Question 1: Download functions from link and obtain function expression and derivative for using Sympy

I have downloaded the functions from the link provided.

```
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(24*(z[0]**2+z[1]**2), (z[0]+3)**2+(z[1]+3)**2)
    count=count+1
  return y/count
```

Question (a)(i): Implements mini-batch Stochastic Gradient Descent (SGD) with various step sizes

Stochastic Gradient Descent is a variant of Gradient Descent technique. It is an optimisation technique which is used to **minimise the cost functions** in machine learning algorithms. Cost function is a function which is used **to analyse the adaptability and generalisation of the model** over training data.

In **gradient descent**, the cost is computed by calculating the **gradients of the cost function for entire training set of data** and then cost is minimised over a set of iterations. If the dataset is large, this operation can be **resource-intensive as well as time consuming**. **Mini-batch Stochastic Gradient Descent** is a variant of Gradient Descent where cost is minimised by **calculating gradients for a small subset of the training data**. These mini-batches are then used to minimize the cost functions for a set of iterations.

The primary reason for using this technique is that mini-batches provide unique advantage over using full dataset. Mini batches aid the SGD to **randomly explore various parts of the cost function**, which can help it to **escape from local minima** in the cost function and **search global minima.**

As part of this assignment, we were provided with the cost function, mentioned above, to be minimised. I have created a python class to execute the SGD that utilises this function. Below are the details:

```
def execute_stochastic_gradient_descent(self, gradient_descent_type = 'Constant', alpha = 0.1, beta = 0.9, beta2 = 0.999, batch_size = 10) :
    np.random.shuffle(self.train_data)
    for j in range(0, len(self.train_data), batch_size) :

      if (j + batch_size) > len(self.train_data) :
        continue

      if gradient_descent_type == 'Constant Step Size' :
        self.x1, self.x2 = self.execute_constant_size_gradient_descent(self.train_data[j : (j + batch_size)], alpha)

      elif gradient_descent_type == 'Polyak' :
        self.x1, self.x2 = self.execute_polyak_gradient_descent(self.train_data[j : (j + batch_size)])

      elif gradient_descent_type == 'RMSProp':
        self.x1, self.x2 = self.execute_rmsprop_gradient_descent(self.train_data[j : (j + batch_size)], alpha, beta)

      elif gradient_descent_type == 'HeavyBall':
        self.x1, self.x2 = self.execute_heavyball_gradient_descent(self.train_data[j : (j + batch_size)], alpha, beta)

      elif gradient_descent_type == 'Adam':
        self.x1, self.x2 = self.execute_adam_gradient_descent(self.train_data[j : (j + batch_size)], alpha, beta, beta2)


    self.x1_values = np.append(self.x1_values, [self.x1], axis = 0)
    self.x2_values = np.append(self.x2_values, [self.x2], axis = 0)
    self.function_values = np.append(self.function_values, [self.f([self.x1, self.x2], self.train_data)])

    return self.x1_values, self.x2_values, self.function_values
```

This function takes the following parameters as input:

- **gradient_descent_type:** This parameter is used to define the type of step to be used in gradient descent algorithm. These are Constant, Polyak, RMSPRop, Heavyball and Adam. Choice of step will affect the overall optimisation of the cost function as these algorithms use different techniques to calculate step using gradients.
- **alpha, beta, beta2:** These parameters are used in various step algorithms.
- **batch_size:** Mini-batch size that is to be used in SGD

First of all. I **shuffle the training data** in each iteration. Shuffling training data is important as **it introduces noise in the data** and we can observe the efficiency of algorithm in minimising the cost more effectively. Then, I **split the training data into mini-batches** as passed in the input argument. Finally, on the basis of **type of step algorithm** to be implemented, I pass this mini-batch of training data to compute the gradients and **update x1, x2, and f(x1, x2) values**. These values are stored in an array and returned from the function for each iteration.

Details of the various step algorithms were submitted as part of previous assignment.

```
def get_batch_derivative(self, x1, x2, sample_batch) :
    sum_derivative_x1_value = 0
    sum_derivative_x2_value = 0

    for w1, w2 in sample_batch :
      derivative_x1_value, derivative_x2_value = self.get_derivative_value(x1, w1, x2, w2)
      sum_derivative_x1_value += derivative_x1_value
      sum_derivative_x2_value += derivative_x2_value

    batch_derivative_x1_value = sum_derivative_x1_value / len(sample_batch)
    batch_derivative_x2_value = sum_derivative_x2_value / len(sample_batch)

    return batch_derivative_x1_value, batch_derivative_x2_value
```

This function is used to **calculate approximate derivative for a batch of training data.** Previously, we used to calculate derivative for all the training points independently to update step. In this function, we take initial points x1, x2, and mini-batch as input and we **are approximating the derivative over a set of training points by calculating the average of derivative over these points**.

Question a(ii): Plot a wireframe and a contour plot of 'f' for N = T

I have used matplot library to generate contour of function f (described on Page 1). Below are the plots generated.

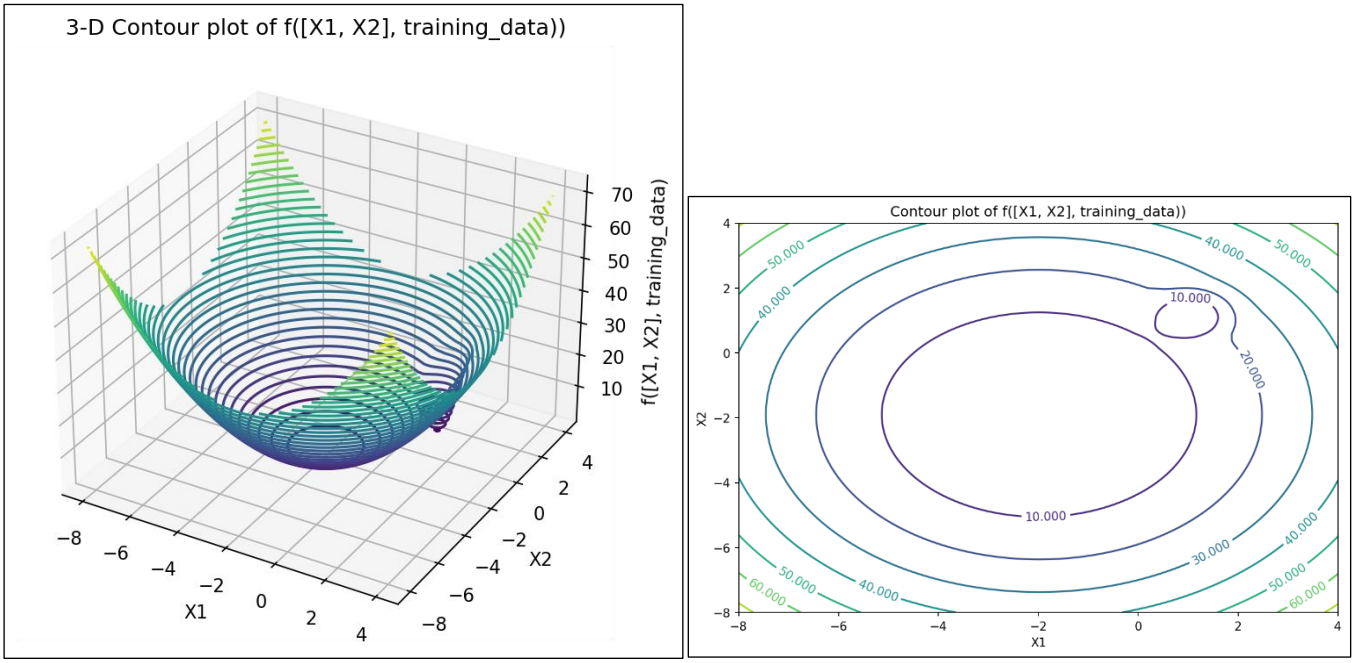

Figure 1: Contour plots of function f over complete training data in 3-D and 2-D

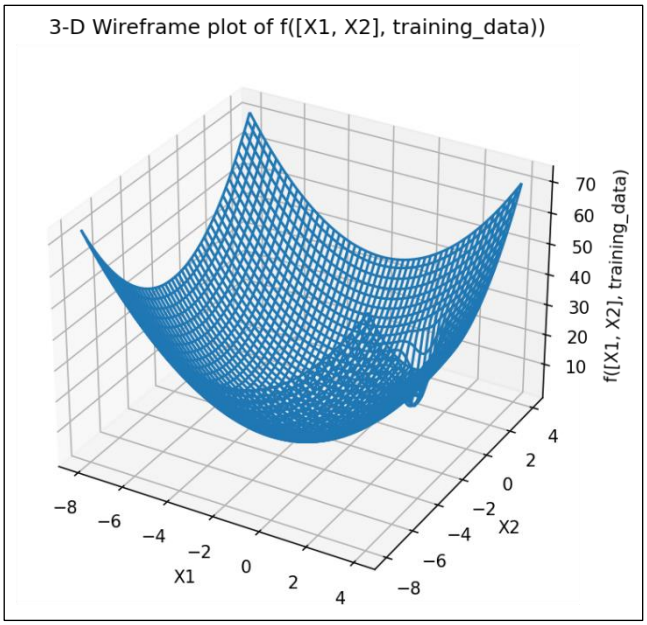

Figure 2: 3-D Wireframe plot of function f over complete training data

These plots were generated for range -8 < x1 < 4, and -8 < x2 < 4. I have used **linespace** function from **numpy** library generated 200 points in this range for both x1 and x2. The primary reason behind choosing this range is that it gives clear view of both local and global minima of the cost function. I can clearly identify a **steep curve when x1 and x2 are greater than 0 which represents a local-minima** and a **flatter and smoother convex surface when x1 and x2 < -2, which represent global minima**

Question a(iii): Get Sympy Derivative for the downloaded function

I have used **Sympy library** to implement the functions and obtain its derivatives with respect to **x1** and **x2**. This method also **includes a new parameter** in the function i.e., **'w', which represents the mini-batch training set.**

Function: Min(24 * ((x1 - w1 - 1) ** 2 + (x2 - w2 - 1) ** 2), ((x1 - w1 - 1) + 3) ** 2 + ((x2 - w2 - 1) + 3) ** 2)

```
class Question_a_iii :
    def get_equation_notation(self) :
        x1, x2, w1, w2 = sympy.symbols('x1 x2 w1 w2', real = True)
        equation = sympy.Min(24 * ((x1 - w1 - 1) ** 2 + (x2 - w2 - 1) ** 2), ((x1 - w1 - 1) + 3) ** 2 + ((x2 - w2 - 1) + 3) ** 2)
        return equation

    def get_sympy_derivative_notation_x1(self) :
        x1 = sympy.symbols('x1', real = True)
        derivative_x1 = sympy.diff(self.get_equation_notation(), x1)
        return derivative_x1

    def get_sympy_derivative_notation_x2(self) :
        x2 = sympy.symbols('x2', real = True)
        derivative_x2 = sympy.diff(self.get_equation_notation(), x2)
        return derivative_x2
```

Input:

```
q_a_iii = Question_a_iii()
f = q_a_iii.get_equation_notation()
print(f"Equation for Function f is: {f}"
```

Output:

Equation for Function f is: **Min(24*(-w1 + x1 - 1)**2 + 24*(-w2 + x2 - 1)**2, (-w1 + x1 + 2)**2 + (-w2 + x2 + 2)**2)**

Input:

```
derivative_df_dx1 = q_a_iii.get_sympy_derivative_notation_x1()
print(f"Derivative for f = {f} with respect to x1 is: {derivative_df_dx1}")
```

Output:

Derivative for f = Min(24*(-w1 + x1 - 1)**2 + 24*(-w2 + x2 - 1)**2, (-w1 + x1 + 2)**2 + (-w2 + x2 + 2)**2) with respect to x1 is:
(-48*w1 + 48*x1 - 48)*Heaviside(-24*(-w1 + x1 - 1)2 + (-w1 + x1 + 2)**2 - 24*(-w2 + x2 - 1)**2 + (-w2 + x2 + 2)**2) + (-2*w1 + 2*x1 + 4)*Heaviside(24*(-w1 + x1 - 1)**2 - (-w1 + x1 + 2)**2 + 24*(-w2 + x2 - 1)**2 - (-w2 + x2 + 2)**2)**

Input:

```
derivative_df_dx2 = q_a_iii.get_sympy_derivative_notation_x2()
print(f"Derivative for f = {f} with respect to x1 is: {derivative_df_dx2}")
```

Output:

Derivative for f = Min(24*(-w1 + x1 - 1)**2 + 24*(-w2 + x2 - 1)**2, (-w1 + x1 + 2)**2 + (-w2 + x2 + 2)**2) with respect to x1 is:
(-48*w2 + 48*x2 - 48)*Heaviside(-24*(-w1 + x1 - 1)2 + (-w1 + x1 + 2)**2 - 24*(-w2 + x2 - 1)**2 + (-w2 + x2 + 2)**2) + (-2*w2 + 2*x2 + 4)*Heaviside(24*(-w1 + x1 - 1)**2 - (-w1 + x1 + 2)**2 + 24*(-w2 + x2 - 1)**2 - (-w2 + x2 + 2)**2)**

Question b(i): SGD with a constant step-size to minimise the loss function starting from initial x = [3, 3]

I have implemented the SGD with Constant Step Size algorithm for a variety of **α in range [0.001, 0.01, 0.1**]. The initial conditions for function are **x_1 = 3 and x_2 = 3 and number of iterations = 100**. I have varied the α and tried to analyse its impact on overall convergence of the cost function. Furthermore, batch size is fixed to training data. Below are the plots generated.

