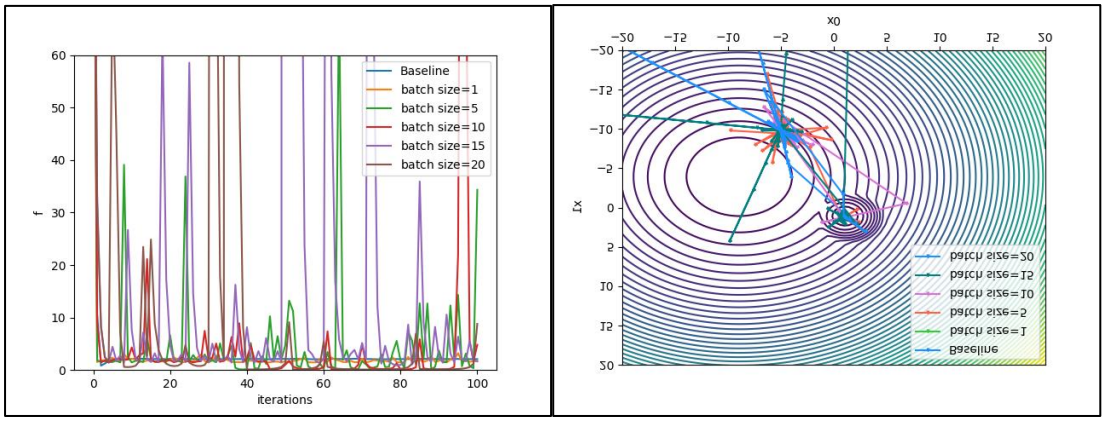


Figure 9

Comparing these findings with the previous observations from (iii), where varying mini-batch sizes were considered, it becomes evident that while batch size influences the stability of convergence, the step size is predominantly responsible for the rate at which convergence occurs. A balanced step size of 0.1 proves to be optimal in the context of the given optimization problem, showing a fine balance between efficient convergence and stability of the path, even when there was potential noise in the training data.

C)

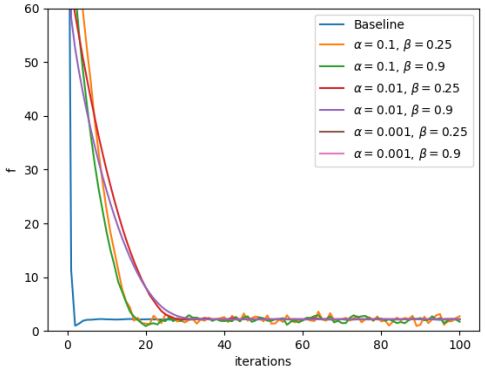
(1)Polyak



With the Polyak step size, the descent is no longer uniform across iterations. Instead, it dynamically adjusts potentially allowing for more aggressive steps when the gradient is large and more cautious updates when it is small. This behavior can result in a more responsive approach to reaching the function's minimum, as shown in the Figure above. In this figure, we observe a varied descent rate across different mini-batch sizes. The initial descent is rapid, reflecting the algorithm's ability to take larger steps when far from the minimum. However, as the iterations progress, we see significant oscillations, especially with smaller batch sizes. These oscillations are indicative of an overstep in the descent path, where the step size overshoots the minimum, leading to a corrective backtrack in subsequent iterations.

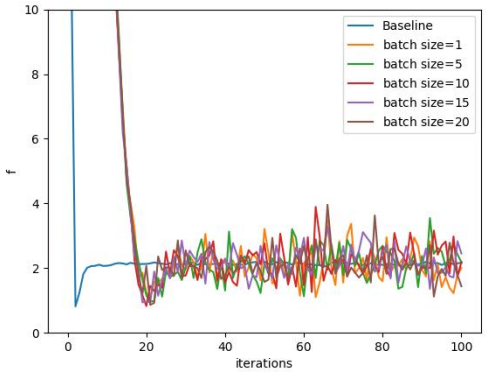
In comparison to the baseline, the Polyak approach demonstrates an increased sensitivity to the choice of mini-batch size. Smaller batches lead to larger, more erratic steps due to higher variance in the gradient estimates, while larger batches provide a more averaged and stable gradient, smoothing out the convergence path. Therefore, we can say that it performs worse than baseline.

