Assignment 1

Danielle Buggle 20333829

March 12, 2025

Part A

```
def mini_batch_SGD(X, initial_x0, num_iters, batch_size, step_size_func, alpha0, beta0,
    beta1, epsilon):
    m, n = X.shape
    theta = np.array(initial_x0)  # # Initialize theta (parameters)
    moving_avg = np.zeros_like(theta)  # RMSPROP
    z_t = np.zeros_like(theta)  # Heavy Ball
    m_t = np.zeros_like(theta)  # Adam - First momentum term
    v_t = np.zeros_like(theta)  # Adam - Second momentum term
    t = 1  # Adam - time step=
    theta_history = [theta.copy()]
    f_values = [f(theta, X)]
    epoch_count = 0
    for iter_num in range(num_iters):
        indices = np.random.permutation(m)
        X_shuffled = X[indices]
        epoch_count += 1
        for i in range(0, m, batch_size):
            minibatch = X_shuffled[i:i + batch_size]
            # Calculate the approximate gradient for the current mini-batch
            grad = approx_gradient(f, theta, minibatch)
            if step_size_func == polyak_step_size:
                f_x = f(theta, minibatch)
                alpha = step_size_func(f_x, F_STAR, grad)
            elif step_size_func == rmsprop_step:
                alpha, average = step_size_func(grad, moving_avg, alpha0, beta0)
                moving_avg = average
            elif step_size_func == heavy_ball_step:
                z = heavy_ball_step(z_t, grad, alpha0, beta0)
                alpha = z
            elif step_size_func == adam_step:
                m_t, v_t, alpha = step_size_func(m_t, v_t, grad, alpha0, beta0, beta1,
                    epsilon, t)
                t += 1
            else:
                alpha = alpha0
            theta = theta - (alpha * grad)
            # Append current theta to history
            theta_history.append(theta.copy())
            f_values.append(f(theta, X))

    return np.array(theta_history), np.array(f_values), epoch_count
```

Listing 1: SGD algorithm

(i)

Listing 1 above is the function I created that implements Stochastic Gradient Descent with a choice of constant step size, Polyak, RMSProp, HeavyBall or Adam. It takes in the full training data, the initial start point, the

number of epochs to be run for, the batch size and the name of the step size algorithm to be used along with alpha and beta parameters. First, theta is initialised to the starting point, and any extra parameters needed for any of the step sizes chosen are initialised.

It begins the first run of the full training data by shuffling the training data. Following that, it loops the full training data by splitting it into batch sizes and calculating the approximate derivative of that batch. The approximate derivative is calculated by getting the exact derivative of each value in the batch and then averaging those to return an approximate derivative. The step size is calculated depending on which step size function is chosen. The next theta value, i.e. the next point, is calculated by taking away the step size multiplied by the approximate derivative from the previous theta.

(ii)