From **Figure 3,** I can clearly analyse that I was able to achieve **best convergence for α=0.1.** The primary reason for this behaviour can be established from Contour Plot. I can clearly analyse **that only for α=0.1, the function was able to escape the steep local minima** and was able to **converge towards the flatter and smoother global minima**.
For, **α=0.001 and 0.01, the step size calculated using gradients was not sufficient enough to escape the steep local minima** and therefore we see its value never converges but becomes constant after certain iterations as it never escapes that steep curve.

For **α=0.1**, function **never reaches minima but get very close to minima** after 100 iterations. The primary reason for this is the large mini batch size used in SGD due to which it takes more iterations to converge to global minima. It is discussed in detail in b(iii). Thus, we can conclude that **α=0.1 is ideal choice** for this function using SGD.

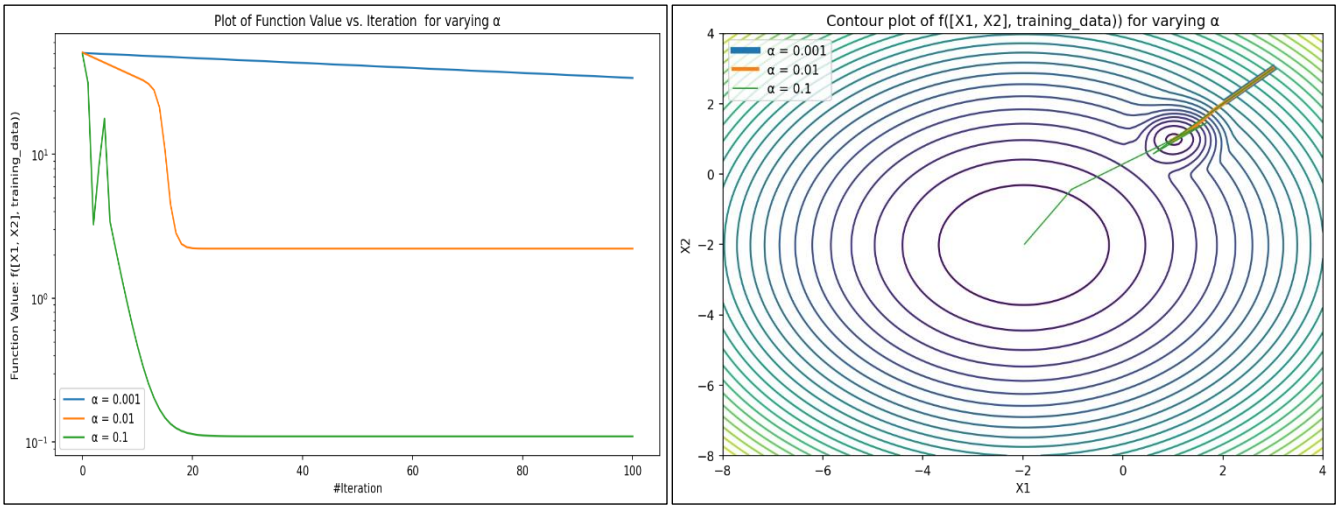

Figure 3: Plot of Cost Function Vs. Iterations and Contour Plot for Constant Step Size for range of α

Question b(ii): SGD with a mini-batch size of 5 Run the SGD several times and plot how f and x change over time.

I have implemented the SGD with Constant Step Size algorithm. The **α is set to 0.1 based on the optimisation results achieved above**. The initial conditions for function are **x_1 = 3 and x_2 = 3** and **number of iterations = 25**. I executed this for **5 trials** in order to analyse the impact of changing the training data in overall convergence of the cost function. Below are the plots generated:

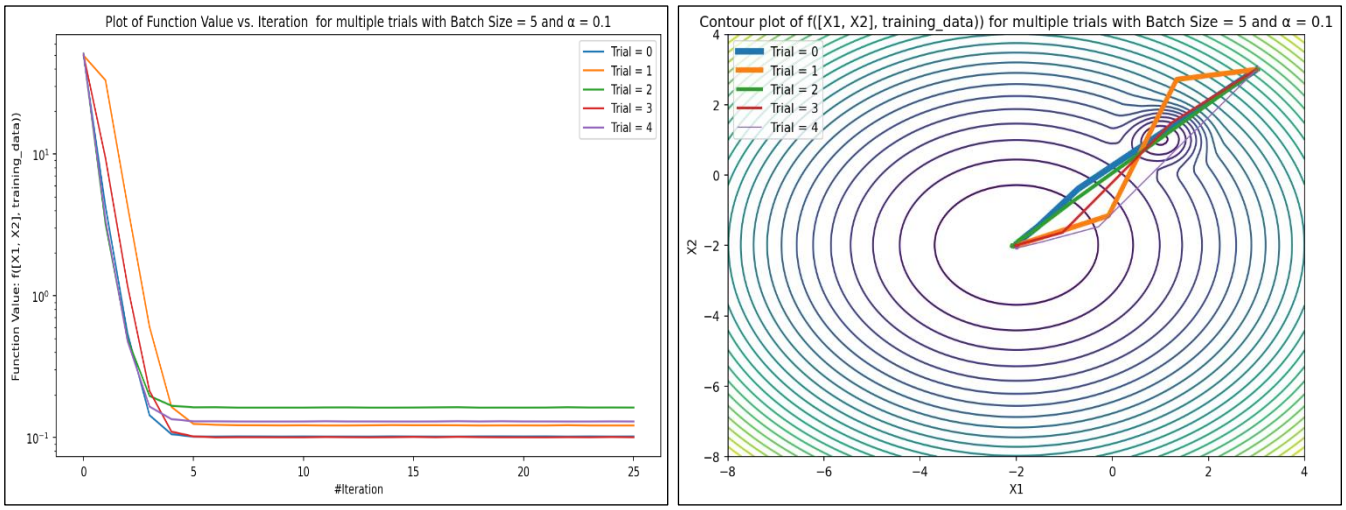

Figure 4: Plot of Cost Function Vs. Iterations and Contour Plot for five trials for fixed α

From **Figure 4**, I can clearly analyse that **for each of the trial, the cost function was converged to minimum**. From the contour plot, I can analyse that **the path of the convergence was diffrernt in each trial**. The primary reason for this behavior is that in each trial, the **training data was shuffled**. Due to the shuffling, **noise was introduced in the data**, which lead to different paths to convergence but it does not have any impact in overall convergence of the cost function. Furthermore, I observed that in all the trials, **cost function was converged to minimum**. The primary reason for this behaviour is that we used **the optimised α for all the trials based on b(i)** and **batch size is small enough to escape the local minimum**.

When **compared to SGD convergence in Figure 3**, I can analyse that **cost function was minimised in almost 5 iterations for all the trials while it took 20 iterations to converge in Figure 3** and it converged only for α=0.1 while it did not converge for α=0.001 and 0.01. The primary reason for this behaviour is the change in batch size as **smaller batch size leads to faster convergence** to minima when compared full training dataset used in b(i).

Question b(iii): SGD with step size fixed, vary the mini-batch size

I have implemented the SGD with Constant Step Size algorithm. The **α is set to 0.1** based on the optimisation results achieved above. I have executed the this for a variety **of batch size in range [1, 5, 10, 15, 20, 25]**. The initial conditions for function **are x_1 = 3 and x_2 = 3 and number of iterations = 25**. I have varied the batch size and tried to analyse its impact on overall convergence of the cost function. Below are the plots generated.

From **Figure 5**, I can clearly analyse that as **batch size increased from 1 to 25, it took more iterations to converge to global minima**. The primary reason for this behaviour is that for **larger batch size like 10 to 25**, the function **gets stuck in local minima for first few iterations** and it take a while to escape the local minima and converge toward global minima. For **smaller batch size like 1 and 5**, it **escapes the local minima and converges to global minima in just first 5 iterations**.

I can also analyse that for **smaller batch size like 1 and 5, the function completely converged to global minima** while for **larger batch size 10 to 25, it has not completely converged to global minima in the given number of iterations.**

The primary reason for **larger batch size to get stuck in local minima** is that **gradient is calculated for large data points** and it leads to **overfitting of the training data**. Due to this **model is not updated frequently** as compared to smaller batch sizes and **it gets stuck in the local minimum for that specific batch** and it takes more iterations to escape the local minima. On the other hand, **smaller batch size** leads to more **noise in the calculated gradient** because of which it is able to **escape the local minima quickly** without getting trapped. So, **noise is actually helping the function to converge to escape the local minima**.

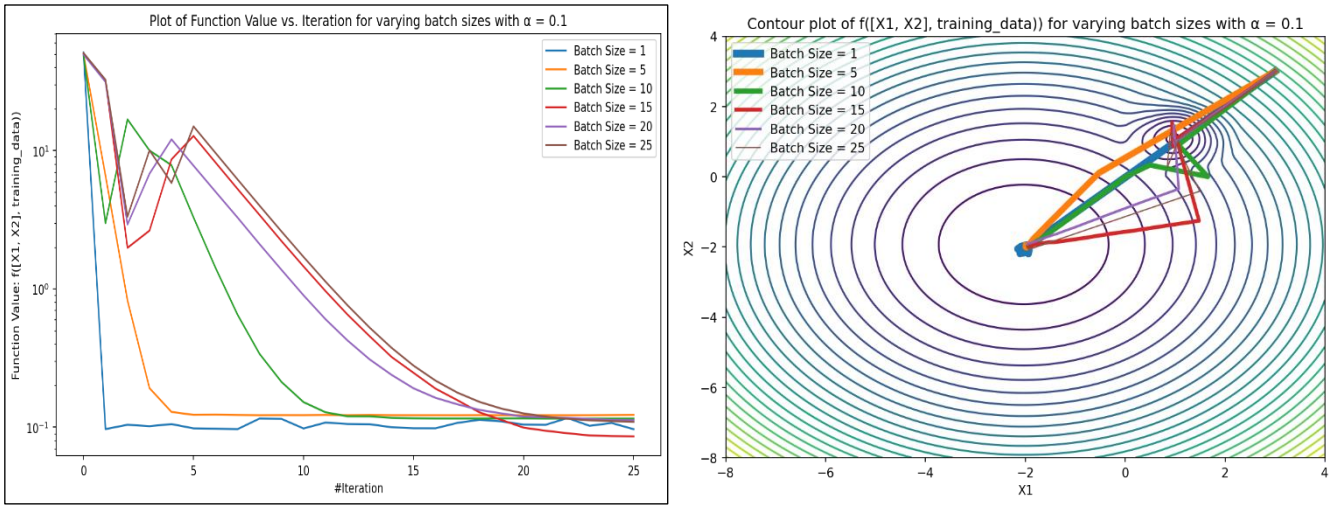

Figure 5: Plot of Cost Function Vs. Iterations and Contour Plot for range of batch size and fixed α

Question b(iv): SGD with the mini-batch size fixed at 5, vary the step size

I have implemented the SGD for a variety of **α in range [0.001, 0.01, 0.1]**. The initial conditions for function are **x_1 = 3 and x_2 = 3 and number of iterations = 25**. I have varied the α and tried to analyse its impact on overall convergence of the cost function. Furthermore, **batch size is fixed to 5**. Below are the plots generated.

