Optimisation Algorithms for Data Analysis - Assignment 3

CS7DS2

Maurice Kiely - 23350601

23/03/2024

Questions

a) i)

```
    def SGD(optimizer, f, starting_point, data, batch_size=32,
        max_iters=1000, **kwargs):
    optimizer = optimizer.strip().lower()
    if optimizer not in ['constant', 'polyak', 'rmsprop', 'hb',
        'adam']:
        raise ValueError("Optimizer argument not valid")

    if optimizer == 'constant':
        alpha = kwargs.get('alpha', 0.01)
        tol = kwargs.get('tol', 1e-8)
        path, f_vals = constant(f=f, starting_point=starting_point,
            data=data, alpha=alpha, tol=tol, max_iters=max_iters,
            batch_size=batch_size)
        print("Stochastic-Gradient-Descent using Constant Optimiser")
        return path, f_vals

    elif optimizer == 'polyak':
        ...
        ...
        ...
    else optimizer == 'adam':
        alpha = kwargs.get('alpha', 0.001)
        beta1 = kwargs.get('beta1', 0.9)
        beta2 = kwargs.get('beta2', 0.999)

        epsilon = kwargs.get('epsilon', 1e-8)
        tol = kwargs.get('tol', 1e-5)

        path, f_vals = adam(f=f, starting_point=starting_point,
            data=data, alpha=alpha, beta1=beta1, beta2=beta2,
            epsilon=epsilon, max_iters=max_iters, tol=tol,
            batch_size=32)
        print("Stochastic-Gradient-Descent using Adam Optimiser")
        return path, f_vals
```

The function I implemented for SGD takes an argument to specify which optimization algorithm to use. Once this is done, the function uses kwargs to take in the necessary hyperparameters for each function.

Each function is essentially the same as in the previous assignment, however, there are 2 key differences due to the lack of an explicit function.

Firstly, the training data is broken into batches with the mini_batch function and secondly, the gradient is calculated using finite-difference for each mini-batch.

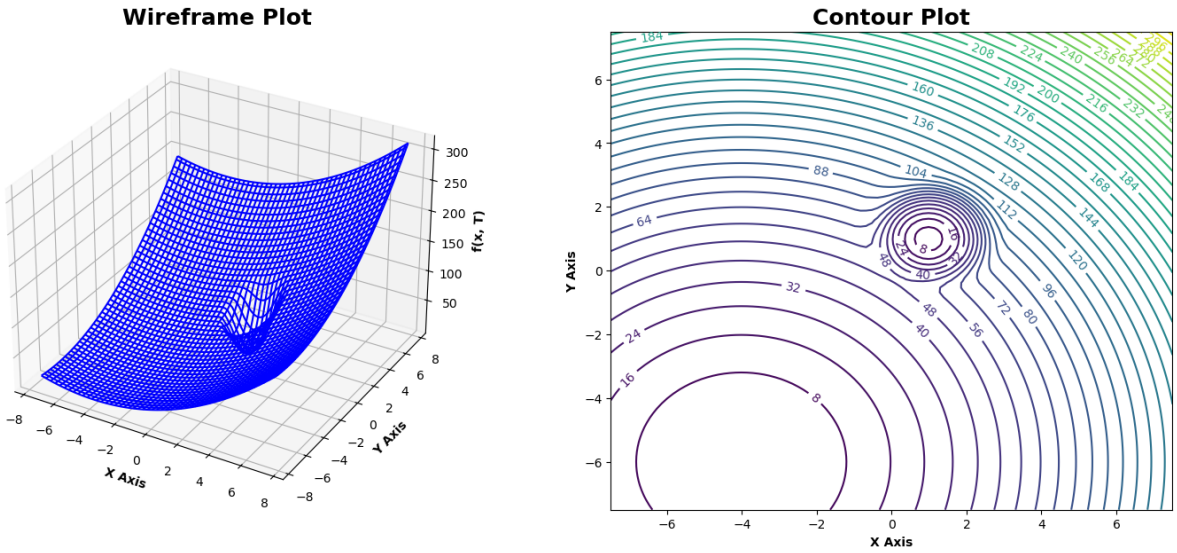
Both of these functions can be seen here as follows:

```
def mini_batch(data, batch_size):
    batches = []
    np.random.shuffle(data)
    n = len(data)
    for i in np.arange(0, n, batch_size):
        batch = data[i:(i + batch_size)] if i + batch_size < n
            else data[i:n]
        batches.append(batch)
    return batches
```

```
def batch_finite_diff(f, pt, batch, h=1e-8):
    batch_dx = (f([pt[0] + h, pt[1]], batch) - f(pt, batch))/h
    batch_dy = (f([pt[0], pt[1] + h], batch) - f(pt, batch))/h
    return batch_dx, batch_dy
```

In each of the optimizing algorithms, the batches are used to get both a function value and gradient.

a) ii)



I chose to plot the function from x,y $\in [-7.5, 7.5]$ as outside of these values the function is well-behaved and clearly bowl-shaped. The most important feature I thought to capture was the valley at $(x, y) \approx (1.8, 1.8)$. This is where many of our SGD algorithms will converge. The code used to calculate the grid is seen below:

```
training_data = generate_trainingdata()
x = np.linspace(-7.5, 7.5, 100)
y = np.linspace(-7.5, 7.5, 100)
z = []
for i in x:
    for j in y:
        z.append(f([i,j], training_data))
```

```
    z = np.array(z).reshape(100, 100)
    x, y = np.meshgrid(x, y)
```

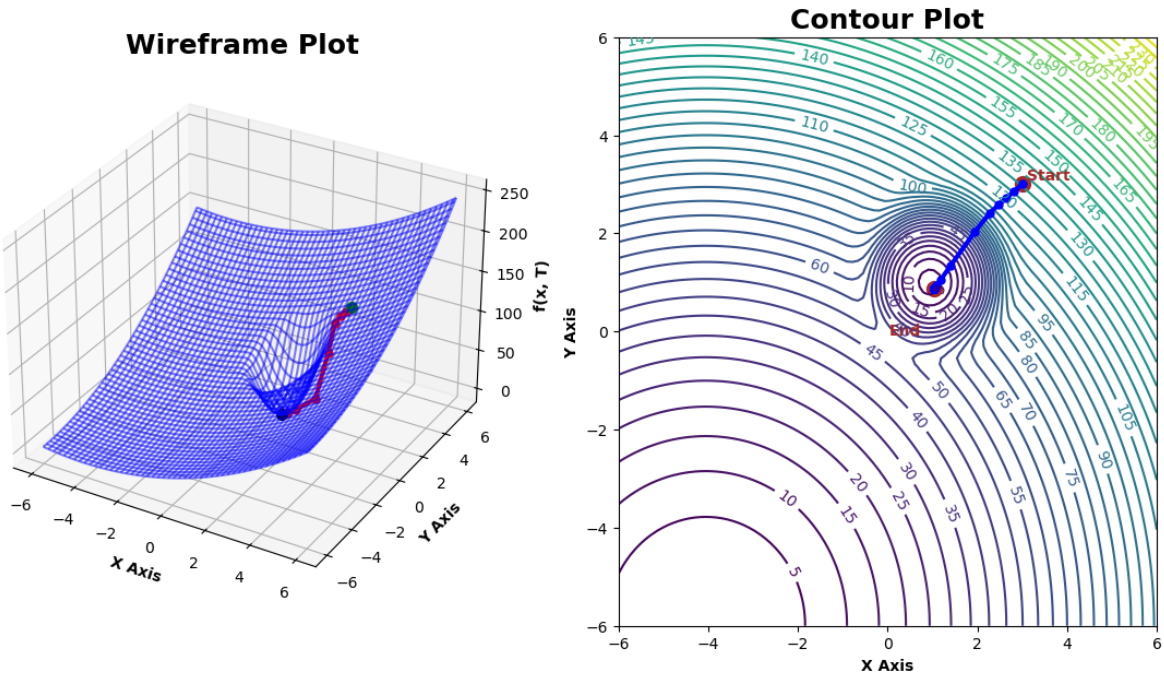
a) iii)

The method I used to find the derivative at each point was that of finite difference.

```
    def batch_finite_diff(f, pt, batch, h=1e-8):
        batch_dx = (f([pt[0] + h, pt[1]], batch) - f(pt, batch))/h
        batch_dy = (f([pt[0], pt[1] + h], batch) - f(pt, batch))/h
        return batch_dx, batch_dy
```

The code implements the function: $f'(x, N) \approx \lim_{h \to 0} \frac{f(x+h, N) - f(x, N)}{h}$ for each $x_i \in \vec{x}$ which is the derivative. This calculates the rate of change of the loss function based on the points in the batch.

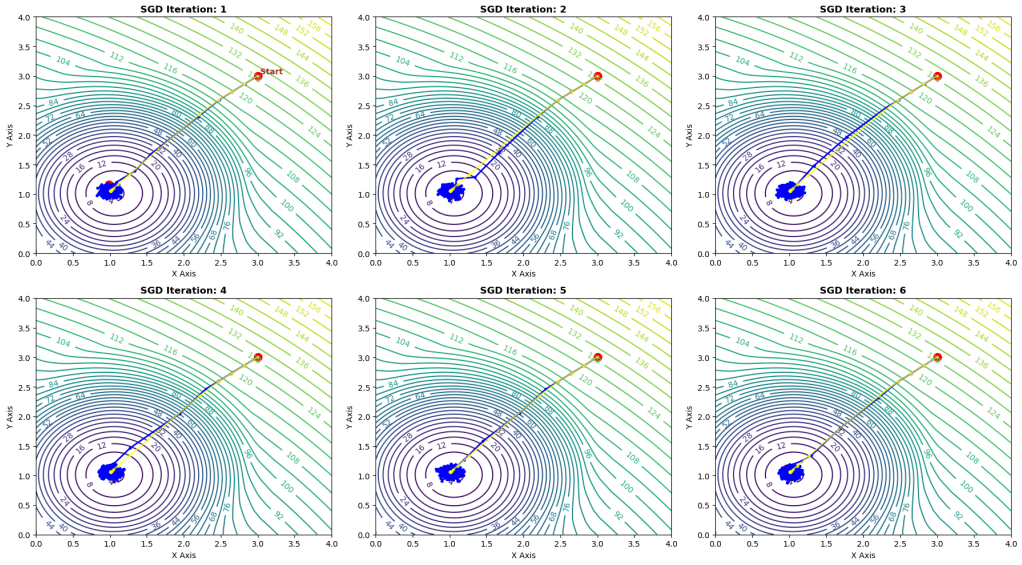
b) i)



Here, the step size that I chose to use was $\alpha = 0.01$ for 100 iterations. I chose alpha as it will converge relatively quickly and won't overshoot the valley for a steep gradient.