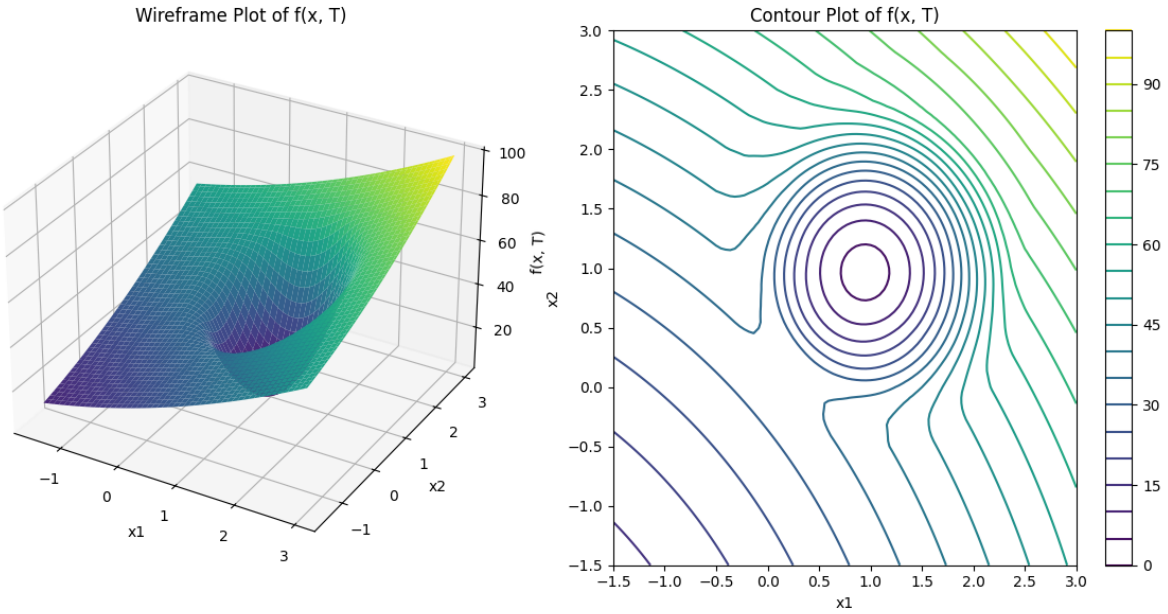


Figure 1: Wireframe Plot and Contour Plot

Figure 1 shows the wireframe and contour plot of the generated training data applied to the loss function. I chose a range of [-1.5, 3] as it focuses on the dip/valley in the function where the algorithms can get stuck because it can appear to be a minimum. I'm also displaying enough of the rest of the function to show the function values decreasing towards [-1, -1] and increasing towards [3, 3].

(iii)

```
def symbolic_f(x, minibatch):
    y = 0
    count = 0
    for w in minibatch:
        # Make symbolic z and use symbolic Min
        z1 = x[0] - w[0] - 1  # x[0] - w[0] - 1
        z2 = x[1] - w[1] - 1  # x[1] - w[1] - 1
        term1 = 37 * (z1 ** 2 + z2 ** 2)
        term2 = (z1 + 5) ** 2 + (z2 + 5) ** 2
        y += sp.Min(term1, term2)
        count += 1
    return y / count
```

Listing 2: Symbolic loss function.

```
def compute_symbolic_derivative(x, minibatch):
    grad_x1 = sp.diff(symbolic_f(x, minibatch), x[0])
    grad_x2 = sp.diff(symbolic_f(x, minibatch), x[1])
    return grad_x1, grad_x2


def numerical_derivative(x1_val, x2_val, grad_x1, grad_x2):
    grad_x1_value = grad_x1.subs({x1: x1_val, x2: x2_val})
    grad_x2_value = grad_x2.subs({x1: x1_val, x2: x2_val})
    return grad_x1_value, grad_x2_value
```

Listing 3: Functions to calculate symbolic and numerical derivative.

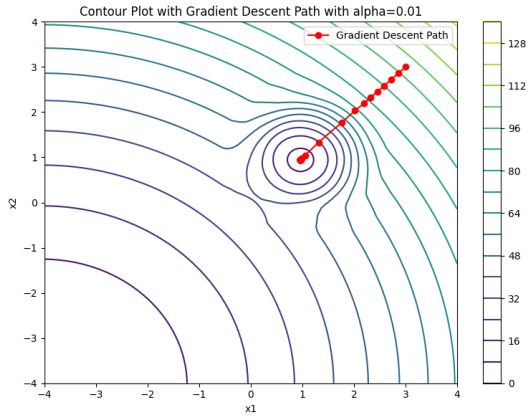
Listings 2 and 3 show the code I used to calculate the exact derivative using sympy. First of all, I converted the loss function provided into a symbolic function. This is the exact same as the provided function; it just uses `sp.Min` instead to make it symbolic, allowing for differentiation.

When calculating the exact derivative at a point, I first call `compute_symbolic_derivative` which finds the symbolic derivative of the function with respect to x1, and then x2, returning both. Then, I call `numerical_derivative` to evaluate the derivative numerically at a provided point - e.g.[3,3] (`x1_val = 3, x2_val = 3`) using the symbolic gradients from the previous function. It substitutes the x1 and x2 values into the symbolic function, which finds the numerical derivative value at both of the features.

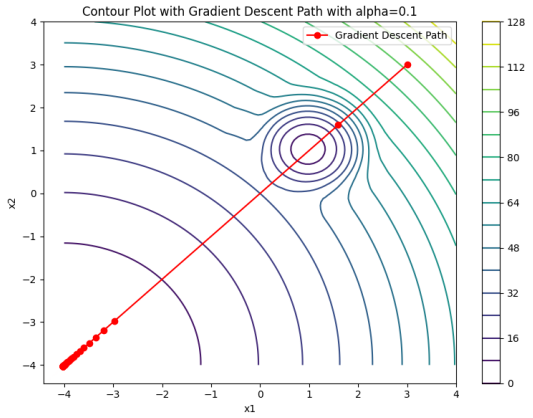
Part B

(i)