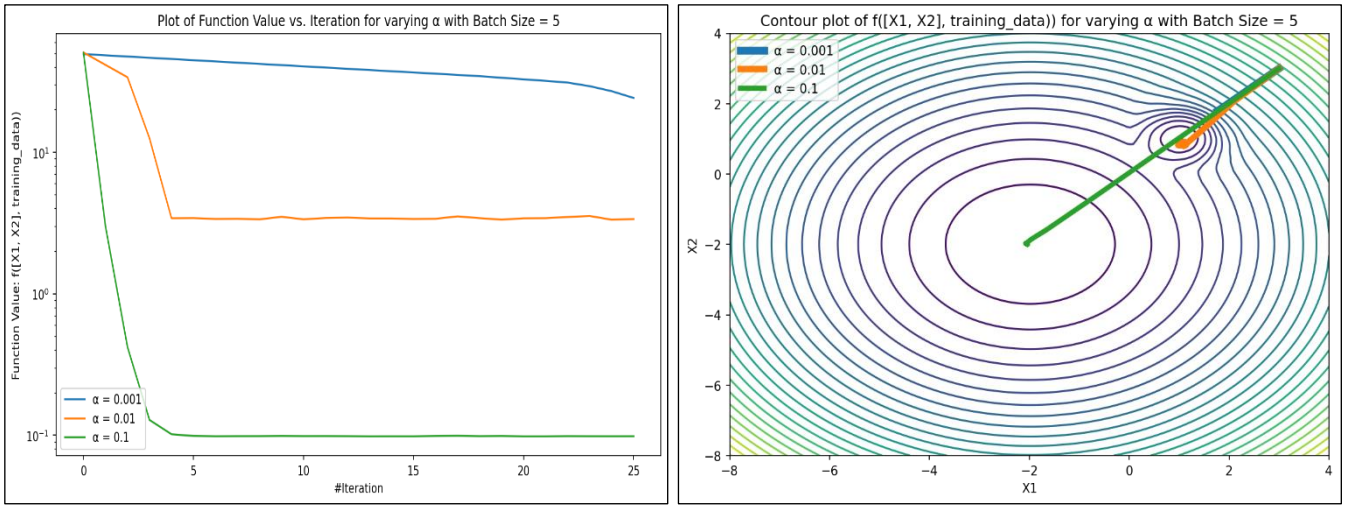

Figure 6: Plot of Cost Function Vs. Iterations and Contour Plot for range α and fixed batch size of 5

From **Figure 6**, I can analyse that the value **converged to minimum only for α=0.1** while it failed to converge to minima for smaller α 0.001 and 0.01. The primary reason for this behaviour is that for **smaller values of α, the step calculated is not sufficient enough to escape the steep local minima** of the cost curve. Even though batch size is small to introduce enough noise in the mini batches, still it gets **stuck because of the small steps and steep curve**. Therefore, we see its value never converges but becomes constant after certain iterations as it never escapes that steep curve.

Thus, I can conclude that both **α and batch size play a critical role in overall convergence** of the cost function. The **larger value of α** ensures that **step size is sufficient enough to escape local minima** but not choose very large value of α that leads to divergence. Furthermore, **smaller batch sizes** ensure that enough **noise in introduced** in the training data so that the **model does not overfit** to mini batch and takes **less iterations to escape local minima**.

Question c(i): Polyak with SGD

I have implemented the SGD with Polyak step for a variety of batch size in range [1, 5, 10, 15, 20, 25]. The initial conditions for function are **x_1 = 3 and x_2 = 3 and number of iterations = 100**. Since, batch size is the only parameter that can be optimised for Polyak step, I have varied it and tried to analyse its impact on overall convergence of the cost function. Also, I have used SGD with Constant Step as baseline model. The tuned hyperparameters used for baseline model are α=0.1 and batch size=5. Below are the plots generated:

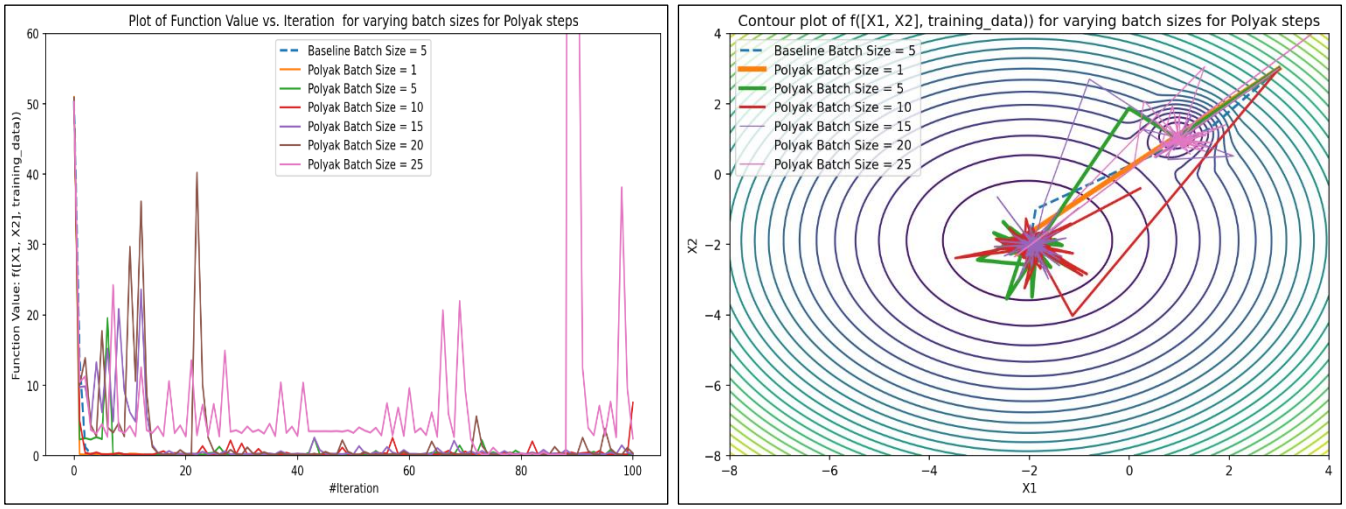

Figure 7: Plot of Cost Function Vs. Iterations and Contour Plot for range of batch size for Polyak step

From Figure 7, I can analyse that function converges to global minima but it keeps on oscillating over the minimum even after 100 iterations and never fully converge to global minima. Furthermore, for large batch size like 15 to 25, the function gets stuck in steep local minima and take a few iterations to escape it. Even in the local minim it keeps on oscillating and never becomes stable.

For smaller batch like 1 to 10, the function is able to escape local minima quickly but it is also not stable even though it has converged in the flatter and smoother global minima and it keeps on oscillating over it and never converging fully into it. The primary reason for this behaviour is because of the momentum involved in the Polyak step. High momentum due to squared gradients can lead overshooting from the global minima because of large value of step size. For larger batch size, the momentum will be higher and therefore it gets stuck in the local minima.

When compared to Constant Step baseline model, I can clearly analyse that baseline model performs better that Polyak step model for all variations of batch size. Convergence of the Baseline model is very smooth and involves no oscillations. It converges to global minima in very few iterations and from contour plot I can analyse that it never gets trapped in local minima. Although Polyak model with small batch size of 1 to 10 also converges quickly, but still, it is unstable as it keeps on oscillating over the minima without fully converging into it.

Question c(ii): RMSProp with SGD

I have implemented the SGD with RMSProp step for a variety of α in range [0.001, 0.01, 0.1] and β in range [0.25, 0.9] in order to tune these hyper parameters. The initial conditions for function are **x_1 = 3 and x_2 = 3 and number of iterations = 100.** Below are the plots generated:

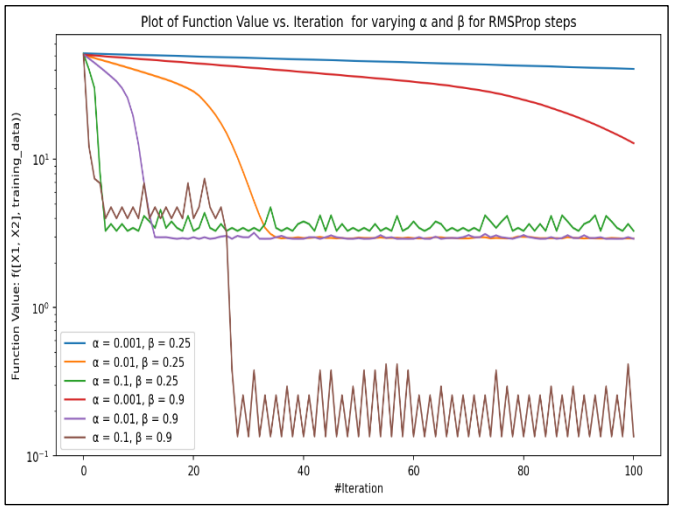

Figure 8: Plot of Cost Function Vs. Iterations for range of α and β for RMSProp step

From Figure 8, I can clearly analyse that tuned hyper parameter for RMSProp are α=0.1 and β=0.9 as function converges to minimum for these hyper parameters after just 30 iteration. For other values of α and β, the value never converges to minima.

I have used these hyper parameters to implement the SGD with RMSProp step for a variety of batch size in range [1, 5, 10, 15, 20, 25]. The initial conditions for function are **x_1 = 3, x_2 = 3, number of iterations = 100, α=0.1 and β=0.9**. I have varied batch size and tried to analyse its impact on overall convergence of the cost function. Also, I have used SGD with Constant Step as baseline model. The tuned hyperparameters used for baseline model are α=0.1 and batch size=5. Below are the plots generated:

From Figure 9, I can clearly analyse that RMSProp model gets trapped in local minima for large values of batch in range 10 to 25, while it is able to escape the local minima for batch size 1 and 5. However, it is able to converge towards minima only for batch size 1.

When compared to the baseline model with Constant step size, the baseline model performs better that RMSProp model for all variations of the batch size. Convergence of the Baseline model is very smooth and involves no oscillations. It converges to global minima in very few

iterations and from contour plot I can analyse that it never gets trapped in local minima. Only for batch size 1, the RMSProp model is able to converge quickly but again it is unstable and keeps on oscillating over the global minima rather than fully converging over it. The primary reason for oscillation is that RMSProp uses the average of the squares of the past gradients to update the model parameters. This value changes in each iteration due to large variance in the gradients of each parameter leading to oscillations.

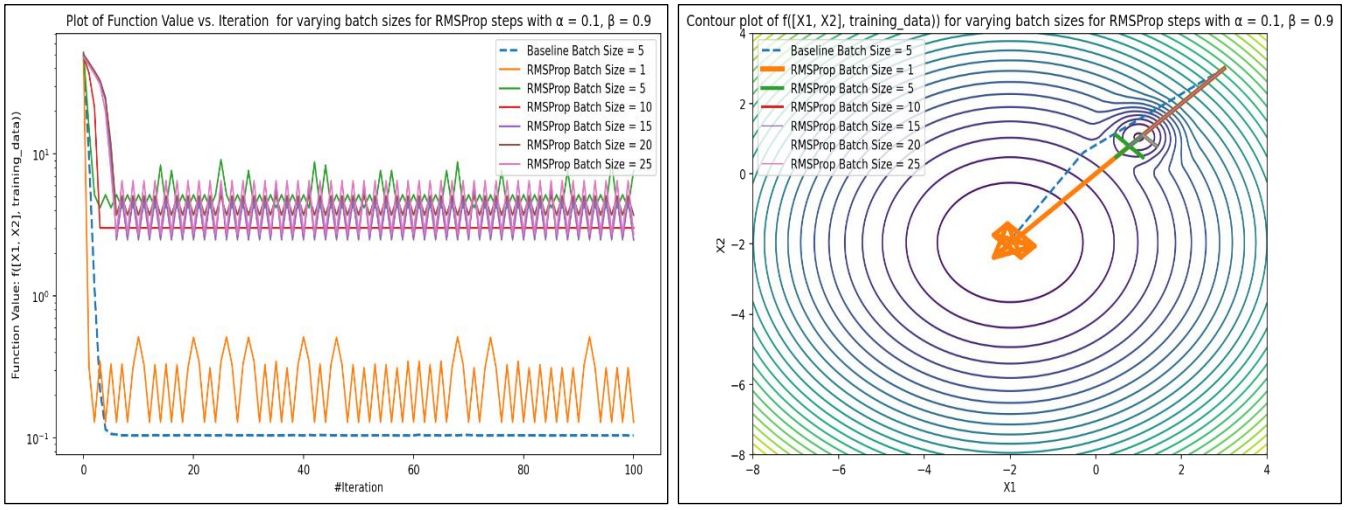

Figure 9: Plot of Cost Function Vs. Iterations and Contour Plot for range of batch size for RMSProp step

Question c(iii): Heavy Ball with SGD

I have implemented the SGD with Heavy Ball step for a variety of α in range [0.001, 0.01, 0.1] and β in range [0.25, 0.9] in order to tune these hyper parameters. The initial conditions for function are **x_1 = 3 and x_2 = 3 and number of iterations = 100.** Below are the plots generated:

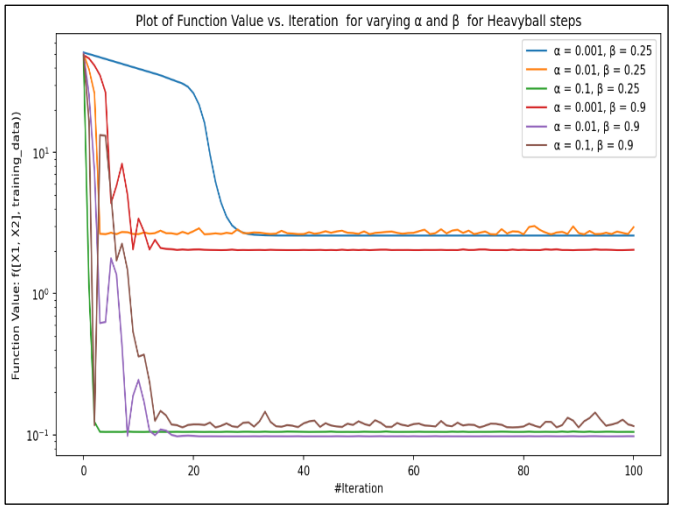

Figure 10: Plot of Cost Function Vs. Iterations for range of α and β for Heavy Ball step

From Figure 10, I can clearly analyse that tuned hyper parameter for Heavy Ball are α=0.1 and β=0.25 as function converges to minimum for these hyper parameters after just 5 iteration.