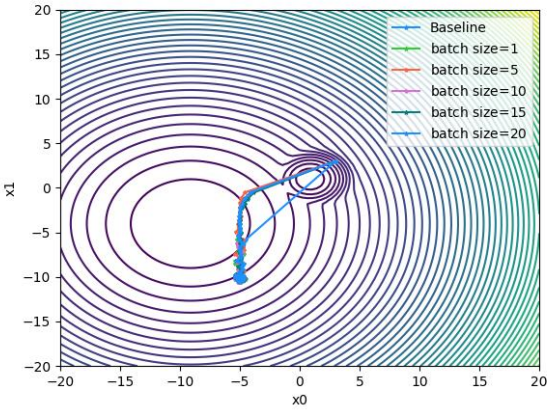
(2) RMSprop



For RMSprop, with values of alpha as same as above alpha=0.1,0.01,0.001 and beta size of 0.9 and 0.25 as used in last assignment the results can be seen the above plots.

It can be seen that correct convergence occurs in the cases when alpha= 0.1 with the most stable results occurring when β = 0.9. As such, these values will be used when varying the batch size in below figures.



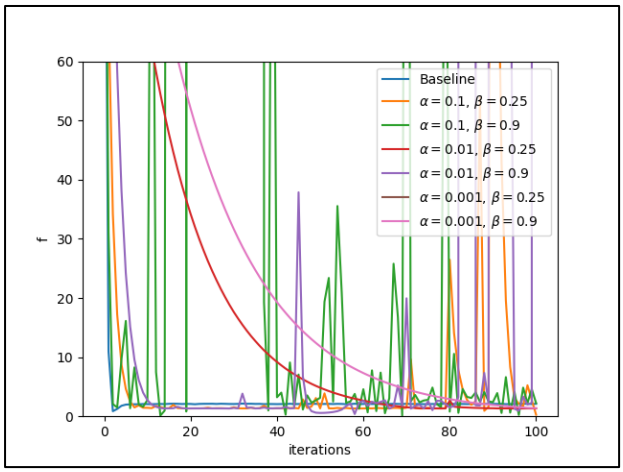


All of the trials appear to perform much better than when Polyak step size was used but they still fail to outperform the baseline. The blue line represents the baseline approach shows quick convergence without apparent overshooting. Convergence in case of on the global minimum takes about three times longer than the baseline with every trial failing to converge on a stable value. Unlike Polyak, flatter regions of the function do not result in an increased step size. However, the step size does not completely stabilise as it is dependent on the moving average of square gradient sums which is constantly changing during iterations. Overall, it can be seen that there are lots of fluctuation.

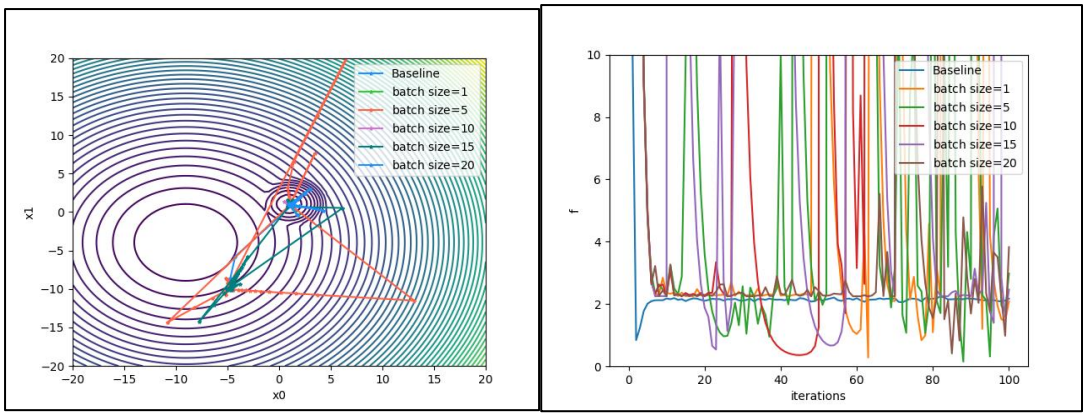
Mini-Batch Size Effect: Larger mini-batch sizes, such as 15 and 20, show little smoother paths towards the centre, indicating more stable and reliable gradient estimations. Smaller mini-batch sizes, such as 1 and 5, show more variation and a less direct path, which can be beneficial for escaping potential local minima but may also result in a longer time to converge

(3) Heavyball

For Heavyball, with values of alpha as same as above alpha=0.1,0.01,0.001 and beta size of 0.9 and 0.25 as used in last assignment the results can be seen the below plots.



It can be seen that better convergence occurs in the case of alpha=0.1 and beta=0,9. The convergence occurs fatsest when beta is 0.9 it will be used in below plots for varying the batch size.



For the contour plot, it can be seen that the path taken by baseline is relatively smooth and direct towards the center, which suggests that the constant step size is well-suited for this specific loss function, quickly converging towards a minimum.

Compared to the baseline, the Heavy Ball paths are more erratic. This could be due to the momentum term causing the optimizer to overshoot the minimum and then correct its course, which can be particularly pronounced with smaller batch sizes batch size = 1 or 5.