This gradient-descent was done by using the SGD function shown above with a constant optimizer and batch_size = training_data

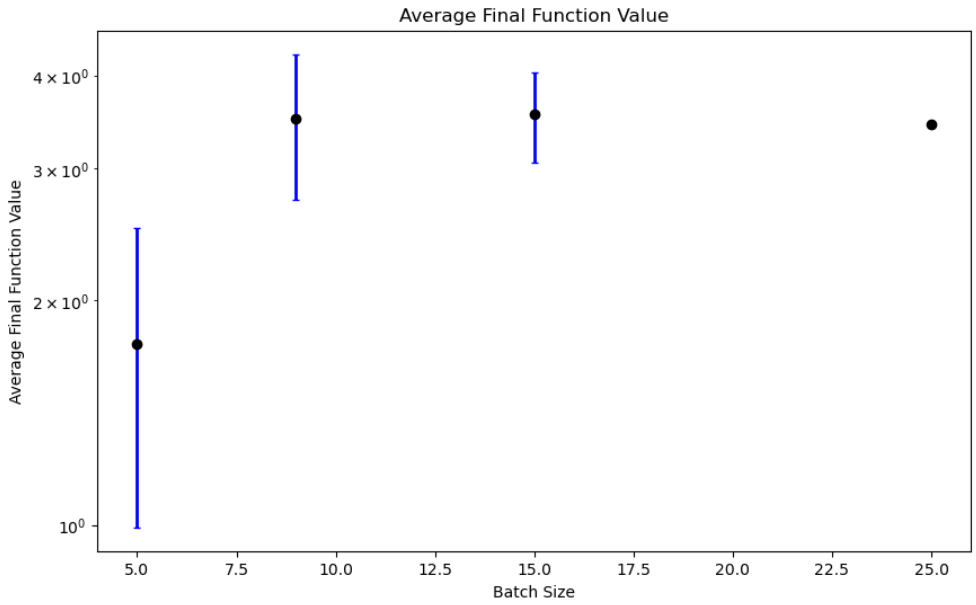
b) ii)



In this image, we can see the path taken by 6 different iterations of the same function. This is simply a SGD with a batch size of 5. The path taken here is the blue line, with the yellow being that of the gradient-descent calculated above. It can be seen that the blue lines seem to have far more points. This is due to x updating 5 times per epoch compared to the 1 as before. This leads to quicker convergence. The stochasticity can be seen by comparing the blue lines in each graph. The path taken to the minimum is slightly different in each iteration. This happens due to the batch used giving marginally different values for $f(x, N)$ and $f'(x, N)$.

b) iii)

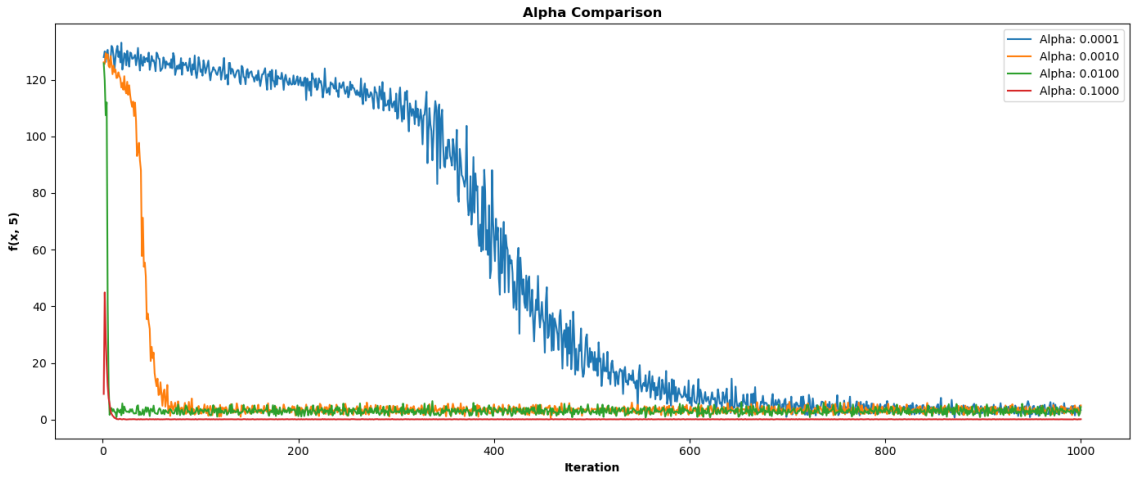
Using a fixed α of 0.01, I performed a few iterations of constant SGD for each batch_size. These returned the following



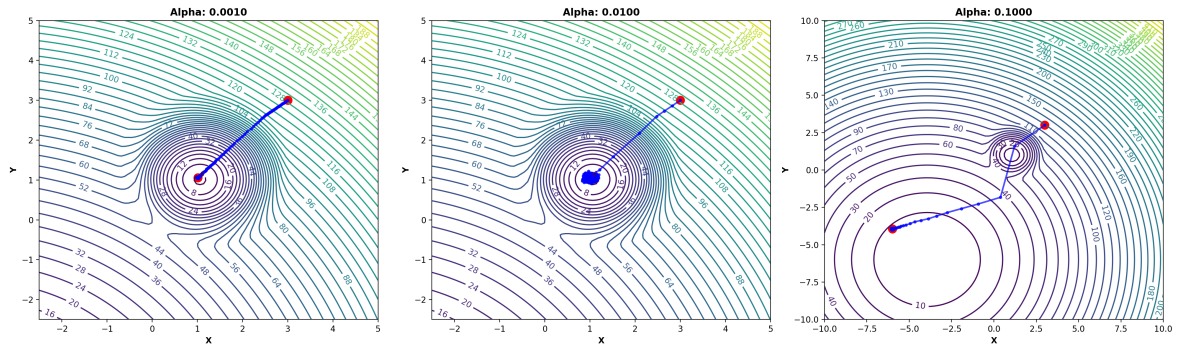
We can see that as the batch size increases, the noise decreases. This is simply an affect of the Law of Large numbers. The more samples used, the less variance is present. A trade-off of the higher variance is quicker convergence due to x updating after each iteration rather than each epoch. This stochasticity from different evaluating points leads to marginally different convergence points: batch_size: 5 finished at [0.9962, 0.9397]. batch_size: 9 finished at [0.9466, 0.9463]. batch_size: 15 finished at [1.0008, 0.9440]. batch_size: 25 finished at [1.0062, 0.9455].

b) iv)

I chose to perform analysis of $\alpha \in [0, 0001, 0.001, 0.01, 0.1]$. There were 2 noticeable differences in the results of varying alphas. The first was the speed of convergence. Higher alpha converged to a minimum far quicker than lower values. This is to be expected due to the bowl-shape of the loss function. We can see this in the plot below:



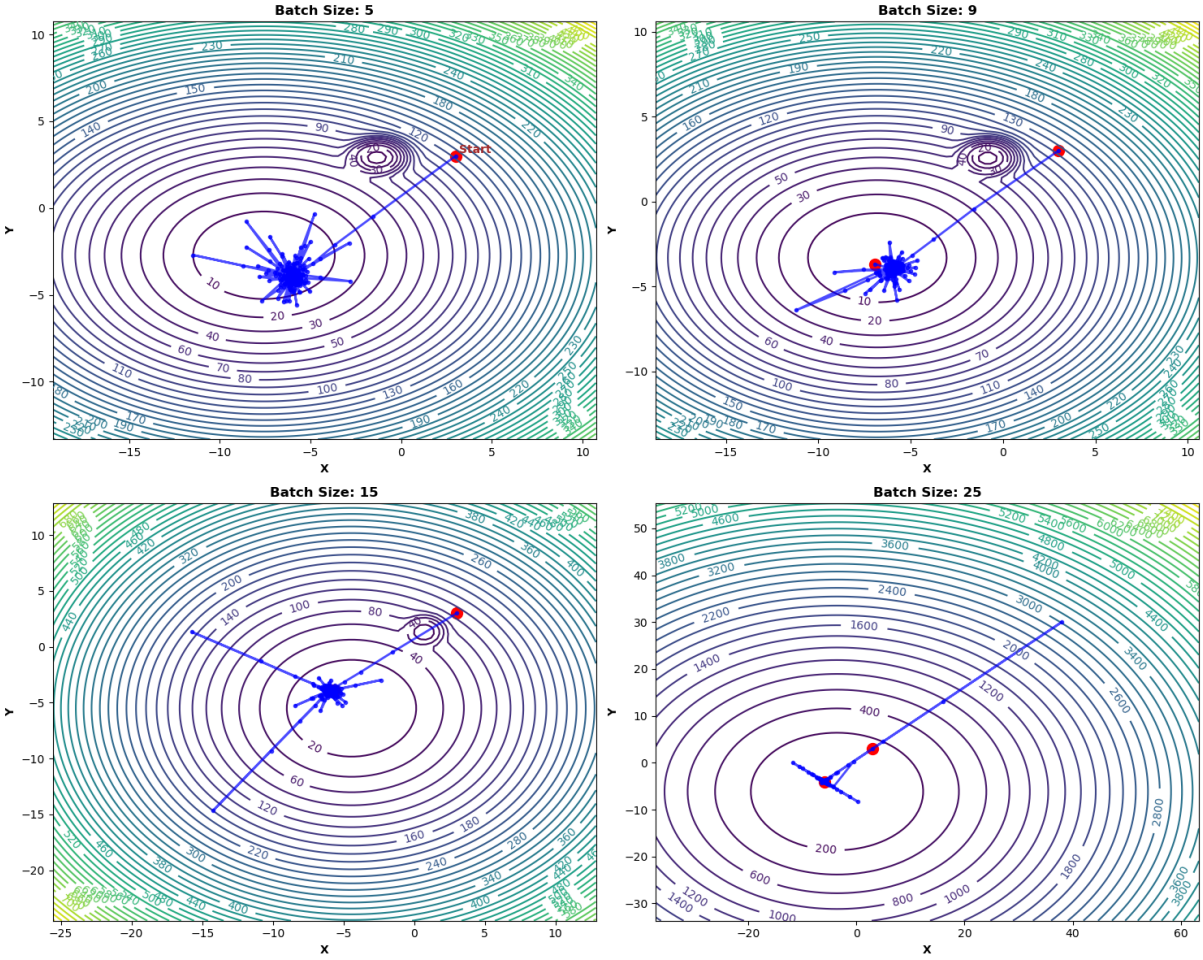
The second thing worth noting is the point of convergence of the different values of α. A larger value will lead the SGD function to overshoot the minimum in the valley and converge to the minimum value in the centre of the bowl, however, the smaller α values will settle in the valley. Although a larger step-size generally results in more noise, here this doesn't seem to be the case.