I have used these hyper parameters to implement the SGD with Heavy Ball step for a variety of batch size in range [1, 5, 10, 15, 20, 25]. The initial conditions for function are **x_1 = 3, x_2 = 3, number of iterations = 100, α=0.1 and β=0.25**. I have varied batch size and tried to analyse its impact on overall convergence of the cost function. Also, I have used SGD with Constant Step as baseline model. The tuned hyperparameters used for baseline model are α=0.1 and batch size=5. Below are the plots generated.

From Figure 11, I can analyse that Heavy ball SGD was able to escape the local minima for all variations of the batch sizes. However, it only converges to minima only for small batch size in range 5 to 15. For very small batch size like 1, it oscillates over global minima while for very large batch size like 20 to 25, it never fully converges to minima. I was able to achieve best results for batch size 5.

When compared to the baseline model with Constant step size, the Heavy ball model with batch size 5 performs better than baseline model. It converges faster to global minima than baseline model. The primary reason for this behaviour is that momentum in heavy ball model allows it to move in the direction of the descent even if the gradient changes its direction for small mini batch. Due to this, random fluctuations in the gradient of small batch size gets smoothened out.

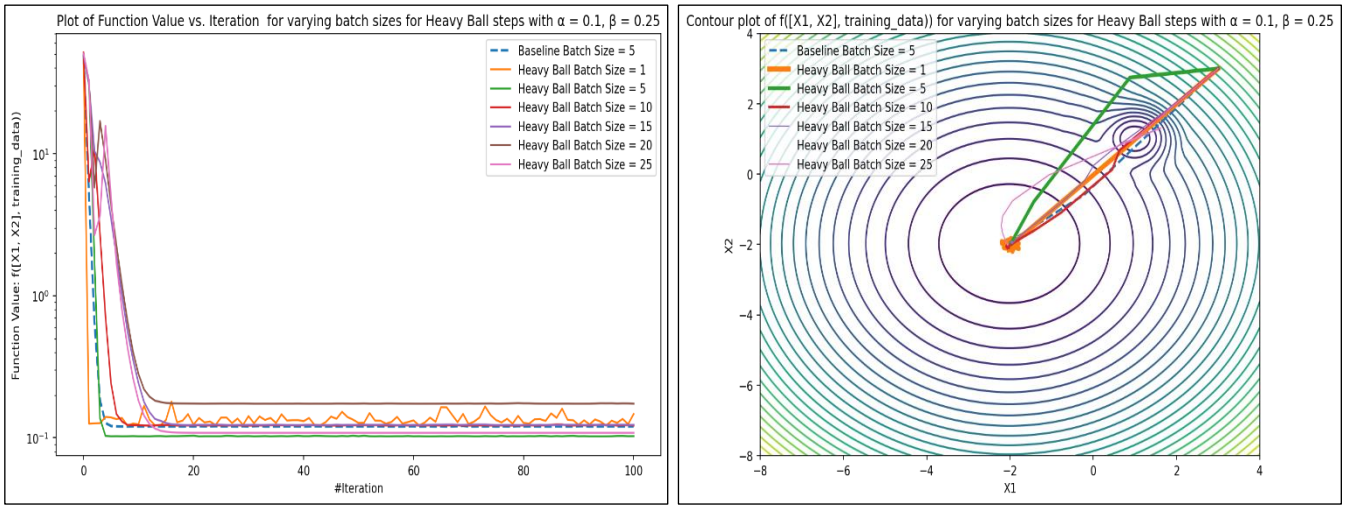

Figure 11: Plot of Cost Function Vs. Iterations and Contour Plot for range of batch size for Heavy Ball step

Question c(iv): Adam with SGD

I have implemented the SGD with Adam step for a variety of α in range [0.001, 0.01, 0.1], β1 in range [0.25, 0.9], and β2 in range [0.9, 0.999] in order to tune these hyper parameters. The initial conditions for function are **x_1 = 3 and x_2 = 3 and number of iterations = 100.** Below are the plots generated:

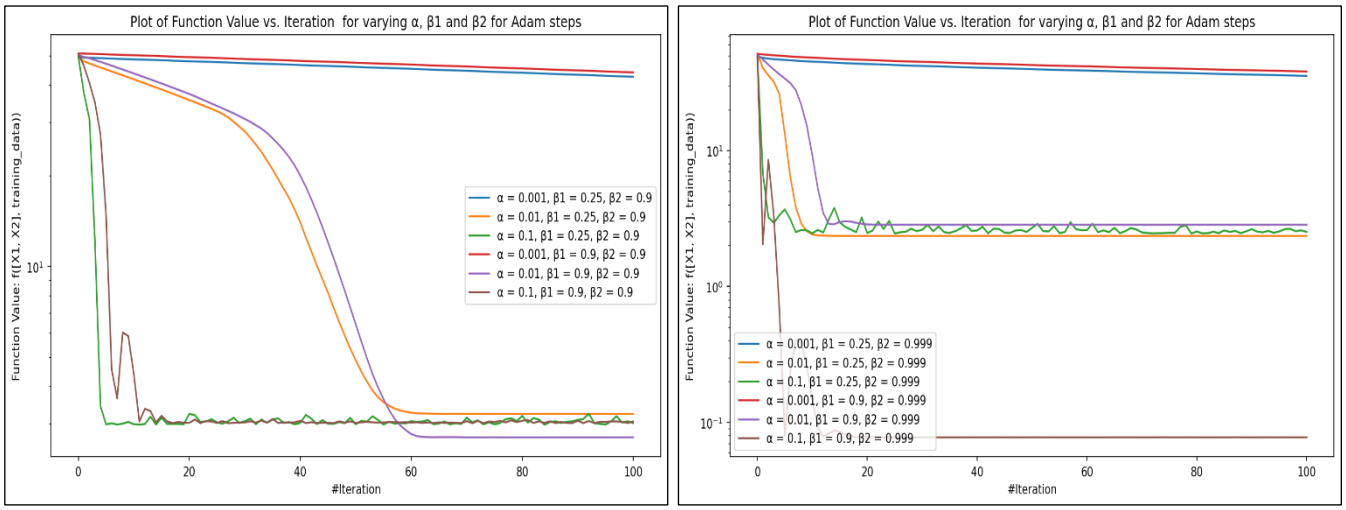

Figure 12: Plot of Cost Function Vs. Iterations for range of α, β1 and β2 for Adam step

From Figure 12, I can clearly analyse that tuned hyper parameter for Adam are α=0.1, β1=0.9 and β2=0.999 as function converges to minimum for these hyper parameters after just 5 iteration.

I have used these hyper parameters to implement the SGD with Heavy Ball step for a variety of batch size in range [1, 5, 10, 15, 20, 25]. The initial conditions for function are **x_1 = 3, x_2 = 3, number of iterations = 100, α=0.1, β1=0.9 and β2=0.999**. I have varied batch size and tried to analyse its impact on overall convergence of the cost function. Also, I have used SGD with Constant Step as baseline model. The tuned hyperparameters used for baseline model are α=0.1 and batch size=5. Below are the plots generated

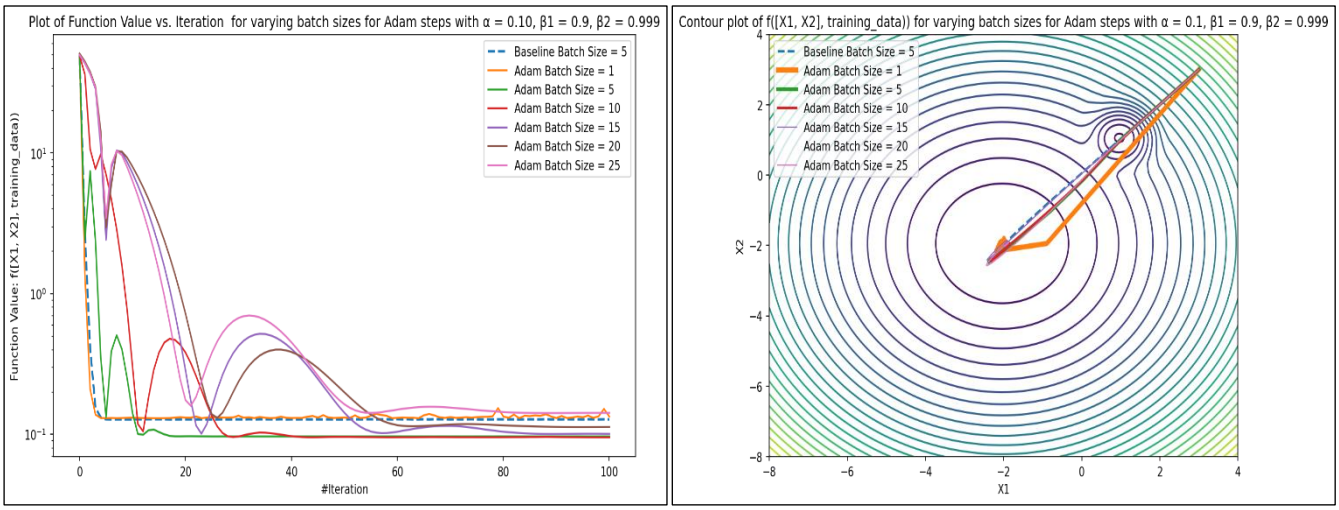

Figure 13: Plot of Cost Function Vs. Iterations and Contour Plot for range of batch size for Adam step

From Figure 11, I can analyse that Adam SGD was able to escape the local minima for all variations of the batch sizes. Furthermore, for batch size in range 5 to 10, it coverages to minima in only 10 iteration. For very small batch size like 1, it oscillates over global minima while for very large batch size like 20 to 25, it takes more than 90 iterations to converge.

When compared to the baseline model with Constant step size, the Adam model with batch size 5 and 10 perform better than Constant Step size model but they take a greater number of iterations to converge to global minima. The primary reason for this behaviour is due to presence of current and past gradients in Adam. **β1** decides the magnitude current gradients and gives velocity to the graph to move towards minima. **β2** provide momentum to graph to move towards minima when function approaches in flatter regions of the curve by giving more wight to sum pf previous gradients. Due to this it is able to quickly escape the local minima ana converge towards global minima.

Appendix: Code for Question a – All Parts

```
import numpy as np
import sympy
from sympy import Heaviside
import random
import math
import matplotlib.pyplot as plot

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(24*(z[0]**2+z[1]**2), (z[0]+3)**2+(z[1]+3)**2)
    count=count+1
  return y/count
```