Larger mini-batch sizes show trajectories that start to smooth out like batch size = 15 and 20, suggesting they are better at utilizing the momentum to converge toward the minimum. However, none of the Heavy Ball paths are as efficient as the baseline in this scenario.

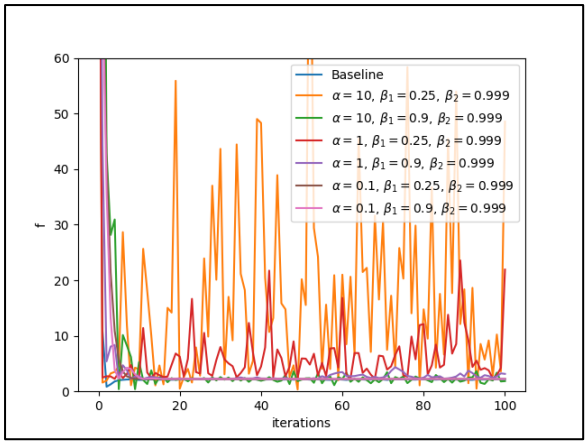
Also it can be seen by the line graph that baseline quickly converges and then stabilizes at a low value of the loss function. This quick stabilization indicates that the step size is large enough to make rapid progress but not so large as to cause instability.

For heavyball it exhibit significant fluctuations in loss value, even after many iterations. These fluctuations could be due to the momentum term accumulating gradient information from the past iterations, which can cause oscillations, especially if the step size is not reduced over time.

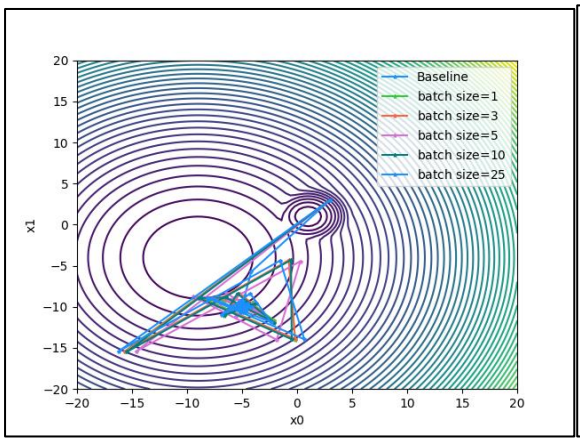
The lines corresponding to smaller mini-batch sizes 1 and 5 show more dramatic oscillations in the loss function, which indicates that noise in the gradient estimates is high, and the momentum term may exacerbate this effect. As the batch size increases, the extent of these oscillations decreases slightly but is still pronounced compared to the baseline.

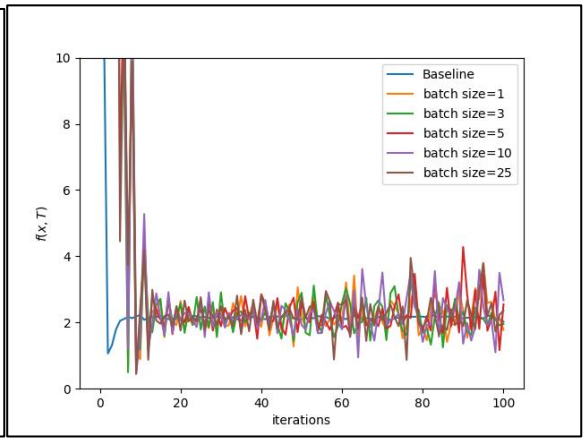
(4) Adam

For Adam, with values of alpha as same as above alpha=0.1,0.01,0.001 and beta size of 0.9 and 0.25 as used in last assignment the results can be seen the below plots.



It can be seen from the above plot that when alpha=10, with beta1 at either 0.9 or 0.25, the algorithm stabilizes more rapidly towards zero, showing fewer and smaller fluctuations in 'f' and will be used in the below plots for varying batch size.





From the contour plot it can be seen that the baseline's path is direct and swift, with minimal oscillation, converging towards the center quickly. This indicates that the constant step size is effectively guiding the optimizer towards the minimum without significant divergence.

While the paths for the Adam algorithm with varying mini-batch sizes start out similarly but then diverge, creating loops and curves away from the direct path. Smaller mini-batch sizes show more erratic behavior, which is typical given the increased noise in gradient estimates from fewer data points. Larger batch sizes, like batch size=25, seem to smooth out the trajectory slightly, yet they still exhibit some looping behavior, indicating a continued struggle to stabilize and hone in on the minimum.