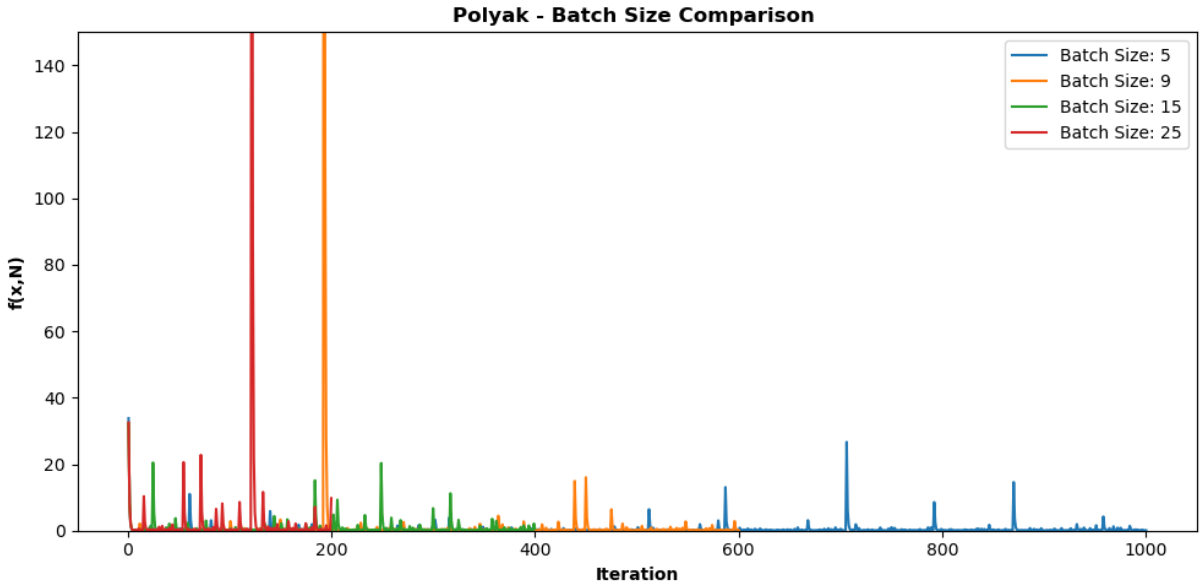
c) i)

Due to the large starting value of the function, the initial step-size overshoots the valley. Because of this, the polyak function converges to the minimum at the middle of the bowl-shaped surface.

The convergence of the polyak to this minimum is quite random with a lot of oscillation.



We can see that the stochasticity involved with a smaller batch size amplifies the oscillations although the increased number of point updates also contributes to this.

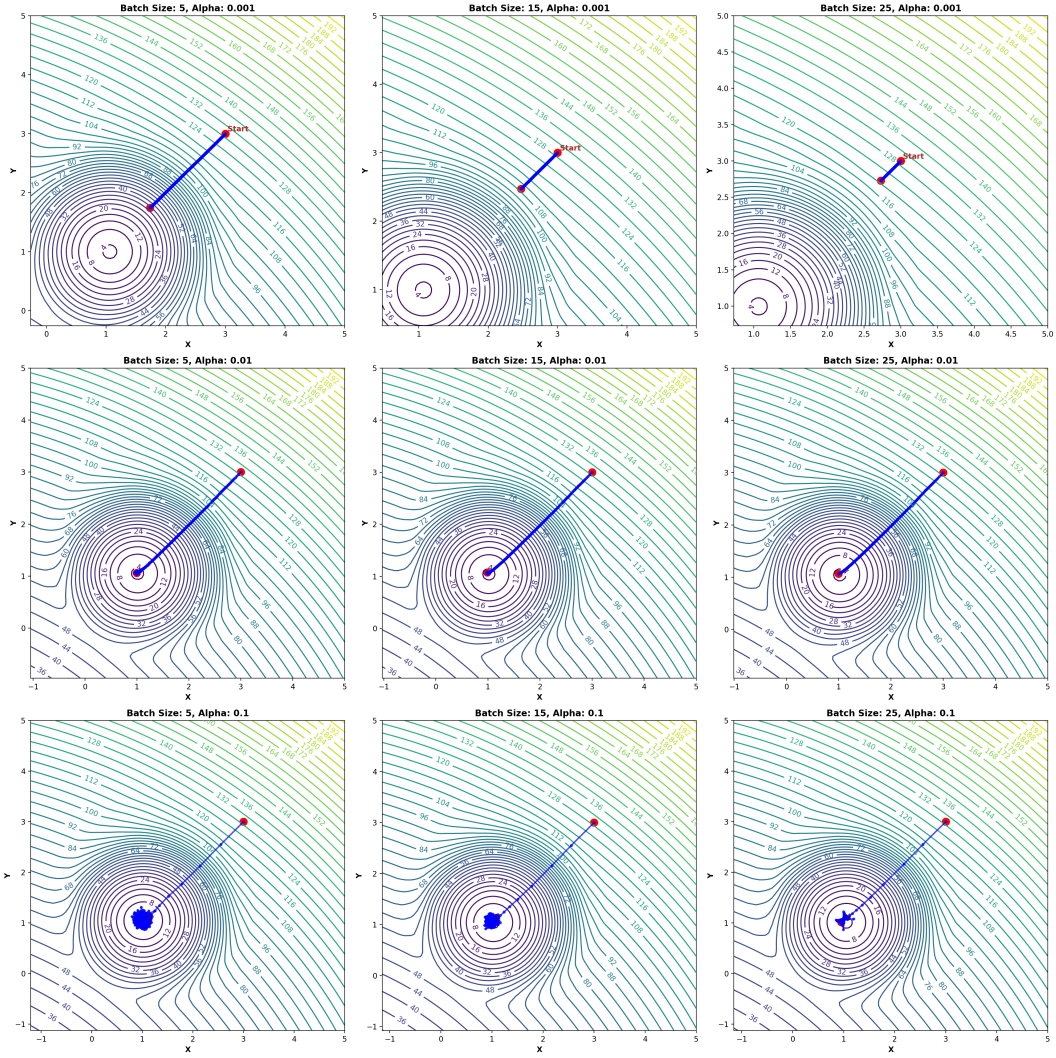


Although Polyak converges quickly, it is unstable due to the lack of momentum and therefore

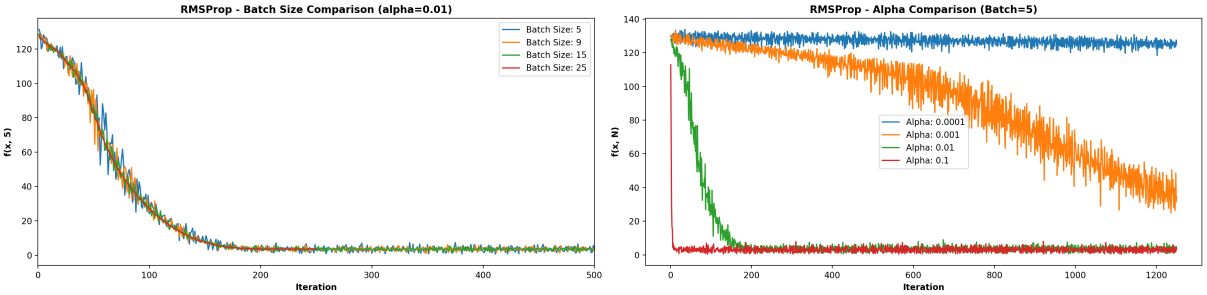
exposure to randomness in the SGD algorithm. The baseline algorithm from b performers better.

c) ii)

The RMSProp algorithm is more suited to SGD than Polyak. Since the gradient used to calculate α has components from past gradients, some of the randomness that comes with a smaller batch size is mitigated. I used a few different values of β but settled on 0.9.



We can see from these plots that again, a larger alpha converges faster, however it also comes with more oscillation around the minimum.