First of all, I ran gradient descent twice with different alpha values. Figure 2 provides a comparison of the gradient descent performance with $\alpha = 0.01$ and $\alpha = 0.1$. When $\alpha = 0.1$ in figure 2b, the step size is large, causing gradient descent to overshoot and jump past the valley. As I mentioned in part (a)(ii), the overall



(a) Alpha = 0.01 for gradient descent



(b) Alpha = 0.1 for gradient descent

Figure 2: Comparison of gradient descent with different alpha values

function values decrease towards the range of [-1, -1] and will continue to decrease in that direction, and we can see it has converged to a minimum in that direction. I set the x-axis limit to -4, so all points overlap there. I chose to use $\alpha = 0.01$ to focus on when gradient descent minimises in the valley.

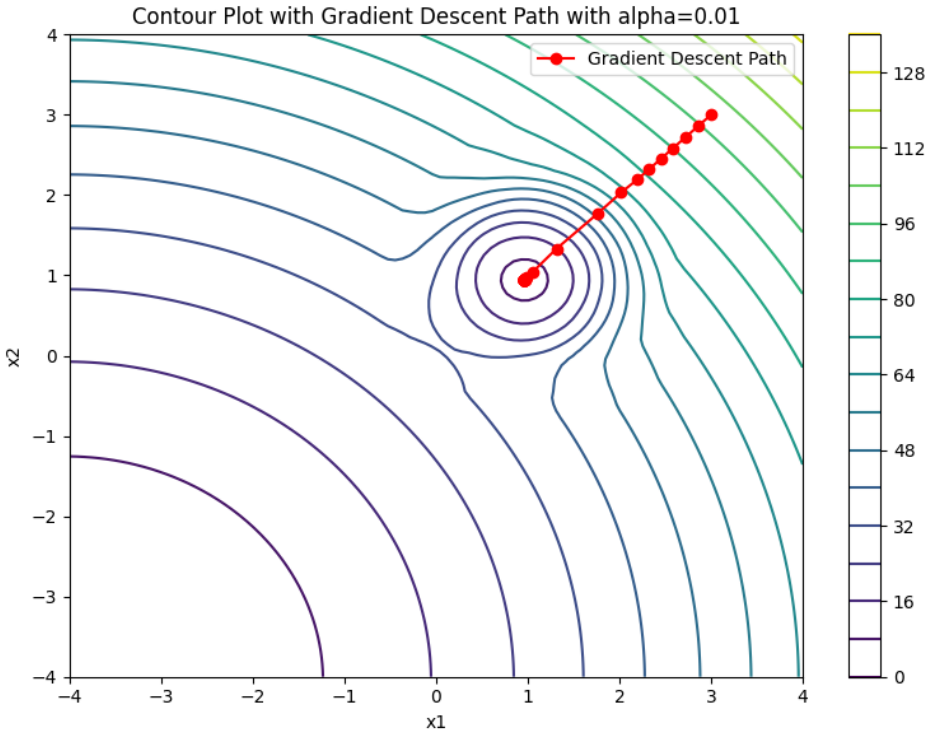


Figure 3: Alpha = 0.01 for gradient descent

Figure 3 shows the full contour plot of gradient descent performance with $\alpha = 0.01$. The performance of gradient descent is quite reasonable here, it starts at the point [3,3], with a high function value of approximately 120, according to the contour legend on the right side. It then follows a path decreasing at each step until it gets caught in the valley converging to a minimum there. This is clear from the dark purple lines in the contour legend indicating low function values.

(ii)