Also from the line graph it can be seen that the baseline shows a steep initial drop in the loss value, leveling off relatively quickly, which is indicative of an effective step size enabling the optimizer to make large, confident strides towards the minimum.While for Adam, it shows variability in the loss values, failing to reach the stability seen in the baseline. They display continued oscillations throughout the optimization process, with no clear sign of stabilization as the iterations progress. Like RMSProp, the step size does not stabilise due to the moving average of square gradient sums constantly changing during iterations.
The smaller mini-batch sizes, such as 1 and 3, show the highest volatility in loss, suggesting that noise in the gradient is significantly impacting the updates. As the mini-batch size increases, the severity of

the oscillations decreases, yet even with larger mini-batch sizes, Adam does not match the baseline's performance.

APPENDIX FOR CODE

```
import matplotlib.pyplot as plt
import numpy as np
from copy import deepcopy
from enum import Enum
import sympy as sp
"""Function for the Assignment"""
def generate_trainingdata(m=25):
    return np.array([0,0])+0.25*np.random.randn(m,2)

def f(x, minibatch):
    # loss function sum_{w in training data} f(x,w)
    y=0; count=0
    for w in minibatch:
        z=x-w-1
        y=y+min(16*(z[0]**2+z[1]**2), (z[0]+5)**2+(z[1]+10)**2)
        count=count+1
    return y/count

TD = generate_trainingdata()
x = np.linspace(start=-20, stop=20, num=100)
y = np.linspace(start=-20, stop=20, num=100)
z = []
for i in x:
    k = []
    for j in y:
        params = [i, j]
        k.append(f(params, TD))
    z.append(k)
z = np.array(z)
x, y = np.meshgrid(x, y)
contour_ax = plt.subplot(111, projection='3d')
contour_ax.contour3D(x, y, z, 60)
contour_ax.set_xlabel('Parameter X1')
contour_ax.set_ylabel('Parameter X2')
contour_ax.set_zlabel('F(x, T)')
contour_ax.set_title('Contour plot for function')
wireFrame_ax = plt.subplot(111, projection='3d')
wireFrame_ax.plot_wireframe(x, y, z, rstride=15, cstride=10)
wireFrame_ax.set_xlabel('Parameter X1')
wireFrame_ax.set_ylabel('Parameter X2')
wireFrame_ax.set_zlabel('F(x, T)')
wireFrame_ax.set_title('WireFrame plot for function')
plt.show()
```

```
#A(iii)
x0, x1, w0, w1 = sp.symbols('x0 x1 w0 w1', real=True)
f = sp.Min(16 * ((x0 - w0)**2 + (x1 - w1)**2), ((x0 - w0 + 5)**2 + (x1 - w1 +
10)**2))
df_one = sp.diff(f, x0)
df_two = sp.diff(f, x1)
print(df_one)
print(df_two)


#b(i)
def generate_trainingdata(m=25):
    return np.array([0,0])+0.25*np.random.randn(m,2)

def f(x, minibatch):
    # loss function sum_{w in training data} f(x,w)
    y=0; count=0
    for w in minibatch:
        z=x-w-1
        y=y+min(16*(z[0]**2+z[1]**2), (z[0]+5)**2+(z[1]+10)**2)
        count=count+1
    return y/count

"""Stochastic_Gradient_Descent Algorithm Implementation"""
class ALGO(Enum):
    constant_algo = 0
    polyak_algo = 1
    rmsprop_algo = 2
    heavyball_algo = 3
    adam_algo = 4


class Stochastic_Gradient_Descent:
    def __init__(self, f, df, x, algorithm, params, batch_size,
training_data):
        self.epsilon = 1e-8
        self.f = f
        self.df = df
        self.x = deepcopy(x)
        self.n = len(x)
        self.params = params
        self.batch_size = batch_size
        self.training_data = training_data
        self.records = {
            'x': [deepcopy(self.x)],
            'f': [self.f(self.x, self.training_data)],
            'step': []
        }
        self.algorithm = self.get_algorithm(algorithm)
```

```
        self.initial(algorithm)

    def minibatch(self):
        # shuffle training data
        np.random.shuffle(self.training_data)
        n = len(self.training_data)
        for i in range(0, n, self.batch_size):
            if i + self.batch_size > n:
                continue
            data = self.training_data[i:(i + self.batch_size)]
            self.algorithm(data)
        self.records['x'].append(deepcopy(self.x))