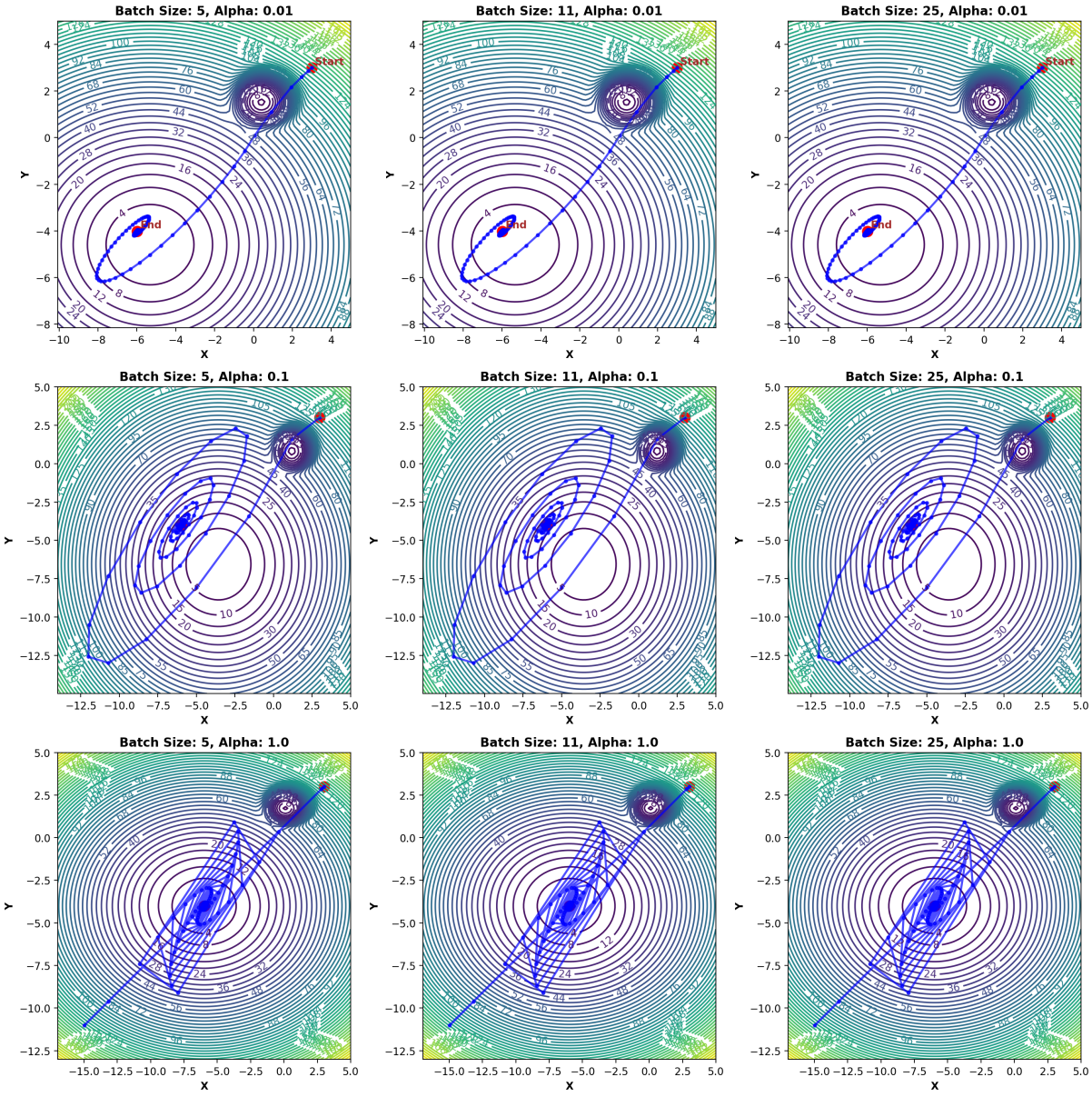


Looking at the different batch sizes, we can see that a lower batch size results in far quicker convergences, however, it also comes with increased noise. It is worth noting that in the plot of convergence with varying batch_size, the iterations for smaller batches are performed $\approx \frac{data}{batch\_size}$ quicker than training-data. The baseline still appears to perform better

c) iii)

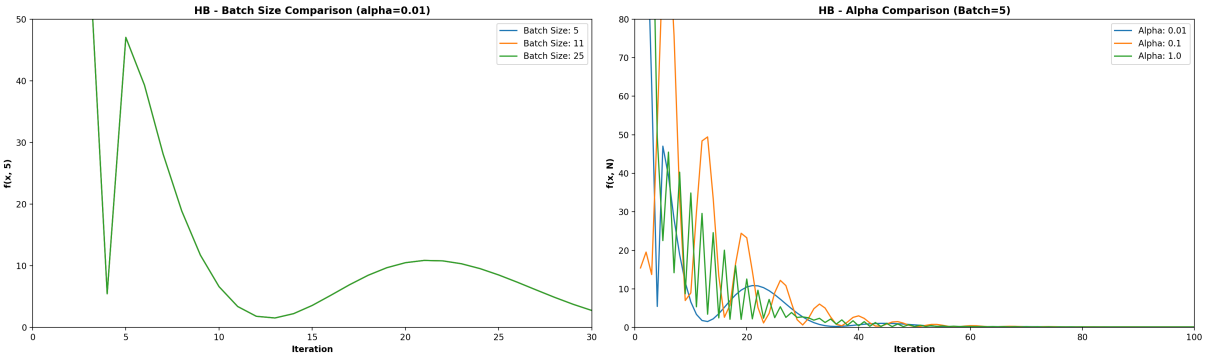
For the heavy-ball implementation, I chose a $\beta = 0.9$ as once again, it seemed to give the best results with everything else held constant. I chose to iterate over $\alpha \in [0.01, 0.1, 1]$ with batch sizes of 5, 11, 25.

Because heavy-ball has a momentum component, we can increase the learning rate alpha as long as the β value is kept relatively large.



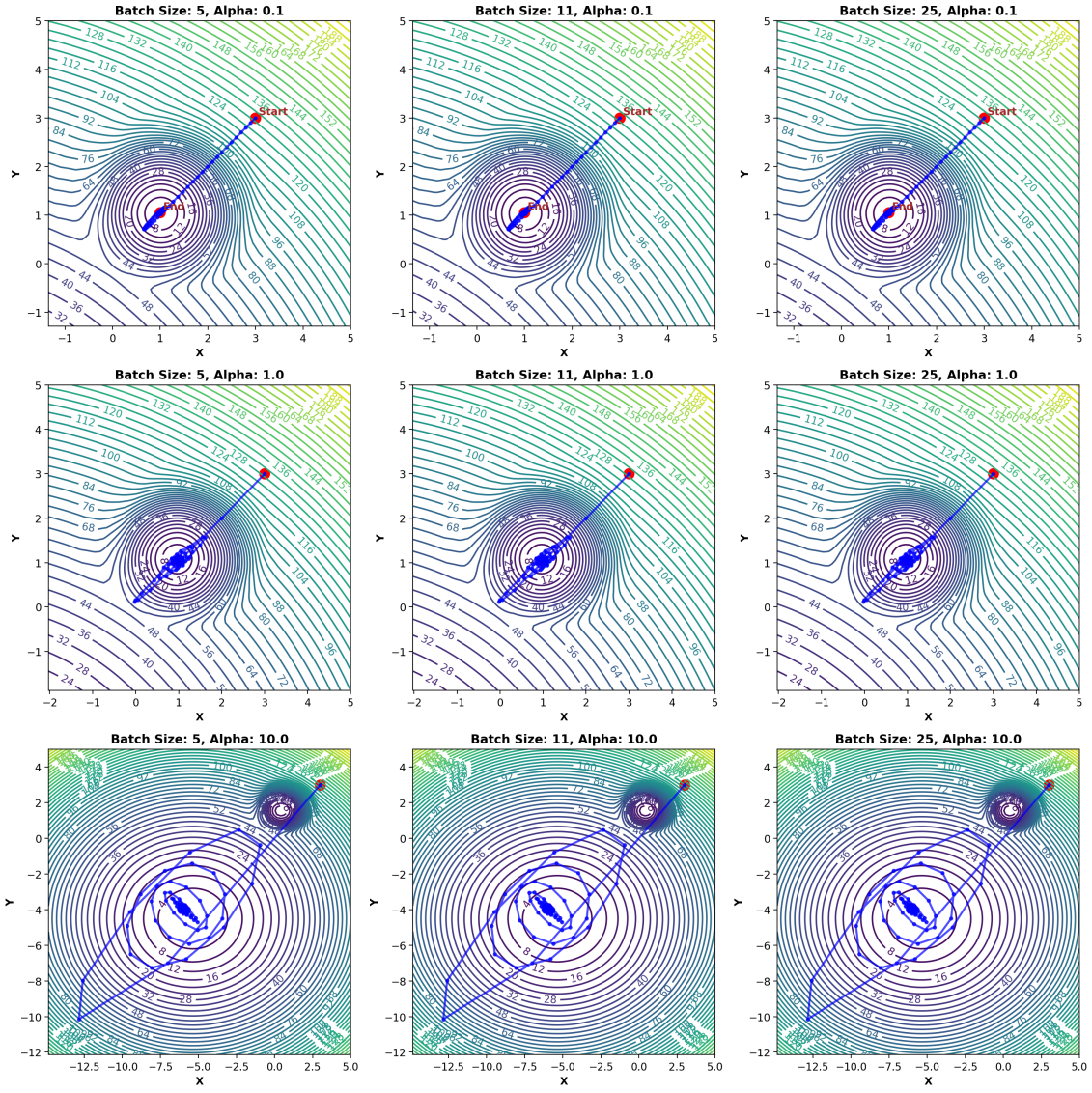
We can see that the momentum component for $\beta = 0.9$ means that the noise from different batch sizes is made almost irrelevant. In the left plot, there are 3 batch sizes however they

are almost indistinguishable. Increasing alpha increases the size of the oscillations to the point where convergence for each alpha is essentially the same.

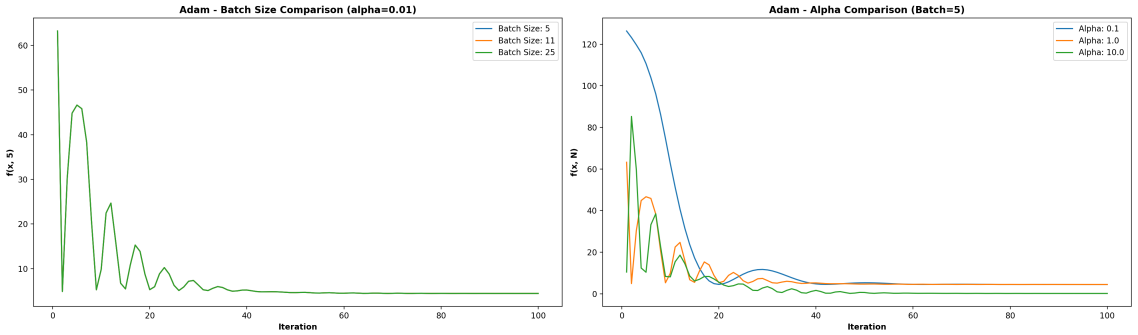


c) iv)

For the Adam algorithm, I chose $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Due to the momentum of this algorithm, it is not affected drastically by the noise associated with SGD



The speed of convergence becomes quicker with larger α, however, it also leads to further instability.