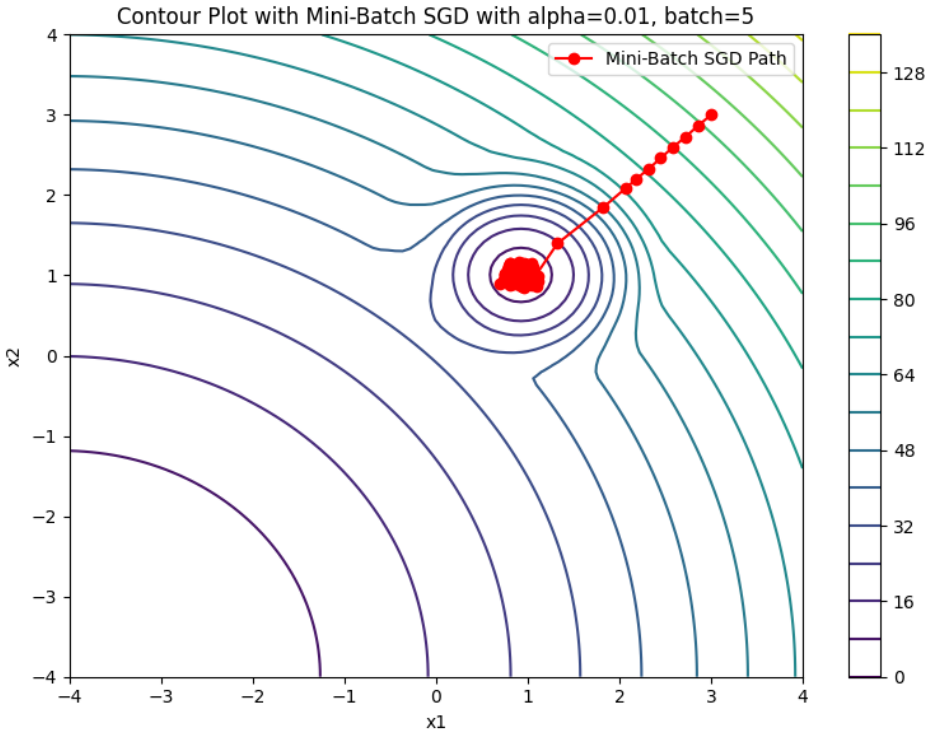
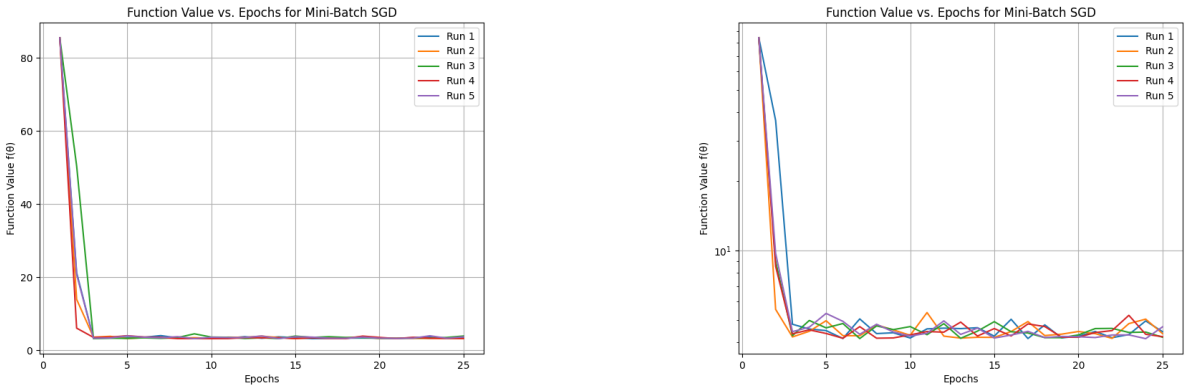


Figure 4: Contour Plot of SGD with Mini-Batch=5

Figure 4 provides a contour plot of one run of SGD with a fixed step size of $\alpha = 0.01$ and a mini-batch size of 5. It performs reasonably well with these parameters, decreasing slowly and entering the valley to converge there. Comparing against figure 3, the only difference is that SGD has a lot more points in the valley because it bounces around in there until it converges.

To compare SGD against itself for different runs, I plotted the function values versus the number of epochs, as seen in figure 5. I reduced the number of epochs for these runs to 25 just for overall run-time; it is shorter, and we can see the same thing if it was 100 epochs, which I used for figure 4.



(a) 5 runs of SGD

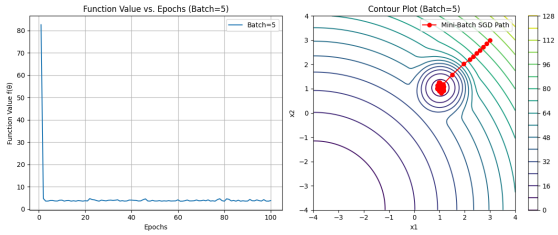
(b) 5 runs of SGD with log scale

Figure 5: Comparison of repeated runs of SGD with a fixed step size and batch-size

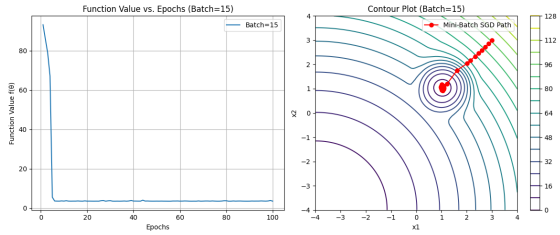
Figure 5 shows SGD run five different times with a normal scale and a log scale. It is clear from both plots that each run tends to perform very similarly. This is because we aren't changing the alpha value or batch size between runs, so they really should perform very similarly. There is only a slight deviation between runs because we get different random batches each time due to the random shuffling of the training data.