        self.records['f'].append(self.f(self.x, self.training_data))

    def get_algorithm(self, algorithm):
        if algorithm == ALGO.constant_algo:
            return self.constant_algo
        elif algorithm == ALGO.polyak_algo:
            return self.polyak_algo
        elif algorithm == ALGO.rmsprop_algo:
            return self.rmsprop_algo
        elif algorithm == ALGO.heavyball_algo:
            return self.heavy_ball_algo
        else:
            return self.adam_algo

    def initial(self, algorithm):
        if algorithm == ALGO.rmsprop_algo:
            self.records['step'] = [[self.params['alpha']] * self.n]
            self.vars = {
                'sums': [0] * self.n,
                'alphas': [self.params['alpha']] * self.n
            }
        elif algorithm == ALGO.heavyball_algo:
            self.records['step'] = [0]
            self.vars = {
                'z': 0
            }
        elif algorithm == ALGO.adam_algo:
            self.records['step'] = [[0] * self.n]
            self.vars = {
                'ms': [0] * self.n,
                'vs': [0] * self.n,
                'step': [0] * self.n,
                't': 0
            }

    def constant_algo(self, data):
```

```
        alpha = self.params['alpha']
        for i in range(self.n):
            self.x[i] -= alpha * self.derivative(i, data)
        self.records['step'].append(alpha)

    def polyak_algo(self, data):
        Sum = 0
        for i in range(self.n):
            Sum = Sum + self.derivative(i, data) ** 2
        step = self.f(self.x, data) / (Sum + self.epsilon)
        for i in range(self.n):
            self.x[i] -= step * self.derivative(i, data)
        self.records['step'].append(step)

    def rmsprop_algo(self, data):
        alpha = self.params['alpha']
        beta = self.params['beta']
        alphas = self.vars['alphas']
        sums = self.vars['sums']
        for i in range(self.n):
            self.x[i] -= alphas[i] * self.derivative(i, data)
            sums[i] = (beta * sums[i]) + ((1 - beta) * (self.derivative(i,
data) ** 2))
            alphas[i] = alpha / ((sums[i] ** 0.5) + self.epsilon)
        self.records['step'].append(deepcopy(alphas))

    def heavy_ball_algo(self, data):
        alpha = self.params['alpha']
        beta = self.params['beta']
        z = self.vars['z']
        Sum = 0
        for i in range(self.n):
            Sum += self.derivative(i, data) ** 2

        z = (beta * z) + (alpha * self.f(self.x, data) / (Sum + self.epsilon))
        for i in range(self.n):
            self.x[i] -= z * self.derivative(i, data)
        self.vars['z'] = z
        self.records['step'].append(z)

    def adam_algo(self, data):
        alpha = self.params['alpha']
        beta1 = self.params['beta1']
        beta2 = self.params['beta2']
        ms = self.vars['ms']
        vs = self.vars['vs']
        step = self.vars['step']
        t = self.vars['t']
```

```
        t += 1
        for i in range(self.n):
            ms[i] = (beta1 * ms[i]) + ((1 - beta1)*self.derivative(i, data))
            vs[i] = (beta2 * vs[i]) + ((1 - beta2)*(self.derivative(i, data)
** 2))
            _m = ms[i] / (1 - (beta1 ** t))
            _v = vs[i] / (1 - (beta2 ** t))
            step[i] = alpha * (_m / ((_v ** 0.5) + self.epsilon))
            self.x[i] -= step[i]
        self.vars['t'] = t
        self.records['step'].append(deepcopy(step))

    def derivative(self, i, data):
        Sum = 0
        for j in range(self.batch_size):
            Sum = Sum + self.df[i](*self.x, *data[j])
        return Sum / self.batch_size


def df_one(x0, x1, w0, w1):
    heaviside_1 = np.heaviside(-16 * (-w0 + x0 - 1)**2 + (-w0 + x0 + 5)**2 -
16 * (-w1 + x1 - 1)**2 + (-w1 + x1 + 10)**2, 0)
    heaviside_2 = np.heaviside(16 * (-w0 + x0 - 1)**2 - (-w0 + x0 + 5)**2 + 16
* (-w1 + x1 - 1)**2 - (-w1 + x1 + 10)**2, 0)

    term1 = (-32 * w0 + 32 * x0 - 32) * heaviside_1
    term2 = (-2 * w0 + 2 * x0 + 10) * heaviside_2