1 Code

```
import numpy as np

#Downloaded Functions
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(31*(z[0]**2+z[1]**2), (z[0]+7)**2+(z[1]+5)**2)
        count=count+1
    return y/count



def mini_batch(data, batch_size):
    batches = []
    np.random.shuffle(data)
    n = len(data)
    for i in np.arange(0, n, batch_size):
        batch = data[i:(i + batch_size)] if i + batch_size < n else
            data[i:n]
        batches.append(batch)
    return batches


def batch_finite_diff(f, pt, batch, h=1e-8):
    batch_dx = (f([pt[0] + h, pt[1]], batch) - f(pt, batch))/h
    batch_dy = (f([pt[0], pt[1] + h], batch) - f(pt, batch))/h
    return batch_dx, batch_dy


def constant(f, starting_point, data, alpha, tol, max_iters=1000,
    batch_size=32):
    x, y = starting_point[0], starting_point[1]
    path=[(x, y)]
    f_vals=[]

    for _ in range(max_iters):
        for batch in mini_batch(data, batch_size=batch_size):
            dx, dy = batch_finite_diff(f, [x,y], batch, h=1e-8)
            x -= alpha * dx
            y -= alpha * dy

```

```
            path.append((x,y))
            f_vals.append(f([x, y], batch))

        if np.sqrt(dx**2 + dy**2) < tol:
            break

    return path, f_vals




def Polyak(f, starting_point, data, epsilon=1e-8, f_star=0,
    max_iters=1000, batch_size=32):
    x, y = starting_point[0], starting_point[1]
    path=[(x, y)]
    f_vals=[]

    for _ in range(max_iters):
        for batch in mini_batch(data, batch_size=batch_size):
            dx, dy = batch_finite_diff(f, [x,y], batch, h=1e-8)
            alpha = (f([x, y], batch) - f_star)/(dx**2 + dy**2 +
                epsilon)
            x -= alpha * dx
            y -= alpha * dy

            path.append((x,y))
            f_vals.append(f([x, y], batch))

    return path, f_vals

def RMSProp(f, starting_point, data, alpha0, beta, epsilon=1e-8,
    max_iters=1000, batch_size=32):
    x, y = starting_point[0], starting_point[1]
    sum_dx = 0
    sum_dy = 0
    path = [(x, y)]
    f_vals = []

    for t in range(1, max_iters + 1):
        for batch in mini_batch(data, batch_size=batch_size):
            # Take gradients for current x, y
            dx, dy = batch_finite_diff(f, [x,y], batch, h=1e-8)

            # Update denominator for x and y using the following
            sum_dx = beta * sum_dx + (1 - beta) * dx**2
            sum_dy = beta * sum_dy + (1 - beta) * dy**2

            # Compute the step sizes for x and y
```

```
            step_size_x = alpha0 / np.sqrt(sum_dx + epsilon)
            step_size_y = alpha0 / np.sqrt(sum_dy + epsilon)

            # Update x and y
            x -= step_size_x * dx
            y -= step_size_y * dy

            path.append((x, y))
            f_vals.append(f([x, y], batch))

    return path, f_vals

def heavy_ball(f, starting_point, data, alpha, beta, max_iters=1000,
    tol=1e-5, batch_size=32, stopping=True):
    # Initialize x,y, momentum variables and path,f(x,y) lists
    x, y = starting_point[0], starting_point[1]
    ux, uy = 0, 0
    path = [(x, y)]
    f_vals = []

    for _ in range(max_iters):
        for batch in mini_batch(data, batch_size=batch_size):
            # Take gradients for current x, y
            dx, dy = batch_finite_diff(f, [x,y], batch, h=1e-8)

            # Update momentum variable with current and past momentum
            ux = beta * ux + alpha * dx
            uy = beta * uy + alpha * dy

            # Update x and y
            x -= ux
            y -= uy

            # Store the new values
            path.append((x, y))
            f_vals.append(f([x, y], batch))

        if f([x, y], batch) > 1e10:
                break
        # Check if the norm of the gradient is below the tolerance
        if np.sqrt(dx**2 + dy**2) < tol and stopping == True:
            break


    return path, f_vals
```

```
def adam(f, starting_point, data, alpha, beta1, beta2, epsilon=1e-6,
    max_iters=1000, tol=1e-5, batch_size=32,):
    # Initialize x,y, momentum variables and path,f(x,y) lists
    x, y = starting_point[0], starting_point[1]
    m_x, m_y = 0, 0
    v_x, v_y = 0, 0
    path = [(x, y)]
    f_vals = []

    for t in range(1, max_iters + 1):
        for batch in mini_batch(data, batch_size=batch_size):
            # Take gradients for current x, y
            dx, dy = batch_finite_diff(f, [x,y], batch, h=1e-8)

            # Update momentum variable for gradient
            m_x = beta1 * m_x + (1 - beta1) * dx
            m_y = beta1 * m_y + (1 - beta1) * dy
            # Update momentum variable for squared gradient
            v_x = beta2 * v_x + (1 - beta2) * dx**2
            v_y = beta2 * v_y + (1 - beta2) * dy**2
            # Compute bias-corrected estimates
            m_x_hat = m_x / (1 - beta1**t)
            m_y_hat = m_y / (1 - beta1**t)

            v_x_hat = v_x / (1 - beta2**t)
            v_y_hat = v_y / (1 - beta2**t)

            # Update variables
            x -= alpha * m_x_hat / (np.sqrt(v_x_hat) + epsilon)
            y -= alpha * m_y_hat / (np.sqrt(v_y_hat) + epsilon)

            path.append((x, y))
            f_vals.append(f([x, y], batch))
        if f([x, y], batch) > 1e10:
                break

    return path, f_vals


def const_GD(f, starting_point, alpha, training_data, max_iters=1000,
    tol=1e-6):
    x, y = starting_point[0], starting_point[1]
    path=[(x, y)]
    f_vals=[]
    for _ in range(max_iters):
        dx, dy = batch_finite_diff(f, [x,y], training_data, h=1e-8)

        x -= alpha * dx
```

```
        y -= alpha * dy

        path.append((x,y))
        f_vals.append(f([x, y], training_data))


    return path, f_vals


def SGD(optimizer, f, starting_point, data, batch_size=32,
    max_iters=1000, **kwargs):
    optimizer = optimizer.strip().lower()
    if optimizer not in ['constant', 'polyak', 'rmsprop', 'hb',
        'adam']:
        raise ValueError("Optimizer argument not valid")

    if optimizer == 'constant':
        alpha = kwargs.get('alpha', 0.01)
        tol = kwargs.get('tol', 1e-8)
        path, f_vals = constant(f=f, starting_point=starting_point,
            data=data, alpha=alpha, tol=tol, max_iters=max_iters,
            batch_size=batch_size)
        print("Stochastic-Gradient-Descent using Constant Optimiser")
        return path, f_vals

    elif optimizer == 'polyak':
        f_star = kwargs.get('f_star', 0.0)
        epsilon = kwargs.get('epsilon', 1e-8)

        path, f_vals = Polyak(f=f, starting_point=starting_point,
            data=data, f_star=f_star, epsilon=epsilon,
            max_iters=max_iters, batch_size=batch_size)
        print("Stochastic-Gradient-Descent using Polyak Optimiser")
        return path, f_vals

    elif optimizer == 'rmsprop':
        alpha0 = kwargs.get('alpha0', 0.01)
        beta = kwargs.get('beta', 0.9)
        epsilon = kwargs.get('epsilon', 1e-8)

        path, f_vals = RMSProp(f=f, starting_point=starting_point,
            data=data, alpha0=alpha0, beta=beta,epsilon=epsilon,
            max_iters=max_iters, batch_size=batch_size)
        print("Stochastic-Gradient-Descent using RMSProp Optimiser")
        return path, f_vals

    elif optimizer == 'hb':
        alpha = kwargs.get('alpha', 0.01)
```

```
        beta = kwargs.get('beta', 0.9)
        tol = kwargs.get('tol', 1e-5)

        path, f_vals = heavy_ball(f=f, starting_point=starting_point,
            data=data, alpha=alpha, beta=beta, max_iters=max_iters,
            tol=tol, batch_size=32)
        print("Stochastic-Gradient-Descent using Heavy-Ball Optimiser")
        return path, f_vals


    elif optimizer == 'adam':
        alpha = kwargs.get('alpha', 0.001)
        beta1 = kwargs.get('beta1', 0.9)
        beta2 = kwargs.get('beta2', 0.999)

        epsilon = kwargs.get('epsilon', 1e-8)
        tol = kwargs.get('tol', 1e-5)

        path, f_vals = adam(f=f, starting_point=starting_point,
            data=data, alpha=alpha, beta1=beta1, beta2=beta2,
            epsilon=epsilon, max_iters=max_iters, tol=tol,
            batch_size=32)
        print("Stochastic-Gradient-Descent using Adam Optimiser")
        return path, f_vals
```

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from optimizer_functions import *

from IPython.display import clear_output
```

```
training_data = generate_trainingdata()
x = np.linspace(-7.5, 7.5, 100)
y = np.linspace(-7.5, 7.5, 100)
z = []
for i in x:
    for j in y:
        z.append(f([i,j], training_data))

z = np.array(z).reshape(100, 100)
x, y = np.meshgrid(x, y)

fig = plt.figure(figsize=(18, 7))

ax1 = fig.add_subplot(121, projection='3d')
ax1.plot_wireframe(x, y, z, color='blue')
ax1.set_title('Wireframe Plot', fontsize=18, fontweight='bold')
```