(iii)

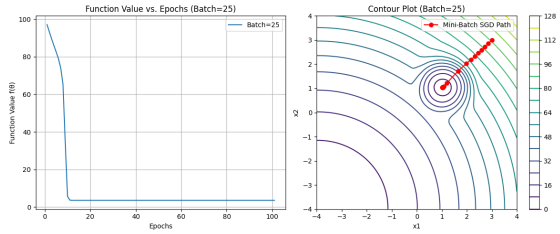
First, I chose my mini-batch size range of [5, 15, 25], stopping at 25 because when generating the training data, there are only 25 data points.



(a) SGD Batch Size of 5



(b) SGD Batch Size of 15



(c) SGD Batch Size of 25

Figure 6: Comparison of SGD with Different Batch Sizes

Comparing figures 6a, 6b, and 6c, looking at the contour plots, each time it converges to a minimum within the valley. The point x tends to wander when near the minimum when it has a smaller batch size because of the "noise" in the approximate derivative.

As the batch size increases, there is less "noise" or bouncing between points in the valley. This is because when calculating the approximate derivative, we are averaging by the batch size. As the batch size increases, we are dividing by a higher number of data points, meaning our approximate derivative is getting closer to the exact derivative.

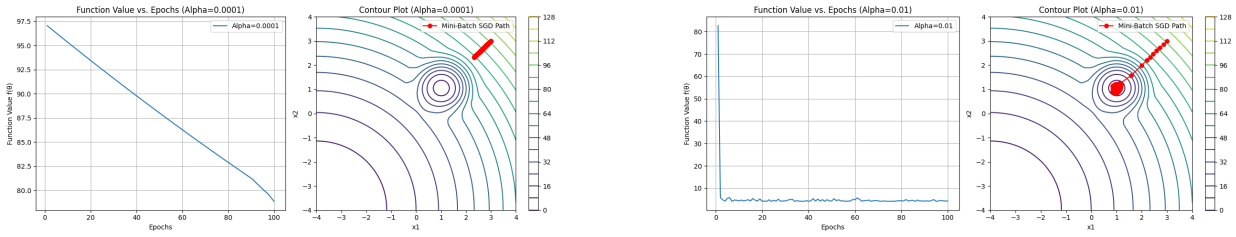
This is all supported by our function value vs epochs plot for each batch size. With a small batch size of 5, the function values bounce a little bit. These bounces are smoothed out when batch size is 15, and then finally for batch size 25, it is completely smooth.

For a batch size of 25 in figure 6c, it has smoothed out completely because now we are averaging by the total number of data points, so our approximate derivative ends up equaling our exact derivative with some noise where the noise is 0. Our noise is 0 because we're calculating the exact derivative of the function at each iteration because the batch size equals the full dataset size.

Overall, increasing the batch size causes the "noise" in SGD to regularise, and it ends up performing very similarly to gradient descent, the only difference being that SGD is more computationally efficient.

(iv)

The range of alpha values I decided to plot were alpha = [0.0001, 0.01, 0.1, 0.5]. For each alpha value I plotted the path it takes on the contour plot and the function values vs epochs. Looking at figures 7a and 7b, when α is a smaller value, it performs the same as before with $\alpha = 0.01$. However, with $\alpha = 0.01$, the step size is too small. It is decreasing steadily but not taking very large steps, meaning it would take much longer to converge to the minimum in the valley, but it would ultimately end up there if given more time. Looking at figures 8a