    return term1 + term2

def df_two(x0, x1, w0, w1):
    heaviside_1 = np.heaviside(-16 * (-w0 + x0 - 1)**2 + (-w0 + x0 + 5)**2 -
16 * (-w1 + x1 - 1)**2 + (-w1 + x1 + 10)**2, 0)
    heaviside_2 = np.heaviside(16 * (-w0 + x0 - 1)**2 - (-w0 + x0 + 5)**2 + 16
* (-w1 + x1 - 1)**2 - (-w1 + x1 + 10)**2, 0)

    term1 = (-32 * w1 + 32 * x1 - 32) * heaviside_1
    term2 = (-2 * w1 + 2 * x1 + 20) * heaviside_2

    return term1 + term2



colors = ['dodgerblue', 'limegreen', 'tomato', 'orchid', 'teal']



def plots(f, training_data, xs, legend):
    X_Data = np.linspace(-20, 20, 100)
```

```
    Y_Data = np.linspace(-20, 20, 100)
    Z_Data = []
    for x in X_Data:
        z = []
        for y in Y_Data: z.append(f([x, y], training_data))
        Z_Data.append(z)
    Z = np.array(Z_Data)
    X, Y = np.meshgrid(X_Data, Y_Data)
    plt.contour(X, Y, Z, 60)
    plt.xlabel('x0')
    plt.ylabel('x1')
    for i in range(len(xs)):
        x0 = [point[0] for point in xs[i]]
        x1 = [point[1] for point in xs[i]]
        plt.plot(x0, x1, color=colors[i % len(colors)], marker='*',
markeredgecolor=colors[i % len(colors)], markersize=3)
        plt.xlim([-20, 20])
        plt.ylim([-20, 20])
    plt.legend(legend)
    plt.show()
"""Question B"""
#b(1)
def plot_sgd_performance1(f, df):
    training_data = generate_trainingdata()
    count = 100
    iters = list(range(count + 1))
    step_sizes = [0.1, 0.01, 0.001, 0.0001]
    labels = [f'step size = ${step_size}$' for step_size in step_sizes]
    xs = []
    fs = []
    for i, step_size in enumerate(step_sizes):
        sgd = Stochastic_Gradient_Descent(f, df, [3, 3], ALGO.constant_algo,
{'alpha': step_size}, batch_size=len(training_data),
                  training_data=training_data)
        for _ in range(count):
            sgd.minibatch()
        plt.plot(iters, sgd.records['f'])
        xs.append(deepcopy(sgd.records['x']))
        fs.append(deepcopy(sgd.records['f']))

    plt.xlabel('iterations')
    plt.ylabel('f')
    plt.legend(labels)
    plt.show()
    plots(f, training_data, xs, labels)
#b2
def plot_sgd_performance2(f, df):
    training_data = generate_trainingdata()
```

```
    times = 5
    count = 80
    iters = list(range(count + 1))
    alpha = 0.1
    labels = [f'Trails ${i + 1}$' for i in range(times)]
    xs = []
    fs = []
    for trial in range(times):
        sgd = Stochastic_Gradient_Descent(f, df, [3, 3], ALGO.constant_algo,
{'alpha': alpha}, 5, training_data)
        for _ in range(count):
            sgd.minibatch()
        plt.plot(iters, sgd.records['f'], label=labels[trial])
        xs.append(deepcopy(sgd.records['x']))
        fs.append(deepcopy(sgd.records['f']))
    plt.ylim([0, 20])
    plt.xlabel('Iterations')
    plt.ylabel('function Value')
    plt.legend()
    plt.show()
    plots(f, training_data, xs, labels)
#b3
def plot_sgd_performance3(f, df):
    training_data = generate_trainingdata()
    count = 50
    iters = list(range(count + 1))
    alpha = 0.1
    batch_sizes = [1, 5, 10, 25, 30]
    labels = [f'$batch size={n}$' for n in batch_sizes]
    xs = []
    fs = []
    for i, n in enumerate(batch_sizes):
        sgd = Stochastic_Gradient_Descent(f, df, [3, 3], ALGO.constant_algo,
{'alpha': alpha}, n, training_data)
        for _ in range(count):
            sgd.minibatch()
        plt.plot(iters, sgd.records['f'], label=labels[i])
        xs.append(deepcopy(sgd.records['x']))
        fs.append(deepcopy(sgd.records['f']))
    plt.ylim([0, 3])
    plt.xlabel('iterations')
    plt.ylabel('f')
    plt.legend()
    plt.show()
    plots(f, training_data, xs, labels)
#b4
def plot_sgd_performance4(f, df):
    training_data = generate_trainingdata()
```

```
    count = 30
    iters = list(range(count + 1))
    step_sizes = [0.1, 0.01, 0.001, 0.0001]
    labels = [f'step size = ${step_size}$' for step_size in step_sizes]
    xs = []
    fs = []
    for i, step_size in enumerate(step_sizes):