```
ax1.set_xlabel('X Axis', fontweight='bold')
ax1.set_ylabel('Y Axis', fontweight='bold')
ax1.set_zlabel('f(x, T)', fontweight='bold')


ax2 = fig.add_subplot(122)
contour = ax2.contour(x, y, z, 50)
plt.clabel(contour, inline=1, fontsize=10)
ax2.set_title('Contour Plot', fontsize=18, fontweight='bold')
ax2.set_xlabel('X Axis', fontweight='bold')
ax2.set_ylabel('Y Axis', fontweight='bold')

plt.show()
```

```
path, f_vals = SGD('constant', f, starting_point=[3,3],
    data=training_data, alpha=0.01, max_iters=100, tol=-0.1,
    batch_size=len(training_data))
path_x, path_y = zip(*path)
path_z = f_vals

training_data = generate_trainingdata()
x = np.linspace(-6, 6, 100)
y = np.linspace(-6, 6, 100)
z = []
for i in x:
    for j in y:
        z.append(f([i,j], training_data))

z = np.array(z).reshape(100, 100)
x, y = np.meshgrid(x, y)

fig = plt.figure(figsize=(14, 7))

ax1 = fig.add_subplot(121, projection='3d')
ax1.plot_wireframe(x, y, z, color='blue', alpha=0.5)
ax1.scatter(path_x[0], path_y[0], path_z[0], color='green', s=50)
ax1.plot(path_x[0:-1], path_y[0:-1], path_z, color='red', linewidth=3,
    marker='o', markersize=5)
ax1.scatter(path_x[-1], path_y[-1], path_z[-1], color='black', s=50)
ax1.set_title('Wireframe Plot', fontsize=18, fontweight='bold')
ax1.set_xlabel('X Axis', fontweight='bold')
ax1.set_ylabel('Y Axis', fontweight='bold')
ax1.set_zlabel('f(x, T)', fontweight='bold')


ax2 = fig.add_subplot(122)
contour = ax2.contour(x, y, z, 50, cmap='viridis')
ax2.plot(path_x, path_y, color='blue', linewidth=3, marker='o',
```

```
    markersize=5)
ax2.scatter(path_x[0], path_y[0], color='brown', s=100)
ax2.scatter(path_x[-1], path_y[-1], color='brown', s=100)
ax2.annotate('Start', xy=(path_x[0], path_y[0]), xytext=(3, 3),
             textcoords='offset points', color='brown', weight='bold')
ax2.annotate('End', xy=(path_x[-1], path_y[-1]), xytext=(-30, -30),
             textcoords='offset points', color='brown', weight='bold')
plt.clabel(contour, inline=1, fontsize=10)
ax2.set_title('Contour Plot', fontsize=18, fontweight='bold')
ax2.set_xlabel('X Axis', fontweight='bold')
ax2.set_ylabel('Y Axis', fontweight='bold')

plt.show()
```

```
x = np.linspace(0, 4, 100)
y = np.linspace(0, 4, 100)
z = []
for i in x:
    for j in y:
        z.append(f([i,j], training_data))

z = np.array(z).reshape(100, 100)
x, y = np.meshgrid(x, y)

final_points = {i:[0.0,0.0] for i in range(1,7)}

fig, axes = plt.subplots(2, 3, figsize=(18, 10))

starting_point = [3, 3]
alpha = 0.01
batch_size = 5
max_iters = 100

for i in range(2):
    for j in range(3):
        ax = axes[i, j]
        path, f_vals = SGD('constant', f, starting_point=[3,3],
            data=training_data, alpha=alpha, max_iters=max_iters,
            tol=1e-4, batch_size=batch_size)
        true_path, true_f_vals = SGD('constant', f,
            starting_point=[3,3], data=training_data, alpha=0.01,
            max_iters=100, tol=-0.1, batch_size=len(training_data))


        path_x, path_y = zip(*path)
        true_path_x, true_path_y = zip(*true_path)

        final_points[3 * i + j + 1][0] += path_x[-1]
```

```
        final_points[3 * i + j + 1][1] += path_y[-1]


        contour = ax.contour(x, y, z, 50, cmap='viridis')
        ax.plot(path_x, path_y, color='blue', linewidth=2, marker='o',
            markersize=3, alpha=1)
        ax.plot(true_path_x, true_path_y, color='yellow', linewidth=2,
            marker='o', markersize=3, alpha=0.7)

        ax.scatter(path_x[0], path_y[0], color='red', s=100)
        if i == 0 and j == 0:
            ax.annotate('Start', xy=(path_x[0], path_y[0]), xytext=(3,
                3),
                  textcoords='offset points', color='brown',
                      weight='bold')



        ax.scatter(path_x[-1], path_y[-1], color='red', s=100)
        ax.clabel(contour, inline=1, fontsize=10)
        ax.set_title(f'SGD Iteration: {3 * i + j + 1}',
            fontweight='bold')
        ax.set_xlabel('X Axis')
        ax.set_ylabel('Y Axis')

plt.tight_layout()
plt.show()

for k, v in final_points.items():
    print(f"Iteration: {k} finished at [{v[0]:0.4f}, {v[1]:0.4f}]")
```

```
x = np.linspace(0, 4, 100)
y = np.linspace(0, 4, 100)
z = []
for i in x:
    for j in y:
        z.append(f([i,j], training_data))

z = np.array(z).reshape(100, 100)
x, y = np.meshgrid(x, y)

final_points = {i:[0.0,0.0] for i in range(1,7)}

fig, axes = plt.subplots(2, 3, figsize=(18, 10))

starting_point = [3, 3]
alpha = 0.01
batch_size = 5
```

```
max_iters = 100

for i in range(2):
    for j in range(3):
        ax = axes[i, j]
        path, f_vals = SGD('constant', f, starting_point=[3,3],
            data=training_data, alpha=alpha, max_iters=max_iters,
            tol=1e-4, batch_size=batch_size)
        true_path, true_f_vals = SGD('constant', f,
            starting_point=[3,3], data=training_data, alpha=0.01,
            max_iters=100, tol=-0.1, batch_size=len(training_data))


        path_x, path_y = zip(*path)
        true_path_x, true_path_y = zip(*true_path)

        final_points[3 * i + j + 1][0] += path_x[-1]
        final_points[3 * i + j + 1][1] += path_y[-1]


        contour = ax.contour(x, y, z, 50, cmap='viridis')
        ax.plot(path_x, path_y, color='blue', linewidth=2, marker='o',
            markersize=3, alpha=1)
        ax.plot(true_path_x, true_path_y, color='yellow', linewidth=2,
            marker='o', markersize=3, alpha=0.7)

        ax.scatter(path_x[0], path_y[0], color='red', s=100)
        if i == 0 and j == 0:
            ax.annotate('Start', xy=(path_x[0], path_y[0]), xytext=(3,
                3),
                  textcoords='offset points', color='brown',
                     weight='bold')



        ax.scatter(path_x[-1], path_y[-1], color='red', s=100)
        ax.clabel(contour, inline=1, fontsize=10)
        ax.set_title(f'SGD Iteration: {3 * i + j + 1}',
            fontweight='bold')
        ax.set_xlabel('X Axis')
        ax.set_ylabel('Y Axis')

plt.tight_layout()
plt.show()

for k, v in final_points.items():
    print(f"Iteration: {k} finished at [{v[0]:0.4f}, {v[1]:0.4f}]")
```

```
    starting_point = [3, 3]
alpha = 0.01
batch_sizes = np.logspace(0.75, np.log10(len(training_data)), num=4,
    base=10).astype(int)

fig, axes = plt.subplots(4, 3, figsize=(15, 20))

final_points = {i.astype(int):[0.0, 0.0] for i in batch_sizes}

for i in range(4):
    for j in range(3):
        ax = axes[i, j]
        batch_size = batch_sizes[i]

        path, f_vals = SGD('constant', f, starting_point,
            data=training_data,
                        alpha=alpha, max_iters=100, tol=1e-6,
                            batch_size=batch_size)


        path_x, path_y = zip(*path)
        path_z = f_vals
        final_points[batch_size][0] += path_x[-1]/3
        final_points[batch_size][1] += path_y[-1]/3


        contour = ax.contour(x, y, z, 50, cmap='viridis')
        ax.plot(path_x, path_y, color='blue', linewidth=2, marker='o',
            markersize=3, alpha=0.7)
        ax.scatter(path_x[0], path_y[0], color='red', s=100)
        if i == 0 and j == 0:
            ax.annotate('Start', xy=(path_x[0], path_y[0]), xytext=(3,
                3),
                  textcoords='offset points', color='brown',
                      weight='bold')

        ax.scatter(path_x[-1], path_y[-1], color='red', s=100)
        ax.clabel(contour, inline=1, fontsize=10)
        ax.set_title(f'Batch Size: {batch_size}, Iteration: {j+1}')
        ax.set_xlabel('X')
        ax.set_ylabel('Y')


        if i == 0 and j == 0:
            ax.legend()


```

```
plt.tight_layout()
plt.show()

for k, v in final_points.items():
    print(f"batch_size: {k} finished at [{v[0]:0.4f}, {v[1]:0.4f}]")
```

```
    starting_point = [3, 3]
alpha = 0.01
iterations = 5
batch_sizes = np.logspace(0.75, np.log10(len(training_data)), num=4,
   base=10).astype(int)

final_f_vals = {batch_size: [] for batch_size in batch_sizes}

for batch_size in batch_sizes:
    for _ in range(iterations):
        path, f_vals = SGD('constant', f, starting_point,
           data=training_data,
                           alpha=alpha, max_iters=100, tol=1e-6,
                              batch_size=batch_size)
        final_f_vals[batch_size].append(f_vals[-1])
avg_f_vals = {batch_size: np.mean(vals) for batch_size, vals in
   final_f_vals.items()}
std_f_vals = {batch_size: np.std(vals) for batch_size, vals in
   final_f_vals.items()}

fig, ax = plt.subplots(figsize=(10, 6))

batch_sizes_list = list(batch_sizes)
avg_vals_list = [avg_f_vals[bs] for bs in batch_sizes_list]
std_vals_list = [std_f_vals[bs] for bs in batch_sizes_list]

ax.errorbar(batch_sizes_list, avg_vals_list, yerr=std_vals_list,
   fmt='o', color='black', ecolor='blue', elinewidth=2, capsize=2)

ax.set_title('Average Final Function Value')
ax.set_xlabel('Batch Size')
ax.set_ylabel('Average Final Function Value')

plt.show()
```

```
starting_point = [3, 3]
alphas = np.logspace(-3, -1, num=3, base=10)
batch_size = 5
max_iters = 200

fig, axes = plt.subplots(1, 3, figsize=(20, 6), dpi=200)

```

```
final_points = {alpha: [0.0, 0.0] for alpha in alphas}

for i, alpha in enumerate(alphas):
    if alpha < 0.05:
        x_range = np.linspace(-2.5, 5, 100)
        y_range = np.linspace(-2.5, 5, 100)
    else:
        x_range = np.linspace(-10, 10, 100)
        y_range = np.linspace(-10, 10, 100)

    z = [f([i, j], training_data) for i in x_range for j in y_range]
    z = np.array(z).reshape(100, 100)
    x, y = np.meshgrid(x_range, y_range)

    ax = axes[i]
    path, f_vals = SGD('constant', f, starting_point,
       data=training_data,
                       alpha=alpha, max_iters=max_iters, tol=1e-6,
                          batch_size=batch_size)
    path_x, path_y = zip(*path)
    final_points[alpha][0] += path_x[-1]
    final_points[alpha][1] += path_y[-1]

    contour = ax.contour(x, y, z, 50, cmap='viridis')
    ax.plot(path_x, path_y, color='blue', linewidth=2, marker='o',
       markersize=3, alpha=0.7)
    ax.scatter(path_x[0], path_y[0], color='red', s=100)
    if i == 0 and j == 0:
        ax.annotate('Start', xy=(path_x[0], path_y[0]), xytext=(3, 3),
                    textcoords='offset points', color='brown',
                       weight='bold')

    ax.scatter(path_x[-1], path_y[-1], color='red', s=100)
    ax.clabel(contour, inline=1, fontsize=10)
    ax.set_title(f'Alpha: {alpha:.4f}', fontweight='bold')
    ax.set_xlabel('X', fontweight='bold')
    ax.set_ylabel('Y', fontweight='bold')

plt.tight_layout()
plt.show()

for k, v in final_points.items():
    print(f"Alpha: {k}, finished at [{v[0]:0.4f}, {v[1]:0.4f}]")
```