Question a(i): Mini-batch Stochastic Gradient Descent for Constant Step Size, Polyak, RMSProp, Heavy Ball and Adam steps

```
class Stochastic_Gradient_Descent :

  def __init__(self, starting_point  = [1, 1]) :
    self.train_data = self.generate_trainingdata()
    self.x1 = starting_point[0]
    self.x2 = starting_point[1]
    self.x1_values = np.array([self.x1])
    self.x2_values = np.array([self.x2])
    self.function_values = np.array([self.f([self.x1, self.x2], self.train_data)])
    self.x_z = self.y_z = self.x_v = self.x_m = self.y_v = self.y_m = self.adam_iteration = 0

  def f(self, 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(24*(z[0]**2+z[1]**2), (z[0]+3)**2+(z[1]+3)**2)
      count=count+1
    return y/count

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

  def get_derivative_value(self, x1, w1, x2, w2) :
    derivative_x1_value = (-48*w1 + 48*x1 - 48)*Heaviside(-24*(-w1 + x1 - 1)**2 + (-w1 + x1 + 2)**2 - 24*(-w2 + x2 - 1)**2 + (-w2 + x2 +
2)**2) + (-2*w1 + 2*x1 + 4)*Heaviside(24*(-w1 + x1 - 1)**2 - (-w1 + x1 + 2)**2 + 24*(-w2 + x2 - 1)**2 - (-w2 + x2 + 2)**2)
    derivative_x2_value = (-48*w2 + 48*x2 - 48)*Heaviside(-24*(-w1 + x1 - 1)**2 + (-w1 + x1 + 2)**2 - 24*(-w2 + x2 - 1)**2 + (-w2 + x2 +
2)**2) + (-2*w2 + 2*x2 + 4)*Heaviside(24*(-w1 + x1 - 1)**2 - (-w1 + x1 + 2)**2 + 24*(-w2 + x2 - 1)**2 - (-w2 + x2 + 2)**2)
    return derivative_x1_value, derivative_x2_value

  def get_batch_derivative(self, x1, x2, sample_batch) :
    sum_derivative_x1_value = 0
    sum_derivative_x2_value = 0

    for w1, w2 in sample_batch :
      derivative_x1_value, derivative_x2_value = self.get_derivative_value(x1, w1, x2, w2)
      sum_derivative_x1_value += derivative_x1_value
      sum_derivative_x2_value += derivative_x2_value

    batch_derivative_x1_value = sum_derivative_x1_value / len(sample_batch)
    batch_derivative_x2_value = sum_derivative_x2_value / len(sample_batch)

    return batch_derivative_x1_value, batch_derivative_x2_value
```

```
  def execute_stochastic_gradient_descent(self, gradient_descent_type = 'Constant', alpha = 0.1, beta = 0.9, beta2 = 0.999, batch_size =
10) :

    np.random.shuffle(self.train_data)
    for j in range(0, len(self.train_data), batch_size) :

      if (j + batch_size) > len(self.train_data) :
        continue

      if gradient_descent_type == 'Constant Step Size' :
        self.x1, self.x2 = self.execute_constant_size_gradient_descent(self.train_data[j : (j + batch_size)], alpha)

      elif gradient_descent_type == 'Polyak' :
        self.x1, self.x2 = self.execute_polyak_gradient_descent(self.train_data[j : (j + batch_size)])

      elif gradient_descent_type == 'RMSProp':
        self.x1, self.x2 = self.execute_rmsprop_gradient_descent(self.train_data[j : (j + batch_size)], alpha, beta)

      elif gradient_descent_type == 'HeavyBall':
        self.x1, self.x2 = self.execute_heavyball_gradient_descent(self.train_data[j : (j + batch_size)], alpha, beta)

      elif gradient_descent_type == 'Adam':
        self.x1, self.x2 = self.execute_adam_gradient_descent(self.train_data[j : (j + batch_size)], alpha, beta, beta2)


    self.x1_values = np.append(self.x1_values, [self.x1], axis = 0)
    self.x2_values = np.append(self.x2_values, [self.x2], axis = 0)
    self.function_values = np.append(self.function_values, [self.f([self.x1, self.x2], self.train_data)])

    return self.x1_values, self.x2_values, self.function_values

  def execute_constant_size_gradient_descent(self, sample_batch, alpha) :
    batch_derivative_x1, batch_derivative_x2 = self.get_batch_derivative(self.x1, self.x2, sample_batch)
    self.x1 -= alpha * batch_derivative_x1
    self.x2 -= alpha * batch_derivative_x2
    return self.x1, self.x2

  def execute_polyak_gradient_descent(self, sample_batch) :
    epsilon = 1e-8
    numerator = self.f([self.x1, self.x2], sample_batch)
    batch_derivative_x1, batch_derivative_x2 = self.get_batch_derivative(self.x1, self.x2, sample_batch)
    denominator = ((batch_derivative_x1) **2 + (batch_derivative_x2) ** 2) + epsilon
    step =  (numerator / denominator) if denominator !=0  else 0
    self.x1 = self.x1 - step * batch_derivative_x1
    self.x2 = self.x2 - step * batch_derivative_x2

    return self.x1, self.x2

  def execute_rmsprop_gradient_descent(self, sample_batch, alpha, beta) :
    epsilon = 1e-8
    x_alpha = alpha
    y_alpha = alpha
    x_sum = 0
    y_sum = 0

    batch_derivative_x1, batch_derivative_x2 = self.get_batch_derivative(self.x1, self.x2, sample_batch)
    x_sum = (x_sum * beta) + ((1 - beta) * ((batch_derivative_x1) ** 2))
    y_sum = (y_sum * beta) + ((1 - beta) * ((batch_derivative_x2) ** 2))

    x_alpha = alpha / (math.sqrt(x_sum) + epsilon)
    y_alpha = alpha / (math.sqrt(y_sum) + epsilon)

    self.x1 = self.x1 - (x_alpha * batch_derivative_x1)
    self.x2 = self.x2 - (y_alpha * batch_derivative_x2)


    return self.x1, self.x2
```

```
    def execute_heavyball_gradient_descent(self, sample_batch, alpha, beta) :
        epsilon = 1e-8
        batch_derivative_x1, batch_derivative_x2 = self.get_batch_derivative(self.x1, self.x2, sample_batch)
        self.x_z = (self.x_z * beta) + (alpha * batch_derivative_x1)
        self.y_z = (self.y_z * beta) + (alpha * batch_derivative_x2)

        self.x1 = self.x1 - self.x_z
        self.x2 = self.x2 - self.y_z

        return self.x1, self.x2

    def execute_adam_gradient_descent(self, sample_batch, alpha, beta_1, beta_2) :
        epsilon = 1e-8

        batch_derivative_x1, batch_derivative_x2 = self.get_batch_derivative(self.x1, self.x2, sample_batch)

        self.x_m = (self.x_m * beta_1) + ((1 - beta_1) * batch_derivative_x1)
        self.y_m = (self.y_m * beta_1) + ((1 - beta_1) * batch_derivative_x2)

        self.x_v = (self.x_v * beta_2) + ((1 - beta_2) * ((batch_derivative_x1) ** 2))
        self.y_v = (self.y_v * beta_2) + ((1 - beta_2) * ((batch_derivative_x2) ** 2))

        x_m_hat = self.x_m / (1 - beta_1 ** self.adam_iteration + 1)
        y_m_hat = self.y_m / (1 - beta_1 ** self.adam_iteration + 1)

        x_v_hat = self.x_v / (1 - beta_2 ** self.adam_iteration + 1)
        y_v_hat = self.y_v / (1 - beta_2 ** self.adam_iteration + 1)

        x_alpha = x_m_hat / (math.sqrt(x_v_hat) + epsilon)
        y_alpha = y_m_hat / (math.sqrt(y_v_hat) + epsilon)

        self.x1 = self.x1 - (x_alpha * alpha)
        self.x2 = self.x2 - (y_alpha * alpha)

        self.adam_iteration += 1

        return self.x1, self.x2
```

Question a(ii): Plot a wireframe and a contour plot of f for N = T

```
class Question_a_ii :

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

    def f(self, 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(24*(z[0]**2+z[1]**2), (z[0]+3)**2+(z[1]+3)**2)
            count=count+1
        return y/count

    def generate_wirefame_contour_plot(self) :
        x1 = np.linspace(-8, 4, 200)
        x2 = np.linspace(-8, 4, 200)
        f = []
        training_data = self.generate_trainingdata()

        for x1_point in x1 :
            data_point = []
            for x2_point in x2 :
                data_point.append(self.f([x1_point, x2_point], training_data))
            f.append(data_point)
        f_x1_x2 = np.array(f)
        X1, X2 = np.meshgrid(x1, x2)
```

```
        plot.figure(figsize=(9, 6), dpi=150)
        ax = plot.axes(projection='3d')
        ax.contour3D(X1, X2, f_x1_x2, 45)
        ax.set_xlabel('X1')
        ax.set_ylabel('X2')
        ax.set_zlabel('f([X1, X2], training_data)')
        ax.set_title('3-D Contour plot of f([X1, X2], training_data))')
        plot.show()

        plot.figure(figsize=(9, 6), dpi=150)
        ax = plot.axes()
        countour = ax.contour(X1, X2, f_x1_x2)
        ax.set_xlabel('X1')
        ax.set_ylabel('X2')
        ax.clabel(countour)
        ax.set_title('Contour plot of f([X1, X2], training_data))')
        plot.show()


        ax = plot.figure(figsize=(9, 6), dpi=150).add_subplot(111, projection='3d')
        ax.plot_wireframe(X1, X2, f_x1_x2)
        ax.set_xlabel('X1')
        ax.set_ylabel('X2')
        ax.set_zlabel('f([X1, X2], training_data)')
        ax.set_title('3-D Wireframe plot of f([X1, X2], training_data))')
        plot.show()


q_a_ii = Question_a_ii()
q_a_ii.generate_wirefame_contour_plot()
```

Question a(iii): Get Sympy Derivative for the downloaded function

```
class Question_a_iii :

    def get_equation_notation(self) :
        x1, x2, w1, w2 = sympy.symbols('x1 x2 w1 w2', real = True)
        equation = sympy.Min(24 * ((x1 - w1 - 1) ** 2 + (x2 - w2 - 1) ** 2), ((x1 - w1 - 1) + 3) ** 2 + ((x2 - w2 - 1) + 3) ** 2)
        return equation

    def get_sympy_derivative_notation_x1(self) :
        x1 = sympy.symbols('x1', real = True)
        derivative_x1 = sympy.diff(self.get_equation_notation(), x1)
        return derivative_x1

    def get_sympy_derivative_notation_x2(self) :
        x2 = sympy.symbols('x2', real = True)
        derivative_x2 = sympy.diff(self.get_equation_notation(), x2)
        return derivative_x2


q_a_iii = Question_a_iii()

f = q_a_iii.get_equation_notation()
print(f"Equation for Function f is: {f}")

derivative_df_dx1 = q_a_iii.get_sympy_derivative_notation_x1()
print(f"Derivative for f = {f} with respect to x1 is: {derivative_df_dx1}")

derivative_df_dx2 = q_a_iii.get_sympy_derivative_notation_x2()
print(f"Derivative for f = {f} with respect to x1 is: {derivative_df_dx2}")
```