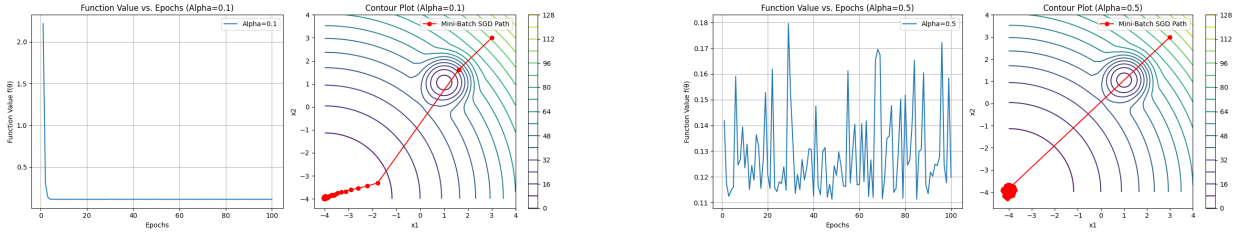


(a) SGD Alpha = 0.0001

(b) SGD Alpha = 0.01

Figure 7: Comparison of SGD with small Alpha Values

and 8b, when α is a larger value, the larger step size causes the algorithm to overshoot past the minimum in the valley. For $\alpha = 0.1$, it has a smooth line in the function value plot, but it has just passed the minimum in the valley and has ended up heading towards the minimum towards the negative sense of the axes. Similarly, for $\alpha = 0.5$, the step size is larger, causing it to jump past the valley and converge towards the negative sense of the axes.



(a) SGD Alpha = 0.1

(b) SGD Alpha = 0.5

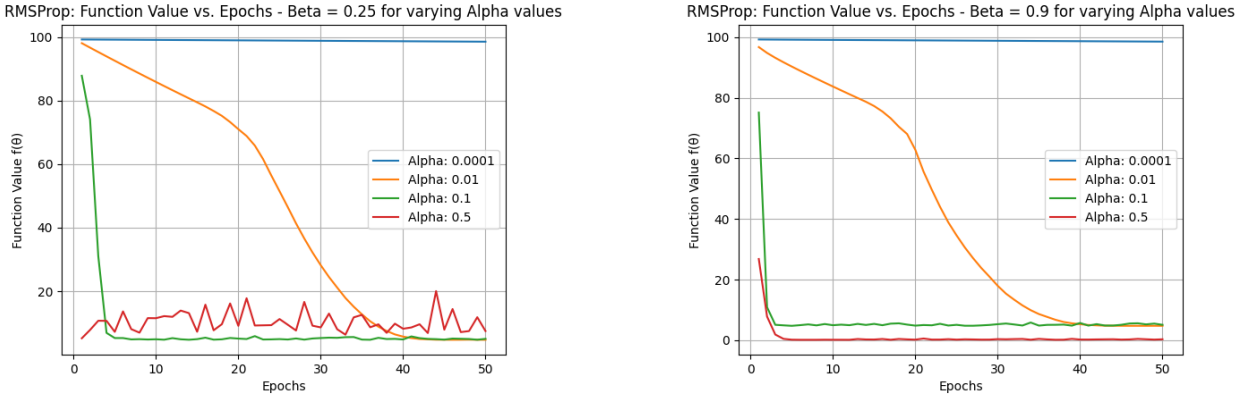
Figure 8: Comparison of SGD with larger Alpha Values

Comparing the SGD algorithm when varying the batch size in (iii) and when we vary the alpha value in (iv), both batch size and learning rate influence the stability and efficiency of SGD, but in different ways. The batch size controls noise and smoothness by averaging gradients, but the learning rate controls step size and convergence speed, where too large a step causes divergence. It is important to tune both of these parameters for the best performance.

Part C

First of all, to choose appropriate parameters, I ran SGD with a batch size of 5 with varying parameters for RMSProp, HeavyBall and Adam. For each, I compared alpha values - [0.0001, 0.01, 0.1, 0.5], beta1 values

[0.25, 0.9] and beta2 values [0.25, 0.99] Figure 9 shows two subplots of beta=0.25 and beta=0.9 with varying