```
starting_point = [3, 3]
alphas = np.logspace(-4, -1, num=4, base=10)
batch_sizes = np.logspace(0.75, np.log10(len(training_data)), num=4,
   base=10).astype(int)
```

```
max_iters = 200

fig, axs = plt.subplots(1, 1, figsize=(14, 6))

for alpha in alphas:
    _, f_vals = SGD('constant', f, starting_point, data=training_data,
                    alpha=alpha, max_iters=max_iters, tol=1e-6,
                       batch_size=5)
    iterations = list(range(1, len(f_vals) + 1))
    axs.plot(iterations, f_vals, label=f'Alpha: {alpha:.4f}')

axs.set_title('Alpha Comparison', fontweight='bold')
axs.set_xlabel('Iteration', fontweight='bold')
axs.set_ylabel('f(x, 5)', fontweight='bold')
axs.legend()


plt.tight_layout()
plt.show()
```

```
training_data = generate_trainingdata()

starting_point = [3, 3]
alpha = 0.01
batch_sizes = np.logspace(0.75, np.log10(len(training_data)), num=4,
   base=10).astype(int)
max_iters = 100

fig, axes = plt.subplots(2, 2, figsize=(15, 12))
axes = axes.flatten()

final_points = {batch: [0.0, 0.0] for batch in batch_sizes}

for i, batch_size in enumerate(batch_sizes):
    ax = axes[i]
    path, f_vals = SGD('polyak', f=f, starting_point=starting_point,
       data=training_data, epsilon=1e-6, max_iters=max_iters,
       batch_size=batch_size)
    path_x, path_y = zip(*path)
    final_points[batch_size][0] += path_x[-1]
    final_points[batch_size][1] += path_y[-1]

    size = max((max(path_x) - min(path_x)), (max(path_y) -
       min(path_y))) + 1

    x = np.linspace(min(path_x) - size / 2, max(path_x) + size / 2,
       100)
    y = np.linspace(min(path_y) - size / 2, max(path_y) + size / 2,
```

```
        100)
    z = [f([a, b], training_data) for a in x for b in y]
    z = np.array(z).reshape(100, 100)
    x, y = np.meshgrid(x, y)

    contour = ax.contour(x, y, z, 50, cmap='viridis')
    ax.plot(path_x, path_y, color='blue', linewidth=2, marker='o',
        markersize=3, alpha=0.7)
    ax.scatter(path_x[0], path_y[0], color='red', s=100)

    ax.annotate('Start', xy=(path_x[0], path_y[0]), xytext=(3, 3),
                textcoords='offset points', color='brown',
                    weight='bold')

    ax.scatter(path_x[-1], path_y[-1], color='red', s=100)
    ax.clabel(contour, inline=1, fontsize=10)
    ax.set_title(f'Batch Size: {batch_size}', fontweight='bold')
    ax.set_xlabel('X', fontweight='bold')
    ax.set_ylabel('Y', fontweight='bold')

plt.tight_layout()
plt.show()

for k, v in final_points.items():
    print(f"Batch size: {k}, finished at [{v[0]:0.4f}, {v[1]:0.4f}]")
```

```
starting_point = [3, 3]
alphas = np.logspace(-4, -1, num=4, base=10)
batch_sizes = np.logspace(0.75, np.log10(len(training_data)), num=4,
    base=10).astype(int)
max_iters = 200

fig, axes = plt.subplots(1, 1, figsize=(10, 5))


for batch_size in batch_sizes:
    _, f_vals = SGD('polyak', f=f, starting_point=starting_point,
        data=training_data, epsilon=1e-6, max_iters=max_iters,
        batch_size=batch_size)
    iterations = list(range(len(f_vals)))
    plt.plot(iterations, f_vals, label=f'Batch Size: {batch_size}')

plt.title('Polyak - Batch Size Comparison ' , fontweight='bold')
plt.xlabel('Iteration', fontweight='bold')
plt.ylim(0, 150)
plt.ylabel('f(x,N)', fontweight='bold')
plt.legend()

```

```
plt.tight_layout()
plt.show()
```

```
training_data = generate_trainingdata()

starting_point = [3, 3]
alphas = np.logspace(-3, -1, num=3, base=10)
beta = 0.95

batch_sizes = np.linspace(5,25, 3 ).astype(int)
max_iters = 250

fig, axes = plt.subplots(3, 3, figsize=(20, 20), dpi=200)

final_points = {batch: [0.0, 0.0] for batch in batch_sizes}

for j, batch_size in enumerate(batch_sizes):
    for i, alpha0 in enumerate(alphas):
        ax = axes[i, j]
        path, f_vals = SGD('rmsprop',f=f,
           starting_point=starting_point, data=training_data,
           alpha0=alpha0, beta=beta, epsilon=1e-6,
           max_iters=max_iters, batch_size=batch_size)
        path_x, path_y = zip(*path)
        final_points[batch_size][0] += path_x[-1]
        final_points[batch_size][1] += path_y[-1]

        size = max((max(path_x) - min(path_x)), (max(path_y) -
           min(path_y))) + 2

        x = np.linspace(min(path_x)-2, max(path_x)+2, 100)
        y = np.linspace(min(path_y)-2, max(path_y)+2, 100)
        z = [f([a, b], training_data) for a in x for b in y]
        z = np.array(z).reshape(100, 100)
        x, y = np.meshgrid(x, y)

        contour = ax.contour(x, y, z, 50, cmap='viridis')
        ax.plot(path_x, path_y, color='blue', linewidth=2, marker='o',
           markersize=3, alpha=0.7)
        ax.scatter(path_x[0], path_y[0], color='red', s=100)

        if i == 0:
            ax.annotate('Start', xy=(path_x[0], path_y[0]), xytext=(3,
               3),
                        textcoords='offset points', color='brown',
                           weight='bold')

        ax.scatter(path_x[-1], path_y[-1], color='red', s=100)
```

```
        ax.clabel(contour, inline=1, fontsize=10)
        ax.set_title(f'Batch Size: {batch_size}, Alpha: {alpha0}',
            fontweight='bold')
        ax.set_xlabel('X', fontweight='bold')
        ax.set_ylabel('Y', fontweight='bold')

plt.tight_layout()
plt.show()

for k, v in final_points.items():
    print(f"Batch size: {k}, finished at [{v[0]:0.4f}, {v[1]:0.4f}]")
```

```
starting_point = [3, 3]
alphas = np.logspace(-4,-1, num=4, base=10)
beta = 0.95
batch_sizes = np.logspace(0.75, np.log10(len(training_data)), num=4,
    base=10).astype(int)
max_iters = 250


fig, axs = plt.subplots(1, 2, figsize=(20, 5), dpi=200)


for batch_size in batch_sizes:
    _, f_vals = SGD('rmsprop',f=f, starting_point=starting_point,
        data=training_data, alpha0=0.01, beta=beta, epsilon=1e-6,
        max_iters=max_iters, batch_size=batch_size)
    iterations = list(range(1, len(f_vals) + 1))
    axs[0].plot(iterations, f_vals, label=f'Batch Size: {batch_size}')

axs[0].set_title('RMSProp - Batch Size Comparison (alpha=0.01)',
    fontweight='bold')
axs[0].set_xlabel('Iteration', fontweight='bold')
axs[0].set_xlim(0, 500)
axs[0].set_ylabel('f(x, 5)', fontweight='bold')
axs[0].legend()


for alpha in alphas:
    _, f_vals = SGD('rmsprop',f=f, starting_point=starting_point,
        data=training_data, alpha0=alpha, beta=beta, epsilon=1e-6,
        max_iters=max_iters, batch_size=5)
    iterations = list(range(1, len(f_vals) + 1))
    axs[1].plot(iterations, f_vals, label=f'Alpha: {alpha}')

axs[1].set_title('RMSProp - Alpha Comparison (Batch=5)',
    fontweight='bold')
axs[1].set_xlabel('Iteration', fontweight='bold')
```