Appendix: Code for Question b – All Parts

```
import numpy as np
import sympy
from sympy import Heaviside
import random
import math
import matplotlib.pyplot as plot

class Stochastic_Gradient_Descent :

    def __init__(self, starting_point  = [1, 1]) :
        self.train_data = self.generate_trainingdata()
        self.x1 = starting_point[0]
        self.x2 = starting_point[1]
        self.x1_values = np.array([self.x1])
        self.x2_values = np.array([self.x2])
        self.function_values = np.array([self.f([self.x1, self.x2], self.train_data)])
        self.x_z = self.y_z = self.x_v = self.x_m = self.y_v = self.y_m = self.adam_iteration = 0

    def f(self, 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(24*(z[0]**2+z[1]**2), (z[0]+3)**2+(z[1]+3)**2)
            count=count+1
        return y/count

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

    def get_derivative_value(self, x1, w1, x2, w2) :
        derivative_x1_value = (-48*w1 + 48*x1 - 48)*Heaviside(-24*(-w1 + x1 - 1)**2 + (-w1 + x1 + 2)**2 - 24*(-w2 + x2 - 1)**2 + (-w2 + x2 +
2)**2) + (-2*w1 + 2*x1 + 4)*Heaviside(24*(-w1 + x1 - 1)**2 - (-w1 + x1 + 2)**2 + 24*(-w2 + x2 - 1)**2 - (-w2 + x2 + 2)**2)
        derivative_x2_value = (-48*w2 + 48*x2 - 48)*Heaviside(-24*(-w1 + x1 - 1)**2 + (-w1 + x1 + 2)**2 - 24*(-w2 + x2 - 1)**2 + (-w2 + x2 +
2)**2) + (-2*w2 + 2*x2 + 4)*Heaviside(24*(-w1 + x1 - 1)**2 - (-w1 + x1 + 2)**2 + 24*(-w2 + x2 - 1)**2 - (-w2 + x2 + 2)**2)
        return derivative_x1_value, derivative_x2_value

    def get_batch_derivative(self, x1, x2, sample_batch) :
        sum_derivative_x1_value = 0
        sum_derivative_x2_value = 0

        for w1, w2 in sample_batch :
            derivative_x1_value, derivative_x2_value = self.get_derivative_value(x1, w1, x2, w2)
            sum_derivative_x1_value += derivative_x1_value
            sum_derivative_x2_value += derivative_x2_value

        batch_derivative_x1_value = sum_derivative_x1_value / len(sample_batch)
        batch_derivative_x2_value = sum_derivative_x2_value / len(sample_batch)

        return batch_derivative_x1_value, batch_derivative_x2_value


    def execute_stochastic_gradient_descent(self, gradient_descent_type = 'Constant', alpha = 0.1, beta = 0.9, beta2 = 0.999, batch_size =
10) :

        np.random.shuffle(self.train_data)
        for j in range(0, len(self.train_data), batch_size) :

            if (j + batch_size) > len(self.train_data) :
                continue

            if gradient_descent_type == 'Constant Step Size' :
                self.x1, self.x2 = self.execute_constant_size_gradient_descent(self.train_data[j : (j + batch_size)], alpha)

            elif gradient_descent_type == 'Polyak' :
                self.x1, self.x2 = self.execute_polyak_gradient_descent(self.train_data[j : (j + batch_size)])
```

```
            elif gradient_descent_type == 'RMSProp':
                self.x1, self.x2 = self.execute_rmsprop_gradient_descent(self.train_data[j : (j + batch_size)], alpha, beta)

            elif gradient_descent_type == 'HeavyBall':
                self.x1, self.x2 = self.execute_heavyball_gradient_descent(self.train_data[j : (j + batch_size)], alpha, beta)

            elif gradient_descent_type == 'Adam':
                self.x1, self.x2 = self.execute_adam_gradient_descent(self.train_data[j : (j + batch_size)], alpha, beta, beta2)


        self.x1_values = np.append(self.x1_values, [self.x1], axis = 0)
        self.x2_values = np.append(self.x2_values, [self.x2], axis = 0)
        self.function_values = np.append(self.function_values, [self.f([self.x1, self.x2], self.train_data)])

        return self.x1_values, self.x2_values, self.function_values

    def execute_constant_size_gradient_descent(self, sample_batch, alpha) :
        batch_derivative_x1, batch_derivative_x2 = self.get_batch_derivative(self.x1, self.x2, sample_batch)
        self.x1 -= alpha * batch_derivative_x1
        self.x2 -= alpha * batch_derivative_x2
        return self.x1, self.x2

    def execute_polyak_gradient_descent(self, sample_batch) :
        epsilon = 1e-8
        numerator = self.f([self.x1, self.x2], sample_batch)
        batch_derivative_x1, batch_derivative_x2 = self.get_batch_derivative(self.x1, self.x2, sample_batch)
        denominator = ((batch_derivative_x1) **2 + (batch_derivative_x2) ** 2) + epsilon
        step =  (numerator / denominator) if denominator !=0  else 0
        self.x1 = self.x1 - step * batch_derivative_x1
        self.x2 = self.x2 - step * batch_derivative_x2

        return self.x1, self.x2

    def execute_rmsprop_gradient_descent(self, sample_batch, alpha, beta) :
        epsilon = 1e-8
        x_alpha = alpha
        y_alpha = alpha
        x_sum = 0
        y_sum = 0

        batch_derivative_x1, batch_derivative_x2 = self.get_batch_derivative(self.x1, self.x2, sample_batch)
        x_sum = (x_sum * beta) + ((1 - beta) * ((batch_derivative_x1) ** 2))
        y_sum = (y_sum * beta) + ((1 - beta) * ((batch_derivative_x2) ** 2))

        x_alpha = alpha / (math.sqrt(x_sum) + epsilon)
        y_alpha = alpha / (math.sqrt(y_sum) + epsilon)

        self.x1 = self.x1 - (x_alpha * batch_derivative_x1)
        self.x2 = self.x2 - (y_alpha * batch_derivative_x2)


        return self.x1, self.x2

    def execute_heavyball_gradient_descent(self, sample_batch, alpha, beta) :
        epsilon = 1e-8
        batch_derivative_x1, batch_derivative_x2 = self.get_batch_derivative(self.x1, self.x2, sample_batch)
        self.x_z = (self.x_z * beta) + (alpha * batch_derivative_x1)
        self.y_z = (self.y_z * beta) + (alpha * batch_derivative_x2)

        self.x1 = self.x1 - self.x_z
        self.x2 = self.x2 - self.y_z

        return self.x1, self.x2

    def execute_adam_gradient_descent(self, sample_batch, alpha, beta_1, beta_2) :
        epsilon = 1e-8

        batch_derivative_x1, batch_derivative_x2 = self.get_batch_derivative(self.x1, self.x2, sample_batch)
```

```
        self.x_m = (self.x_m * beta_1) + ((1 - beta_1) * batch_derivative_x1)
        self.y_m = (self.y_m * beta_1) + ((1 - beta_1) * batch_derivative_x2)

        self.x_v = (self.x_v * beta_2) + ((1 - beta_2) * ((batch_derivative_x1) ** 2))
        self.y_v = (self.y_v * beta_2) + ((1 - beta_2) * ((batch_derivative_x2) ** 2))

        x_m_hat = self.x_m / (1 - beta_1 ** self.adam_iteration + 1)
        y_m_hat = self.y_m / (1 - beta_1 ** self.adam_iteration + 1)

        x_v_hat = self.x_v / (1 - beta_2 ** self.adam_iteration + 1)
        y_v_hat = self.y_v / (1 - beta_2 ** self.adam_iteration + 1)

        x_alpha = x_m_hat / (math.sqrt(x_v_hat) + epsilon)
        y_alpha = y_m_hat / (math.sqrt(y_v_hat) + epsilon)

        self.x1 = self.x1 - (x_alpha * alpha)
        self.x2 = self.x2 - (y_alpha * alpha)

        self.adam_iteration += 1

        return self.x1, self.x2
```

Question b(i): Use gradient descent with a constant step-size to minimise the loss function starting from initial x = [3, 3]

```
alpha_range = [0.001, 0.01, 0.1]
number_of_iterations = 100

contour_plot_params = {}
plot.figure(figsize=(11, 6), dpi=150)
for alpha in alpha_range :
    x1, x2, f = [], [], []
    constant_step_size_sgd = Stochastic_Gradient_Descent([3, 3])
    for _ in range(number_of_iterations):
        x1_values, x2_values, function_values = constant_step_size_sgd.execute_stochastic_gradient_descent(gradient_descent_type =
'Constant Step Size', alpha = alpha, batch_size = 25)
    x1.append(x1_values)
    x2.append(x2_values)
    f.append(function_values)

    plot.plot(range(number_of_iterations + 1), function_values, label = f'α = {alpha}')
    contour_plot_params[alpha] = (x1, x2, f)
plot.xlabel('#Iteration')
plot.ylabel('Function Value: f([X1, X2], training_data))')
plot.title('Plot of Function Value vs. Iteration  for varying α')
plot.yscale("log")
plot.legend()
plot.show()


## Contour Plot
x1 = np.linspace(-8, 4, 200)
x2 = np.linspace(-8, 4, 200)
line_thickness = 5
f = []
training_data = constant_step_size_sgd.generate_trainingdata()

for x1_point in x1 :
    data_point = []
    for x2_point in x2 :
        data_point.append(constant_step_size_sgd.f([x1_point, x2_point], training_data))
    f.append(data_point)
f_x1_x2 = np.array(f)
X1, X2 = np.meshgrid(x1, x2)

plot.figure(figsize=(9, 6), dpi=150)
plot.contour(X1, X2, f_x1_x2, levels=25)
```

```
for alpha in contour_plot_params :
   x1, x2, f = contour_plot_params[alpha]
   plot.plot(x1[0], x2[0], lw = line_thickness, label = f'α = {alpha}')
   line_thickness -= 2


plot.xlabel('X1')
plot.ylabel('X2')
plot.legend()
plot.title('Contour plot of f([X1, X2], training_data)) for varying α')
plot.show()
```

Question b(ii): SGD with a mini-batch size of 5 Run the SGD several times and plot how f and x change over time.

```
n_times = 5

number_of_iterations = 25

alpha = 0.1

contour_plot_params_trial = {}
plot.figure(figsize=(11, 6), dpi=150)

for n in range(n_times) :
   x1, x2, f = [], [], []
   constant_step_size_sgd = Stochastic_Gradient_Descent([3, 3])

   for _ in range(number_of_iterations):
      x1_values, x2_values, function_values = constant_step_size_sgd.execute_stochastic_gradient_descent(gradient_descent_type =
'Constant Step Size', alpha = alpha, batch_size = 5)
   x1.append(x1_values)
   x2.append(x2_values)
   f.append(function_values)

   plot.plot(range(number_of_iterations + 1), function_values, label = f'Trial = {n}')
   contour_plot_params_trial[n] = (x1, x2, f)
plot.xlabel('#Iteration')
plot.ylabel('Function Value: f([X1, X2], training_data))')
plot.title(f'Plot of Function Value vs. Iteration  for multiple trials with Batch Size = 5 and α = {alpha}')
plot.yscale("log")
plot.legend()
plot.show()



## Contour Plot
x1 = np.linspace(-8, 4, 200)
x2 = np.linspace(-8, 4, 200)
line_thickness = 5
f = []
training_data = constant_step_size_sgd.generate_trainingdata()

for x1_point in x1 :
   data_point = []
   for x2_point in x2 :
      data_point.append(constant_step_size_sgd.f([x1_point, x2_point], training_data))
   f.append(data_point)
f_x1_x2 = np.array(f)
X1, X2 = np.meshgrid(x1, x2)

plot.figure(figsize=(9, 6), dpi=150)
plot.contour(X1, X2, f_x1_x2, levels=25)

for n in contour_plot_params_trial :
   x1, x2, f = contour_plot_params_trial[n]
   plot.plot(x1[0], x2[0], lw = line_thickness, label = f'Trial = {n}')
   line_thickness -= 1
```

```
plot.xlabel('X1')
plot.ylabel('X2')
plot.legend()
plot.title(f'Contour plot of f([X1, X2], training_data)) for multiple trials with Batch Size = 5 and α = {alpha}')
plot.show()
```

Question b(iii): SGD with step size fixed, vary the mini-batch size

```
batch_sizes = [1, 5, 10, 15, 20 , 25]

number_of_iterations = 25

alpha = 0.1

contour_plot_batch_size_trial = {}
plot.figure(figsize=(11, 6), dpi=150)

for batch_size in batch_sizes :
   x1, x2, f = [], [], []
   constant_step_size_sgd = Stochastic_Gradient_Descent([3, 3])

   for _ in range(number_of_iterations):
      x1_values, x2_values, function_values = constant_step_size_sgd.execute_stochastic_gradient_descent(gradient_descent_type =
'Constant Step Size', alpha = alpha, batch_size = batch_size)
   x1.append(x1_values)
   x2.append(x2_values)
   f.append(function_values)

   plot.plot(range(number_of_iterations + 1), function_values, label = f'Batch Size = {batch_size}')
   contour_plot_batch_size_trial[batch_size] = (x1, x2, f)
plot.xlabel('#Iteration')
plot.ylabel('Function Value: f([X1, X2], training_data))')
plot.title('Plot of Function Value vs. Iteration for varying batch sizes with α = 0.1')
plot.yscale("log")
plot.legend()
plot.show()


## Contour Plot
x1 = np.linspace(-8, 4, 200)
x2 = np.linspace(-8, 4, 200)
line_thickness = 6
f = []
training_data = constant_step_size_sgd.generate_trainingdata()

for x1_point in x1 :
   data_point = []
   for x2_point in x2 :
      data_point.append(constant_step_size_sgd.f([x1_point, x2_point], training_data))
   f.append(data_point)
f_x1_x2 = np.array(f)
X1, X2 = np.meshgrid(x1, x2)

plot.figure(figsize=(9, 6), dpi=150)
plot.contour(X1, X2, f_x1_x2, levels=25)

for batch_size in contour_plot_batch_size_trial :
   x1, x2, f = contour_plot_batch_size_trial[batch_size]
   plot.plot(x1[0], x2[0], lw = line_thickness, label = f'Batch Size = {batch_size}')
   line_thickness -= 1


plot.xlabel('X1')
plot.ylabel('X2')
plot.legend()
plot.title('Contour plot of f([X1, X2], training_data)) for varying batch sizes with α = 0.1')
plot.show()
```

Question b(iv): SGD with the mini-batch size fixed at 5, vary the step size

```
batch_size = 5

number_of_iterations = 25

step_sizes = [0.001, 0.01, 0.1]

contour_plot_alpha_trial = {}
plot.figure(figsize=(11, 6), dpi=150)

for alpha in step_sizes :
   x1, x2, f = [], [], []
   constant_step_size_sgd = Stochastic_Gradient_Descent([3, 3])

   for _ in range(number_of_iterations):
      x1_values, x2_values, function_values = constant_step_size_sgd.execute_stochastic_gradient_descent(gradient_descent_type =
'Constant Step Size', alpha = alpha, batch_size = batch_size)
   x1.append(x1_values)
   x2.append(x2_values)
   f.append(function_values)

   plot.plot(range(number_of_iterations + 1), function_values, label = f'α = {alpha}')
   contour_plot_alpha_trial[alpha] = (x1, x2, f)
plot.xlabel('#Iteration')
plot.ylabel('Function Value: f([X1, X2], training_data))')
plot.title('Plot of Function Value vs. Iteration for varying α with Batch Size = 5')
plot.yscale("log")
plot.legend()
plot.show()

## Contour Plot
x1 = np.linspace(-8, 4, 200)
x2 = np.linspace(-8, 4, 200)
line_thickness = 6
f = []
training_data = constant_step_size_sgd.generate_trainingdata()

for x1_point in x1 :
   data_point = []
   for x2_point in x2 :
      data_point.append(constant_step_size_sgd.f([x1_point, x2_point], training_data))
   f.append(data_point)
f_x1_x2 = np.array(f)
X1, X2 = np.meshgrid(x1, x2)

plot.figure(figsize=(9, 6), dpi=150)
plot.contour(X1, X2, f_x1_x2, levels=25)

for alpha in contour_plot_alpha_trial :
   x1, x2, f = contour_plot_alpha_trial[alpha]
   plot.plot(x1[0], x2[0], lw = line_thickness, label = f'α = {alpha}')
   line_thickness -= 1


plot.xlabel('X1')
plot.ylabel('X2')
plot.legend()
plot.title('Contour plot of f([X1, X2], training_data)) for varying α with Batch Size = 5')
plot.show()
```