        sgd = Stochastic_Gradient_Descent(f, df, [3, 3], ALGO.constant_algo,
{'alpha': step_size}, 5, training_data)
        for _ in range(count):
            sgd.minibatch()
        plt.plot(iters, sgd.records['f'], label=labels[i])
        xs.append(deepcopy(sgd.records['x']))
        fs.append(deepcopy(sgd.records['f']))
    plt.ylim([0, 120])
    plt.xlabel('iterations')
    plt.ylabel('f')
    plt.legend()
    plt.show()
    plots(f, training_data, xs, labels)

"""Question C"""
def SGD_Polyak(f, df):
    training_data = generate_trainingdata()
    count = 100
    iters = list(range(count + 1))
    xs = []
    fs = []
    labels = ['Baseline']
    Sgd_base = Stochastic_Gradient_Descent(f, df, [3, 3], ALGO.constant_algo,
{'alpha': 0.1}, 5, training_data)
    for _ in range(count):
        Sgd_base.minibatch()
    plt.plot(iters, Sgd_base.records['f'], label=labels[0])
    xs.append(deepcopy(Sgd_base.records['x']))
    fs.append(deepcopy(Sgd_base.records['f']))
    batch_sizes = [1, 5, 10, 15, 20]
    for n in batch_sizes:
        sgd = Stochastic_Gradient_Descent(f, df, [3, 3], ALGO.polyak_algo, {},
n, training_data)
        for _ in range(count):
            sgd.minibatch()
        labels.append(f'batch size=${n}$')
        plt.plot(iters, sgd.records['f'], label=labels[-1])
        xs.append(deepcopy(sgd.records['x']))
        fs.append(deepcopy(sgd.records['f']))
    plt.ylim([0, 60])
    plt.xlabel('iterations')
```

```
    plt.ylabel('f')
    plt.legend()
    plt.show()
    plots(f, training_data, xs, labels)

def SGD_RMSProp(f, df):
    count = 100
    iters = list(range(count + 1))
    training_data = generate_trainingdata()
    xs = []
    fs = []
    labels = ['Baseline']
    Sgd_base = Stochastic_Gradient_Descent(f, df, [3, 3], ALGO.constant_algo,
{'alpha': 0.1}, 5, training_data)
    for _ in range(count):
        Sgd_base.minibatch()
    plt.plot(iters, Sgd_base.records['f'], label=labels[0])
    xs.append(deepcopy(Sgd_base.records['x']))
    fs.append(deepcopy(Sgd_base.records['f']))
    alphas = [0.1, 0.01, 0.001]
    betas = [0.25, 0.9]
    for alpha in alphas:
        for beta in betas:
            sgd = Stochastic_Gradient_Descent(f, df, [3, 3],
ALGO.rmsprop_algo,
                        {'alpha': alpha, 'beta': beta}, 5, training_data)
            for _ in range(count):
                sgd.minibatch()
            labels.append(f'$\\alpha={alpha},\\,\\beta={beta}$')
            plt.plot(iters, sgd.records['f'], label=labels[-1])
            xs.append(deepcopy(sgd.records['x']))
            fs.append(deepcopy(sgd.records['f']))
    plt.ylim([0, 60])
    plt.xlabel('iterations')
    plt.ylabel('f')
    plt.legend()
    plt.show()
    xs = []
    fs = []
    labels = ['Baseline']
    plt.plot(iters, Sgd_base.records['f'], label=labels[0])
    xs.append(deepcopy(Sgd_base.records['x']))
    fs.append(deepcopy(Sgd_base.records['f']))
    batch_sizes = [1, 5, 10, 15, 20]
    for batch_size in batch_sizes:
        sgd = Stochastic_Gradient_Descent(f, df, [3, 3], ALGO.rmsprop_algo,
{'alpha': 0.1, 'beta': 0.9}, 5, training_data)
        for _ in range(count):
```

```
            sgd.minibatch()
        labels.append(f'batch size=${batch_size}$')
        plt.plot(iters, sgd.records['f'], label=labels[-1])
        xs.append(deepcopy(sgd.records['x']))
        fs.append(deepcopy(sgd.records['f']))
    plt.ylim([0, 10])
    plt.xlabel('iterations')
    plt.ylabel('f')
    plt.legend()
    plt.show()
    plots(f, training_data, xs, labels)


def SGD_HeavyBall(f, df):
    training_data = generate_trainingdata()
    count = 100
    iters = list(range(count + 1))
    xs = []
    fs = []
    labels = ['Baseline']
    Sgd_base = Stochastic_Gradient_Descent(f, df, [3, 3], ALGO.constant_algo,
{'alpha': 0.1}, 5, training_data)
    for _ in range(count):
        Sgd_base.minibatch()
    plt.plot(iters, Sgd_base.records['f'], label=labels[0])
    xs.append(deepcopy(Sgd_base.records['x']))
    fs.append(deepcopy(Sgd_base.records['f']))
    alphas = [0.1, 0.01, 0.001]