```
axs[1].set_ylabel('f(x, N)', fontweight='bold')
axs[1].legend()

plt.tight_layout()
plt.show()
```

```
training_data = generate_trainingdata()

starting_point = [3, 3]
alphas = np.logspace(-2, 0, num=3, base=10)
beta = 0.9

batch_sizes = np.logspace(0.75, np.log10(len(training_data)), num=3,
    base=10).astype(int)
max_iters = 250

fig, axes = plt.subplots(3, 3, figsize=(15, 15), dpi=200)

final_points = {batch: {
    alpha: [0.0, 0.0] for alpha in alphas
} for batch in batch_sizes}

for j, batch_size in enumerate(batch_sizes):
    for i, alpha in enumerate(alphas):
        ax = axes[i, j]
        path, f_vals = SGD('hb',f=f, starting_point=starting_point,
            data=training_data, alpha=alpha, beta=beta, epsilon=1e-6,
            max_iters=max_iters, tol=-0.1, batch_size=batch_size)
        path_x, path_y = zip(*path)
        final_points[batch_size][alpha][0] = path_x[-1]
        final_points[batch_size][alpha][1] = path_y[-1]

        size = max((max(path_x) - min(path_x)), (max(path_y) -
            min(path_y))) + 2

        x = np.linspace(min(path_x)-2, max(path_x)+2, 100)
        y = np.linspace(min(path_y)-2, max(path_y)+2, 100)
        z = [f([a, b], training_data) for a in x for b in y]
        z = np.array(z).reshape(100, 100)
        x, y = np.meshgrid(x, y)

        contour = ax.contour(x, y, z, 50, cmap='viridis')
        ax.plot(path_x, path_y, color='blue', linewidth=2, marker='o',
            markersize=3, alpha=0.7)
        ax.scatter(path_x[0], path_y[0], color='red', s=100)

        if i == 0:
            ax.annotate('Start', xy=(path_x[0], path_y[0]), xytext=(3,
```

```
                3),
38                          textcoords='offset points', color='brown',
                               weight='bold')
39              ax.annotate('End', xy=(path_x[-1], path_y[-1]), xytext=(3,
                3),
40                          textcoords='offset points', color='brown',
                               weight='bold')
41
42          ax.scatter(path_x[-1], path_y[-1], color='red', s=100)
43          ax.clabel(contour, inline=1, fontsize=10)
44          ax.set_title(f'Batch Size: {batch_size}, Alpha: {alpha}',
                fontweight='bold')
45          ax.set_xlabel('X', fontweight='bold')
46          ax.set_ylabel('Y', fontweight='bold')
47
48  plt.tight_layout()
49  plt.show()
50
51  for k, a in final_points.items():
52      for x, v in a.items():
53          print(f"Batch size: {k}, Alpha: {x}\t finished at
                [{v[0]:0.4f}, {v[1]:0.4f}]")
```

```
1   starting_point = [3, 3]
2   alphas = np.logspace(-2, 0, num=3, base=10)
3   beta = 0.9
4   batch_sizes = np.logspace(0.75, np.log10(len(training_data)), num=3,
        base=10).astype(int)
5   max_iters = 250
6
7   fig, axs = plt.subplots(1, 2, figsize=(20, 6), dpi=200)
8
9   for batch_size in batch_sizes:
10      _, f_vals = SGD('hb',f=f, starting_point=starting_point,
            data=training_data, alpha=0.01, beta=beta, epsilon=1e-6,
            max_iters=max_iters, tol=-0.1, batch_size=batch_size)
11      iterations = list(range(1, len(f_vals) + 1))
12      axs[0].plot(iterations, f_vals, label=f'Batch Size: {batch_size}' )
13
14  axs[0].set_title('HB - Batch Size Comparison (alpha=0.01)',
        fontweight='bold')
15  axs[0].set_xlabel('Iteration', fontweight='bold')
16  axs[0].set_xlim(0,30)
17  axs[0].set_ylim(0,50)
18  axs[0].set_ylabel('f(x, 5)', fontweight='bold')
19  axs[0].legend()
20
21  for alpha in alphas:
```

```
    _, f_vals = SGD('hb',f=f, starting_point=starting_point,
        data=training_data, alpha=alpha, beta=beta, epsilon=1e-6,
        max_iters=max_iters, tol=-0.1, batch_size=5)
    iterations = list(range(1, len(f_vals) + 1))
    axs[1].plot(iterations, f_vals, label=f'Alpha: {alpha}')

axs[1].set_title('HB - Alpha Comparison (Batch=5)', fontweight='bold')
axs[1].set_xlabel('Iteration', fontweight='bold')
axs[1].set_ylabel('f(x, N)', fontweight='bold')
axs[1].set_xlim(0,100)
axs[1].set_ylim(0,80)
axs[1].legend()

plt.tight_layout()
plt.show()
```

```
starting_point = [3, 3]
alphas = np.logspace(-2, 0, num=3, base=10)
beta = 0.9
batch_sizes = np.logspace(0.75, np.log10(len(training_data)), num=3,
    base=10).astype(int)
max_iters = 250

fig, axs = plt.subplots(1, 2, figsize=(20, 6), dpi=200)

for batch_size in batch_sizes:
    _, f_vals = SGD('hb',f=f, starting_point=starting_point,
        data=training_data, alpha=0.01, beta=beta, epsilon=1e-6,
        max_iters=max_iters, tol=-0.1, batch_size=batch_size)
    iterations = list(range(1, len(f_vals) + 1))
    axs[0].plot(iterations, f_vals, label=f'Batch Size: {batch_size}' )

axs[0].set_title('HB - Batch Size Comparison (alpha=0.01)',
    fontweight='bold')
axs[0].set_xlabel('Iteration', fontweight='bold')
axs[0].set_xlim(0,30)
axs[0].set_ylim(0,50)
axs[0].set_ylabel('f(x, 5)', fontweight='bold')
axs[0].legend()

for alpha in alphas:
    _, f_vals = SGD('hb',f=f, starting_point=starting_point,
        data=training_data, alpha=alpha, beta=beta, epsilon=1e-6,
        max_iters=max_iters, tol=-0.1, batch_size=5)
    iterations = list(range(1, len(f_vals) + 1))
    axs[1].plot(iterations, f_vals, label=f'Alpha: {alpha}')

axs[1].set_title('HB - Alpha Comparison (Batch=5)', fontweight='bold')
```

```
axs[1].set_xlabel('Iteration', fontweight='bold')
axs[1].set_ylabel('f(x, N)', fontweight='bold')
axs[1].set_xlim(0,100)
axs[1].set_ylim(0,0)
axs[1].legend()

plt.tight_layout()
plt.show()
```

```
training_data = generate_trainingdata()

starting_point = [3, 3]
alphas = np.logspace(-1, 1, num=3, base=10)
beta1 = 0.9
beta2 = 0.999

batch_sizes = np.logspace(0.75, np.log10(len(training_data)), num=3,
    base=10).astype(int)
max_iters = 250

fig, axes = plt.subplots(3, 3, figsize=(15, 15), dpi=200)

final_points = {batch: {
    alpha: [0.0, 0.0] for alpha in alphas
} for batch in batch_sizes}

for j, batch_size in enumerate(batch_sizes):
    for i, alpha in enumerate(alphas):
        ax = axes[i, j]
        path, f_vals = SGD('adam',f=f, starting_point=starting_point,
            data=training_data, alpha=alpha, beta1=beta1, beta2=beta2,
            epsilon=1e-6,  max_iters=max_iters, tol=-0.1,
            batch_size=batch_size)
        path_x, path_y = zip(*path)
        final_points[batch_size][alpha][0] = path_x[-1]
        final_points[batch_size][alpha][1] = path_y[-1]

        size = max((max(path_x) - min(path_x)), (max(path_y) -
            min(path_y))) + 2

        x = np.linspace(min(path_x)-2, max(path_x)+2, 100)
        y = np.linspace(min(path_y)-2, max(path_y)+2, 100)
        z = [f([a, b], training_data) for a in x for b in y]
        z = np.array(z).reshape(100, 100)
        x, y = np.meshgrid(x, y)

        contour = ax.contour(x, y, z, 50, cmap='viridis')
        ax.plot(path_x, path_y, color='blue', linewidth=2, marker='o',
```

```
            markersize=3, alpha=0.7)
        ax.scatter(path_x[0], path_y[0], color='red', s=100)

        if i == 0:
            ax.annotate('Start', xy=(path_x[0], path_y[0]), xytext=(3,
                3),
                        textcoords='offset points', color='brown',
                            weight='bold')
            ax.annotate('End', xy=(path_x[-1], path_y[-1]), xytext=(3,
                3),
                        textcoords='offset points', color='brown',
                            weight='bold')

        ax.scatter(path_x[-1], path_y[-1], color='red', s=100)
        ax.clabel(contour, inline=1, fontsize=10)
        ax.set_title(f'Batch Size: {batch_size}, Alpha: {alpha}',
            fontweight='bold')
        ax.set_xlabel('X', fontweight='bold')
        ax.set_ylabel('Y', fontweight='bold')

plt.tight_layout()
plt.show()

for k, a in final_points.items():
    for x, v in a.items():
        print(f"Batch size: {k}, Alpha: {x}\t finished at
            [{v[0]:0.4f}, {v[1]:0.4f}]")
```

```
    starting_point = [3, 3]
alphas = np.logspace(-1, 1, num=3, base=10)
beta1 = 0.9
beta2 = 0.999
batch_sizes = np.logspace(0.75, np.log10(len(training_data)), num=3,
    base=10).astype(int)
max_iters = 100


fig, axs = plt.subplots(1, 2, figsize=(20, 6), dpi=200)


for batch_size in batch_sizes:
    _, f_vals = SGD('adam',f=f, starting_point=starting_point,
        data=training_data, alpha=1, beta1=beta1, beta2=beta2,
        epsilon=1e-6,  max_iters=max_iters, tol=-0.1,
        batch_size=batch_size)
    iterations = list(range(1, len(f_vals) + 1))
    axs[0].plot(iterations, f_vals, label=f'Batch Size: {batch_size}',
        )
```

```
16
17 axs[0].set_title('Adam - Batch Size Comparison (alpha=0.01)',
       fontweight='bold')
18 axs[0].set_xlabel('Iteration', fontweight='bold')
19 axs[0].set_ylabel('f(x, 5)', fontweight='bold')
20 axs[0].legend()
21
22
23 for alpha in alphas:
24     _, f_vals = SGD('adam',f=f, starting_point=starting_point,
           data=training_data, alpha=alpha, beta1=beta1, beta2=beta2,
           epsilon=1e-6,  max_iters=max_iters, tol=-0.1, batch_size=5)
25     iterations = list(range(1, len(f_vals) + 1))
26     axs[1].plot(iterations, f_vals, label=f'Alpha: {alpha}')
27
28 axs[1].set_title('Adam - Alpha Comparison (Batch=5)',
       fontweight='bold')
29 axs[1].set_xlabel('Iteration', fontweight='bold')
30 axs[1].set_ylabel('f(x, N)', fontweight='bold')
31 axs[1].legend()
32
33 plt.tight_layout()
34 plt.show()
```