Appendix: Code for Question c – All Parts

```
import numpy as np
import sympy
from sympy import Heaviside
import random
import math
import matplotlib.pyplot as plot

class Stochastic_Gradient_Descent :

    def __init__(self, starting_point  = [1, 1]) :
        self.train_data = self.generate_trainingdata()
        self.x1 = starting_point[0]
        self.x2 = starting_point[1]
        self.x1_values = np.array([self.x1])
        self.x2_values = np.array([self.x2])
        self.function_values = np.array([self.f([self.x1, self.x2], self.train_data)])
        self.x_z = self.y_z = self.x_v = self.x_m = self.y_v = self.y_m = self.adam_iteration = 0

    def f(self, 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(24*(z[0]**2+z[1]**2), (z[0]+3)**2+(z[1]+3)**2)
            count=count+1
        return y/count

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

    def get_derivative_value(self, x1, w1, x2, w2) :
        derivative_x1_value = (-48*w1 + 48*x1 - 48)*Heaviside(-24*(-w1 + x1 - 1)**2 + (-w1 + x1 + 2)**2 - 24*(-w2 +
x2 - 1)**2 + (-w2 + x2 + 2)**2) + (-2*w1 + 2*x1 + 4)*Heaviside(24*(-w1 + x1 - 1)**2 - (-w1 + x1 + 2)**2 + 24*(-w2
+ x2 - 1)**2 - (-w2 + x2 + 2)**2)
        derivative_x2_value = (-48*w2 + 48*x2 - 48)*Heaviside(-24*(-w1 + x1 - 1)**2 + (-w1 + x1 + 2)**2 - 24*(-w2 +
x2 - 1)**2 + (-w2 + x2 + 2)**2) + (-2*w2 + 2*x2 + 4)*Heaviside(24*(-w1 + x1 - 1)**2 - (-w1 + x1 + 2)**2 + 24*(-w2
+ x2 - 1)**2 - (-w2 + x2 + 2)**2)
        return derivative_x1_value, derivative_x2_value

    def get_batch_derivative(self, x1, x2, sample_batch) :
        sum_derivative_x1_value = 0
        sum_derivative_x2_value = 0

        for w1, w2 in sample_batch :
            derivative_x1_value, derivative_x2_value = self.get_derivative_value(x1, w1, x2, w2)
            sum_derivative_x1_value += derivative_x1_value
            sum_derivative_x2_value += derivative_x2_value

        batch_derivative_x1_value = sum_derivative_x1_value / len(sample_batch)
        batch_derivative_x2_value = sum_derivative_x2_value / len(sample_batch)

        return batch_derivative_x1_value, batch_derivative_x2_value
```

```
  def execute_stochastic_gradient_descent(self, gradient_descent_type = 'Constant', alpha = 0.1, beta = 0.9,
beta2 = 0.999, batch_size = 10) :

    np.random.shuffle(self.train_data)
    for j in range(0, len(self.train_data), batch_size) :

      if (j + batch_size) > len(self.train_data) :
        continue

      if gradient_descent_type == 'Constant Step Size' :
        self.x1, self.x2 = self.execute_constant_size_gradient_descent(self.train_data[j : (j + batch_size)], alpha)

      elif gradient_descent_type == 'Polyak' :
        self.x1, self.x2 = self.execute_polyak_gradient_descent(self.train_data[j : (j + batch_size)])

      elif gradient_descent_type == 'RMSProp':
        self.x1, self.x2 = self.execute_rmsprop_gradient_descent(self.train_data[j : (j + batch_size)], alpha, beta)

      elif gradient_descent_type == 'HeavyBall':
        self.x1, self.x2 = self.execute_heavyball_gradient_descent(self.train_data[j : (j + batch_size)], alpha, beta)

      elif gradient_descent_type == 'Adam':
        self.x1, self.x2 = self.execute_adam_gradient_descent(self.train_data[j : (j + batch_size)], alpha, beta,
beta2)


    self.x1_values = np.append(self.x1_values, [self.x1], axis = 0)
    self.x2_values = np.append(self.x2_values, [self.x2], axis = 0)
    self.function_values = np.append(self.function_values, [self.f([self.x1, self.x2], self.train_data)])

    return self.x1_values, self.x2_values, self.function_values

  def execute_constant_size_gradient_descent(self, sample_batch, alpha) :
    batch_derivative_x1, batch_derivative_x2 = self.get_batch_derivative(self.x1, self.x2, sample_batch)
    self.x1 -= alpha * batch_derivative_x1
    self.x2 -= alpha * batch_derivative_x2
    return self.x1, self.x2

  def execute_polyak_gradient_descent(self, sample_batch) :
    epsilon = 1e-8
    numerator = self.f([self.x1, self.x2], sample_batch)
    batch_derivative_x1, batch_derivative_x2 = self.get_batch_derivative(self.x1, self.x2, sample_batch)
    denominator = ((batch_derivative_x1) **2 + (batch_derivative_x2) ** 2) + epsilon
    step =  (numerator / denominator) if denominator !=0  else 0
    self.x1 = self.x1 - step * batch_derivative_x1
    self.x2 = self.x2 - step * batch_derivative_x2

    return self.x1, self.x2

  def execute_rmsprop_gradient_descent(self, sample_batch, alpha, beta) :
    epsilon = 1e-8
    x_alpha = alpha
    y_alpha = alpha
    x_sum = 0
    y_sum = 0
```

```
        batch_derivative_x1, batch_derivative_x2 = self.get_batch_derivative(self.x1, self.x2, sample_batch)
        x_sum = (x_sum * beta) + ((1 - beta) * ((batch_derivative_x1) ** 2))
        y_sum = (y_sum * beta) + ((1 - beta) * ((batch_derivative_x2) ** 2))

        x_alpha = alpha / (math.sqrt(x_sum) + epsilon)
        y_alpha = alpha / (math.sqrt(y_sum) + epsilon)

        self.x1 = self.x1 - (x_alpha * batch_derivative_x1)
        self.x2 = self.x2 - (y_alpha * batch_derivative_x2)


        return self.x1, self.x2

    def execute_heavyball_gradient_descent(self, sample_batch, alpha, beta) :
        epsilon = 1e-8
        batch_derivative_x1, batch_derivative_x2 = self.get_batch_derivative(self.x1, self.x2, sample_batch)
        self.x_z = (self.x_z * beta) + (alpha * batch_derivative_x1)
        self.y_z = (self.y_z * beta) + (alpha * batch_derivative_x2)

        self.x1 = self.x1 - self.x_z
        self.x2 = self.x2 - self.y_z

        return self.x1, self.x2

    def execute_adam_gradient_descent(self, sample_batch, alpha, beta_1, beta_2) :
        epsilon = 1e-8

        batch_derivative_x1, batch_derivative_x2 = self.get_batch_derivative(self.x1, self.x2, sample_batch)

        self.x_m = (self.x_m * beta_1) + ((1 - beta_1) * batch_derivative_x1)
        self.y_m = (self.y_m * beta_1) + ((1 - beta_1) * batch_derivative_x2)

        self.x_v = (self.x_v * beta_2) + ((1 - beta_2) * ((batch_derivative_x1) ** 2))
        self.y_v = (self.y_v * beta_2) + ((1 - beta_2) * ((batch_derivative_x2) ** 2))

        x_m_hat = self.x_m / (1 - beta_1 ** self.adam_iteration + 1)
        y_m_hat = self.y_m / (1 - beta_1 ** self.adam_iteration + 1)

        x_v_hat = self.x_v / (1 - beta_2 ** self.adam_iteration + 1)
        y_v_hat = self.y_v / (1 - beta_2 ** self.adam_iteration + 1)

        x_alpha = x_m_hat / (math.sqrt(x_v_hat) + epsilon)
        y_alpha = y_m_hat / (math.sqrt(y_v_hat) + epsilon)

        self.x1 = self.x1 - (x_alpha * alpha)
        self.x2 = self.x2 - (y_alpha * alpha)

        self.adam_iteration += 1

        return self.x1, self.x2
```

Question c(i): Polyak with SGD

```
baseline_alpha = 0.1
baseline_batch_size = 5
number_of_iterations = 100
```

```
plot.figure(figsize=(11, 6), dpi=150)
x1, x2, f = [], [], []

##Baseline
constant_step_size_sgd = Stochastic_Gradient_Descent([3, 3])
contour_polyak_plot_params = {}
for _ in range(number_of_iterations):
   x1_values, x2_values, function_values =
constant_step_size_sgd.execute_stochastic_gradient_descent(gradient_descent_type = 'Constant Step Size',
alpha = baseline_alpha, batch_size = baseline_batch_size)
x1.append(x1_values)
x2.append(x2_values)
f.append(function_values)
plot.plot(range(number_of_iterations + 1), function_values, lw = 2, linestyle='dashed',  label = f'Baseline Batch
Size = 5')
contour_polyak_plot_params['Baseline'] = (x1, x2, f)


## Polyak
batch_sizes = [1, 5, 10, 15, 20, 25]
for batch_size in batch_sizes :
   x1, x2, f = [], [], []
   polyak_sgd = Stochastic_Gradient_Descent([3, 3])

   for _ in range(number_of_iterations):
      x1_values, x2_values, function_values =
polyak_sgd.execute_stochastic_gradient_descent(gradient_descent_type = 'Polyak', batch_size = batch_size)
   x1.append(x1_values)
   x2.append(x2_values)
   f.append(function_values)

   plot.plot(range(number_of_iterations + 1), function_values, label = f'Polyak Batch Size = {batch_size}')
   contour_polyak_plot_params[batch_size] = (x1, x2, f)
plot.ylim([0, 60])
plot.xlabel('#Iteration')
plot.ylabel('Function Value: f([X1, X2], training_data))')
plot.title('Plot of Function Value vs. Iteration  for varying batch sizes for Polyak steps')
#plot.yscale("log")
plot.legend()
plot.show()

## Contour Plot
x1 = np.linspace(-8, 4, 200)
x2 = np.linspace(-8, 4, 200)
line_thickness = 5
f = []
training_data = polyak_sgd.generate_trainingdata()

for x1_point in x1 :
   data_point = []
   for x2_point in x2 :
      data_point.append(polyak_sgd.f([x1_point, x2_point], training_data))
   f.append(data_point)
f_x1_x2 = np.array(f)
```

```
X1, X2 = np.meshgrid(x1, x2)

plot.figure(figsize=(9, 6), dpi=150)
plot.contour(X1, X2, f_x1_x2, levels=25)

for batch_size in contour_polyak_plot_params :
   x1, x2, f = contour_polyak_plot_params[batch_size]
   if batch_size == 'Baseline' :
      plot.plot(x1[0], x2[0], lw = 2, linestyle = 'dashed', label = f'Baseline Batch Size = 5')
   else :
      plot.plot(x1[0], x2[0], lw = line_thickness, label = f'Polyak Batch Size = {batch_size}')
   line_thickness -= 1

plot.ylim([-8, 4])
plot.xlim([-8, 4])
plot.xlabel('X1')
plot.ylabel('X2')
plot.legend()
plot.title('Contour plot of f([X1, X2], training_data)) for varying batch sizes for Polyak steps')
plot.show()
```