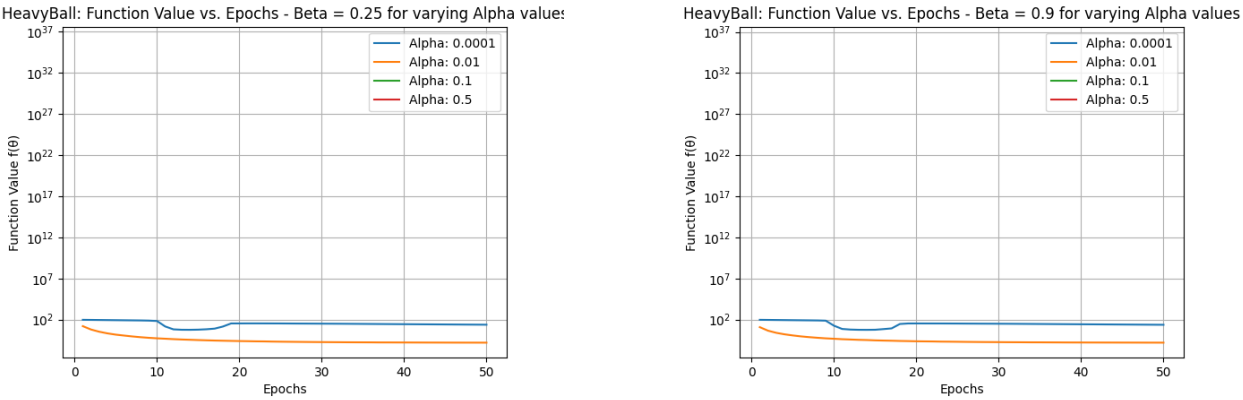


(a) RMSProp Beta = 0.25

(b) RMSProp Beta = 0.9

Figure 9: Comparison of RMSProp parameters

alpha values. As you can see, $\alpha = 0.5$ converges quickly to a minimum. I chose $\beta = 0.25$ because it shows some noise indicating that it is probably bouncing around the valley. If we're focusing on finding the minimum in the valley, this is the behaviour we want. As a result, the parameters I chose for RMSProp were $\alpha = 0.01$ with $\beta = 0.25$.



(a) HeavyBall Beta = 0.25

(b) HeavyBall Beta = 0.9

Figure 10: Comparison of RMSProp parameters

Figure 10 shows a comparison of the varying parameter values for the HeavyBall step size algorithm. As we can see, very large α values don't even appear on the graph because overflow happened, causing the numbers to be impossible to calculate. Both β values perform very similarly, but ultimately, I chose $\alpha = 0.01$ and $\beta = 0.9$.

Figure 11 shows subplots of the various parameter values Adam could have. It's clear from the two bottom plots that the large α value causes Adam to diverge. Focusing on $\alpha = 0.0001$, it decreases at a steady rate for every β value; however, it doesn't reach anywhere close to a minimum within 50 epochs. As a result, I have chosen $\alpha = 0.01$. Focusing on $\alpha = 0.01$, each pair of β values decreases quickly and then jumps again, indicating behaviour that it is going into the valley, then jumping back out and proceeding towards the negative values of the axes where the function values decrease even more.

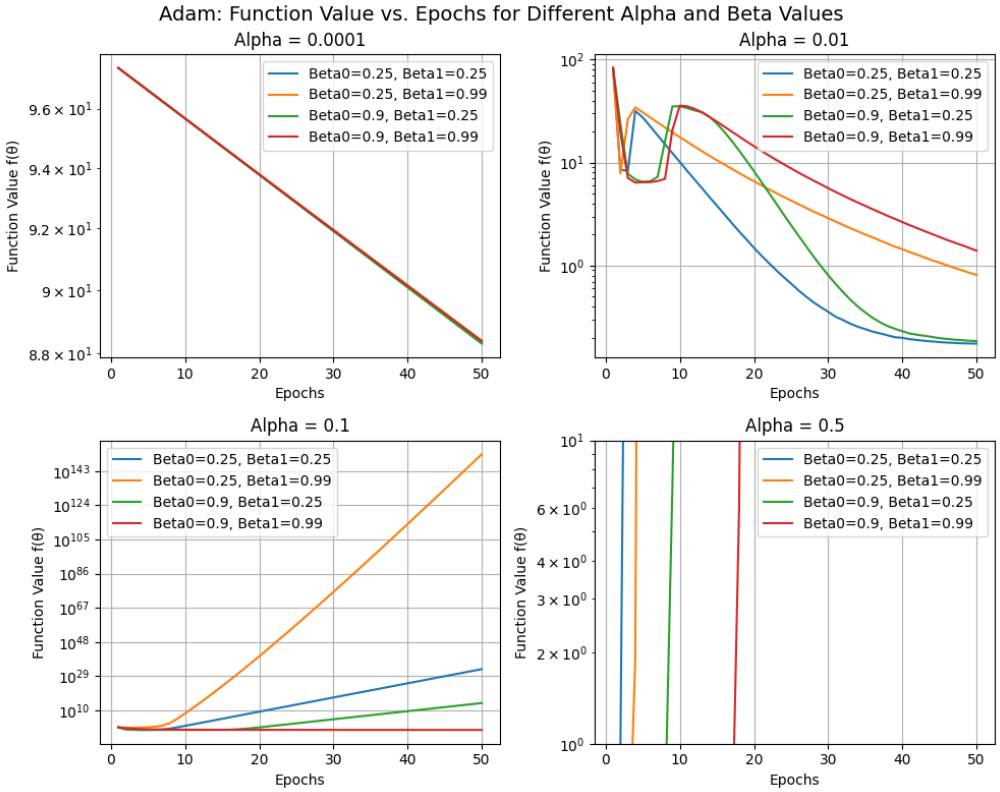


Figure 11: Comparison of Adam parameters

Overall, I have chosen values that will try to search the valley because it is important for the algorithm to check if the minimum is there.