    betas = [0.25, 0.9]
    for alpha in alphas:
        for beta in betas:
            sgd = Stochastic_Gradient_Descent(f, df, [3, 3],
ALGO.heavyball_algo,
                      {'alpha': alpha, 'beta': beta}, 5, training_data)
            for _ in range(count):
                sgd.minibatch()
            labels.append(f'$\\alpha={alpha},\\,\\beta={beta}$')
            plt.plot(iters, sgd.records['f'], label=labels[-1])
            xs.append(deepcopy(sgd.records['x']))
            fs.append(deepcopy(sgd.records['f']))
    plt.ylim([0, 60])
    plt.xlabel('iterations')
    plt.ylabel('f')
    plt.legend()
    plt.show()
    xs, fs = [], []
    labels = ['Baseline']
    plt.plot(iters, Sgd_base.records['f'], label=labels[0])
    xs.append(deepcopy(Sgd_base.records['x']))
```

```
    fs.append(deepcopy(Sgd_base.records['f']))
    batch_sizes = [1, 5, 10, 15, 20]
    for batch_size in batch_sizes:
        sgd = Stochastic_Gradient_Descent(f, df, [3, 3], ALGO.heavyball_algo,
{'alpha': 0.1, 'beta': 0.25}, 5, training_data)
        for _ in range(count):
            sgd.minibatch()
        labels.append(f'batch size=${batch_size}$')
        plt.plot(iters, sgd.records['f'], label=labels[-1])
        xs.append(deepcopy(sgd.records['x']))
        fs.append(deepcopy(sgd.records['f']))
    plt.ylim([0, 10])
    plt.xlabel('iterations')
    plt.ylabel('f')
    plt.legend()
    plt.show()
    plots(f, training_data, xs, labels)

def SGD_Adam(f, df):
    training_data = generate_trainingdata()
    count = 100
    iters = list(range(count + 1))
    xs = []
    fs = []
    labels = ['Baseline']
    Sgd_base = Stochastic_Gradient_Descent(f, df, [3, 3], ALGO.constant_algo,
{'alpha': 0.1}, 5, training_data)
    for _ in range(count):
        Sgd_base.minibatch()
    plt.plot(iters, Sgd_base.records['f'], label=labels[0])
    xs.append(deepcopy(Sgd_base.records['x']))
    fs.append(deepcopy(Sgd_base.records['f']))
    alphas = [10, 1, 0.1]
    beta1s = [0.25, 0.9]
    beta2s = [0.999]
    for alpha in alphas:
        for beta1 in beta1s:
            for beta2 in beta2s:
                sgd = Stochastic_Gradient_Descent(f, df, [3, 3],
ALGO.adam_algo,
                                  {'alpha': alpha, 'beta1': beta1, 'beta2': beta2}, 5,
training_data)
                for _ in range(count):
                    sgd.minibatch()
                labels.append(
                    f'$\\alpha={alpha},\\,\\beta_1={beta1},\\,\\beta_2={beta2}
$'
                )
```

```
                plt.plot(iters, sgd.records['f'], label=labels[-1])
                xs.append(deepcopy(sgd.records['x']))
                fs.append(deepcopy(sgd.records['f']))
    plt.ylim([0, 60])
    plt.xlabel('iterations')
    plt.ylabel('f')
    plt.legend()
    plt.show()
    xs = []
    fs = []
    labels = ['Baseline']
    plt.plot(iters, Sgd_base.records['f'], label=labels[0])
    xs.append(deepcopy(Sgd_base.records['x']))
    fs.append(deepcopy(Sgd_base.records['f']))
    batch_sizes = [1, 3, 5, 10, 25]
    for batch_size in batch_sizes:
        sgd = Stochastic_Gradient_Descent(f, df, [3, 3], ALGO.adam_algo,
                  {'alpha': 10, 'beta1': 0.9, 'beta2': 0.999}, 5,
training_data)
        for _ in range(count):
            sgd.minibatch()
        labels.append(f'batch size=${batch_size}$')
        plt.plot(iters, sgd.records['f'], label=labels[-1])
        xs.append(deepcopy(sgd.records['x']))
        fs.append(deepcopy(sgd.records['f']))
    plt.ylim([0, 10])
    plt.xlabel('iterations')
    plt.ylabel('$f(x, T)$')
    plt.legend()
    plt.show()
    plots(f, training_data, xs, labels)



if __name__ == '__main__':
    plot_sgd_performance1(f, [df_one, df_two])
    plot_sgd_performance2(f, [df_one, df_two])
    plot_sgd_performance3(f, [df_one, df_two])
    plot_sgd_performance4(f, [df_one, df_two])

    SGD_Polyak(f, [df_one, df_two])
    SGD_RMSProp(f, [df_one, df_two])
    SGD_HeavyBall(f, [df_one, df_two])
    SGD_Adam(f, [df_one, df_two])
```