Question c(ii): RMSProp with SGD

```
plot.figure(figsize=(11, 6), dpi=150)
## RMSProp
alpha_range = [0.001, 0.01, 0.1]
number_of_iterations = 100
baseline_batch_size = 5
beta_range = [0.25, 0.9]
for beta in beta_range :
   for alpha in alpha_range :
      rmsprop_sgd = Stochastic_Gradient_Descent([3, 3])
      for _ in range(number_of_iterations):
         x1_values, x2_values, function_values =
rmsprop_sgd.execute_stochastic_gradient_descent(gradient_descent_type = 'RMSProp', alpha = alpha, beta =
beta, batch_size = baseline_batch_size)

      plot.plot(range(number_of_iterations + 1), function_values, label = f'α = {alpha}, β = {beta}')

#plot.ylim([0, 60])
plot.xlabel('#Iteration')
plot.ylabel('Function Value: f([X1, X2], training_data))')
plot.title('Plot of Function Value vs. Iteration  for varying α and β for RMSProp steps')
plot.yscale("log")
plot.legend()
plot.show()

baseline_alpha = 0.1
baseline_batch_size = 5
number_of_iterations = 100


plot.figure(figsize=(11, 6), dpi=150)
x1, x2, f = [], [], []
```

```
##Baseline
constant_step_size_sgd = Stochastic_Gradient_Descent([3, 3])
contour_polyak_plot_params = {}
for _ in range(number_of_iterations):
   x1_values, x2_values, function_values =
constant_step_size_sgd.execute_stochastic_gradient_descent(gradient_descent_type = 'Constant Step Size',
alpha = baseline_alpha, batch_size = baseline_batch_size)
x1.append(x1_values)
x2.append(x2_values)
f.append(function_values)
plot.plot(range(number_of_iterations + 1), function_values, lw = 2, linestyle='dashed',  label = f'Baseline Batch
Size = 5')
contour_polyak_plot_params['Baseline'] = (x1, x2, f)


## RMSProp
batch_sizes = [1, 5, 10, 15, 20, 25]
optimised_alpha = 0.1
optimised_beta = 0.9
for batch_size in batch_sizes :
   x1, x2, f = [], [], []
   rmsprop_sgd = Stochastic_Gradient_Descent([3, 3])

   for _ in range(number_of_iterations):
      x1_values, x2_values, function_values =
rmsprop_sgd.execute_stochastic_gradient_descent(gradient_descent_type = 'RMSProp', alpha =
optimised_alpha, beta = optimised_beta, batch_size = batch_size)

   x1.append(x1_values)
   x2.append(x2_values)
   f.append(function_values)

   plot.plot(range(number_of_iterations + 1), function_values, label = f'RMSProp Batch Size = {batch_size}')
   contour_polyak_plot_params[batch_size] = (x1, x2, f)
#plot.ylim([0, 60])
plot.xlabel('#Iteration')
plot.ylabel('Function Value: f([X1, X2], training_data))')
plot.title('Plot of Function Value vs. Iteration  for varying batch sizes for RMSProp steps with α = 0.1, β = 0.9')
plot.yscale("log")
plot.legend()
plot.show()

## Contour Plot
x1 = np.linspace(-8, 4, 200)
x2 = np.linspace(-8, 4, 200)
line_thickness = 5
f = []
training_data = rmsprop_sgd.generate_trainingdata()

for x1_point in x1 :
   data_point = []
   for x2_point in x2 :
      data_point.append(rmsprop_sgd.f([x1_point, x2_point], training_data))
   f.append(data_point)
f_x1_x2 = np.array(f)
X1, X2 = np.meshgrid(x1, x2)
```

```
plot.figure(figsize=(9, 6), dpi=150)
plot.contour(X1, X2, f_x1_x2, levels=25)

for batch_size in contour_polyak_plot_params :
   x1, x2, f = contour_polyak_plot_params[batch_size]
   if batch_size == 'Baseline' :
      plot.plot(x1[0], x2[0], lw = 2, linestyle = 'dashed', label = f'Baseline Batch Size = 5')
   else :
      plot.plot(x1[0], x2[0], lw = line_thickness, label = f'RMSProp Batch Size = {batch_size}')
   line_thickness -= 1

plot.ylim([-8, 4])
plot.xlim([-8, 4])
plot.xlabel('X1')
plot.ylabel('X2')
plot.legend()
plot.title('Contour plot of f([X1, X2], training_data)) for varying batch sizes for RMSProp steps with α = 0.1, β =
0.9')
plot.show()
```

Question c(iii): Heavy Ball with SGD

```
plot.figure(figsize=(11, 6), dpi=150)
## Heavyball
alpha_range = [0.001, 0.01, 0.1]
beta_range = [0.25, 0.9]
for beta in beta_range :
   for alpha in alpha_range :
      heavyball_sgd = Stochastic_Gradient_Descent([3, 3])
      for _ in range(number_of_iterations):
         x1_values, x2_values, function_values =
heavyball_sgd.execute_stochastic_gradient_descent(gradient_descent_type = 'HeavyBall', alpha = alpha, beta =
beta, batch_size = 5)

      plot.plot(range(number_of_iterations + 1), function_values, label = f'α = {alpha}, β = {beta}')

#plot.ylim([0, 60])
plot.xlabel('#Iteration')
plot.ylabel('Function Value: f([X1, X2], training_data))')
plot.title('Plot of Function Value vs. Iteration  for varying α and β  for Heavyball steps')
plot.yscale("log")
plot.legend()
plot.show()

baseline_alpha = 0.1
baseline_batch_size = 5
number_of_iterations = 100


plot.figure(figsize=(11, 6), dpi=150)
x1, x2, f = [], [], []

##Baseline
constant_step_size_sgd = Stochastic_Gradient_Descent([3, 3])
```

```
contour_polyak_plot_params = {}
for _ in range(number_of_iterations):
    x1_values, x2_values, function_values =
constant_step_size_sgd.execute_stochastic_gradient_descent(gradient_descent_type = 'Constant Step Size',
alpha = baseline_alpha, batch_size = baseline_batch_size)
x1.append(x1_values)
x2.append(x2_values)
f.append(function_values)
plot.plot(range(number_of_iterations + 1), function_values, lw = 2, linestyle='dashed',  label = f'Baseline Batch
Size = 5')
contour_polyak_plot_params['Baseline'] = (x1, x2, f)


## Heavyball
batch_sizes = [1, 5, 10, 15, 20, 25]
optimised_alpha = 0.1
optimised_beta = 0.25
for batch_size in batch_sizes :
    x1, x2, f = [], [], []
    heavyball_sgd = Stochastic_Gradient_Descent([3, 3])

    for _ in range(number_of_iterations):
        x1_values, x2_values, function_values =
heavyball_sgd.execute_stochastic_gradient_descent(gradient_descent_type = 'HeavyBall', alpha =
optimised_alpha, beta = optimised_beta, batch_size = batch_size)

    x1.append(x1_values)
    x2.append(x2_values)
    f.append(function_values)

    plot.plot(range(number_of_iterations + 1), function_values, label = f'Heavy Ball Batch Size = {batch_size}')
    contour_polyak_plot_params[batch_size] = (x1, x2, f)
#plot.ylim([0, 60])
plot.xlabel('#Iteration')
plot.ylabel('Function Value: f([X1, X2], training_data))')
plot.title('Plot of Function Value vs. Iteration  for varying batch sizes for Heavy Ball steps with α = 0.1, β = 0.25')
plot.yscale("log")
plot.legend()
plot.show()

## Contour Plot
x1 = np.linspace(-8, 4, 200)
x2 = np.linspace(-8, 4, 200)
line_thickness = 5
f = []
training_data = heavyball_sgd.generate_trainingdata()

for x1_point in x1 :
    data_point = []
    for x2_point in x2 :
        data_point.append(heavyball_sgd.f([x1_point, x2_point], training_data))
    f.append(data_point)
f_x1_x2 = np.array(f)
X1, X2 = np.meshgrid(x1, x2)

plot.figure(figsize=(9, 6), dpi=150)
```

```
plot.contour(X1, X2, f_x1_x2, levels=25)

for batch_size in contour_polyak_plot_params :
   x1, x2, f = contour_polyak_plot_params[batch_size]
   if batch_size == 'Baseline' :
      plot.plot(x1[0], x2[0], lw = 2, linestyle = 'dashed', label = f'Baseline Batch Size = 5')
   else :
      plot.plot(x1[0], x2[0], lw = line_thickness, label = f'Heavy Ball Batch Size = {batch_size}')
   line_thickness -= 1

plot.ylim([-8, 4])
plot.xlim([-8, 4])
plot.xlabel('X1')
plot.ylabel('X2')
plot.legend()
plot.title('Contour plot of f([X1, X2], training_data)) for varying batch sizes for Heavy Ball steps with α = 0.1, β =
0.25')
plot.show()
```

Question c(iv): Adam with SGD

```
plot.figure(figsize=(11, 6), dpi=150)
## Adam
alpha_range = [0.001, 0.01, 0.1]
beta_range = [0.25, 0.9]
beta_2 = 0.9
for beta in beta_range :
   for alpha in alpha_range :
      adam_sgd = Stochastic_Gradient_Descent([3, 3])
      for _ in range(number_of_iterations):
         x1_values, x2_values, function_values =
adam_sgd.execute_stochastic_gradient_descent(gradient_descent_type = 'Adam', alpha = alpha, beta = beta,
beta2 = beta_2, batch_size = 5)

      plot.plot(range(number_of_iterations + 1), function_values, label = f'α = {alpha}, β1 = {beta}, β2 = {beta_2}')

#plot.ylim([0, 60])
plot.xlabel('#Iteration')
plot.ylabel('Function Value: f([X1, X2], training_data))')
plot.title('Plot of Function Value vs. Iteration  for varying α, β1 and β2 for Adam steps')
plot.yscale("log")
plot.legend()
plot.show()

plot.figure(figsize=(11, 6), dpi=150)
## Adam
alpha_range = [0.001, 0.01, 0.1]
beta_range = [0.25, 0.9]
beta_2 = 0.999
for beta in beta_range :
   for alpha in alpha_range :
      adam_sgd = Stochastic_Gradient_Descent([3, 3])
      for _ in range(number_of_iterations):
```

```
        x1_values, x2_values, function_values =
adam_sgd.execute_stochastic_gradient_descent(gradient_descent_type = 'Adam', alpha = alpha, beta = beta,
beta2 = beta_2, batch_size = 5)

    plot.plot(range(number_of_iterations + 1), function_values, label = f'α = {alpha}, β1 = {beta}, β2 = {beta_2}')

#plot.ylim([0, 60])
plot.xlabel('#Iteration')
plot.ylabel('Function Value: f([X1, X2], training_data))')
plot.title('Plot of Function Value vs. Iteration  for varying α, β1 and β2 for Adam steps')
plot.yscale("log")
plot.legend()
plot.show()

baseline_alpha = 0.1
baseline_batch_size = 5
number_of_iterations = 100


plot.figure(figsize=(11, 6), dpi=150)
x1, x2, f = [], [], []

##Baseline
constant_step_size_sgd = Stochastic_Gradient_Descent([3, 3])
contour_polyak_plot_params = {}
for _ in range(number_of_iterations):
  x1_values, x2_values, function_values =
constant_step_size_sgd.execute_stochastic_gradient_descent(gradient_descent_type = 'Constant Step Size',
alpha = baseline_alpha, batch_size = baseline_batch_size)
x1.append(x1_values)
x2.append(x2_values)
f.append(function_values)
plot.plot(range(number_of_iterations + 1), function_values, lw = 2, linestyle='dashed',  label = f'Baseline Batch
Size = 5')
contour_polyak_plot_params['Baseline'] = (x1, x2, f)


## Adam
batch_sizes = [1, 5, 10, 15, 20, 25]
optimised_alpha = 0.1
optimised_beta1 = 0.9
optimised_beta2 = 0.999
for batch_size in batch_sizes :
  x1, x2, f = [], [], []
  adam_sgd = Stochastic_Gradient_Descent([3, 3])

  for _ in range(number_of_iterations):
    x1_values, x2_values, function_values =
adam_sgd.execute_stochastic_gradient_descent(gradient_descent_type = 'Adam', alpha = optimised_alpha, beta
= optimised_beta1, beta2 = optimised_beta2, batch_size = batch_size)

  x1.append(x1_values)
  x2.append(x2_values)
  f.append(function_values)

  plot.plot(range(number_of_iterations + 1), function_values, label = f'Adam Batch Size = {batch_size}')
```

```
    contour_polyak_plot_params[batch_size] = (x1, x2, f)
#plot.ylim([0, 60])
plot.xlabel('#Iteration')
plot.ylabel('Function Value: f([X1, X2], training_data))')
plot.title('Plot of Function Value vs. Iteration  for varying batch sizes for Adam steps with α = 0.10, β1 = 0.9, β2 = 0.999')
plot.yscale("log")
plot.legend()
plot.show()

## Contour Plot
x1 = np.linspace(-8, 4, 200)
x2 = np.linspace(-8, 4, 200)
line_thickness = 5
f = []
training_data = polyak_sgd.generate_trainingdata()

for x1_point in x1 :
    data_point = []
    for x2_point in x2 :
        data_point.append(polyak_sgd.f([x1_point, x2_point], training_data))
    f.append(data_point)
f_x1_x2 = np.array(f)
X1, X2 = np.meshgrid(x1, x2)

plot.figure(figsize=(9, 6), dpi=150)
plot.contour(X1, X2, f_x1_x2, levels=25)

for batch_size in contour_polyak_plot_params :
    x1, x2, f = contour_polyak_plot_params[batch_size]
    if batch_size == 'Baseline' :
        plot.plot(x1[0], x2[0], lw = 2, linestyle = 'dashed', label = f'Baseline Batch Size = 5')
    else :
        plot.plot(x1[0], x2[0], lw = line_thickness, label = f'Adam Batch Size = {batch_size}')
    line_thickness -= 1

plot.ylim([-8, 4])
plot.xlim([-8, 4])
plot.xlabel('X1')
plot.ylabel('X2')
plot.legend()
plot.title('Contour plot of f([X1, X2], training_data)) for varying batch sizes for Adam steps with α = 0.1, β1 = 0.9, β2 = 0.999')
plot.show()
```