Running each algorithm with the optimal parameters I have chosen above, the figures below show the behaviour of each with varying batch sizes. Figure 12 shows each algorithm with a batch size of 5. I have subplots of

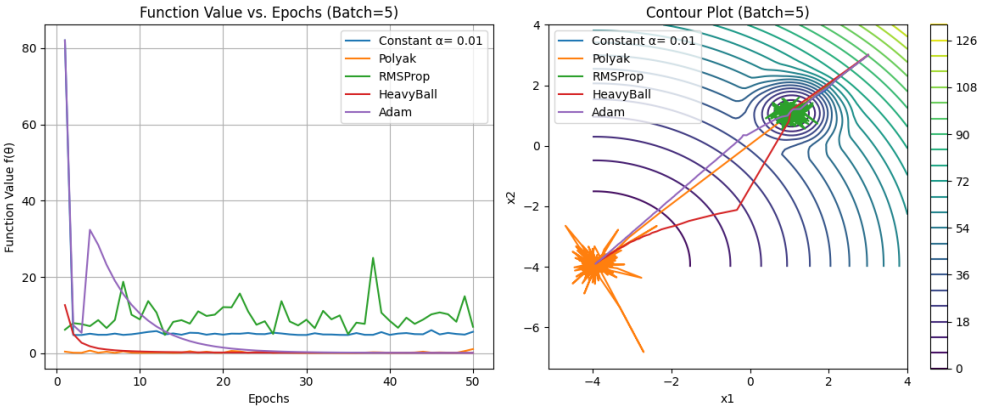


Figure 12: Comparison of algorithms - Batch Size = 5

function values vs epocjs and a contour plot to show its behaviour in the valley specifically. Each algorithm is compared against a baseline from (b) with a constant step size of $\alpha = 0.01$. As we can see, RMSProp converges to the valley and ends up bouncing around in there. The other three algorithms, polyak, heavyball and adam, all end up trying to converge to the minimum outside of the valley - which is smaller than the minimum in the

valley. The only one that searches the valley before jumping out is Adam, and we can see this from the function value plot. Polyak and Heavyball just jump right past the valley and head straight for the overall minimum.

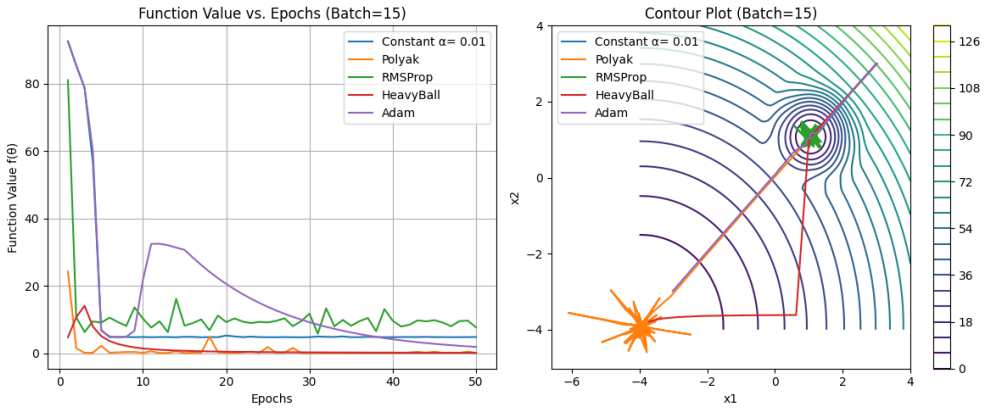


Figure 13: Comparison of algorithms - Batch Size = 15

Figure 13 shows an increase in batch size, and each algorithm performs similarly to before. RMSProp and the baseline end up in the valley, with RMSProp bouncing around in there. However, HeavyBall performs a little differently; it takes a different path, jumping into the valley and then jumping out and heading vertically down the graph, to then head towards the same spot as Polyak. Adam performs similarly again, searching the valley to then exit and head towards the same minimum as Polyak and Heavyball.

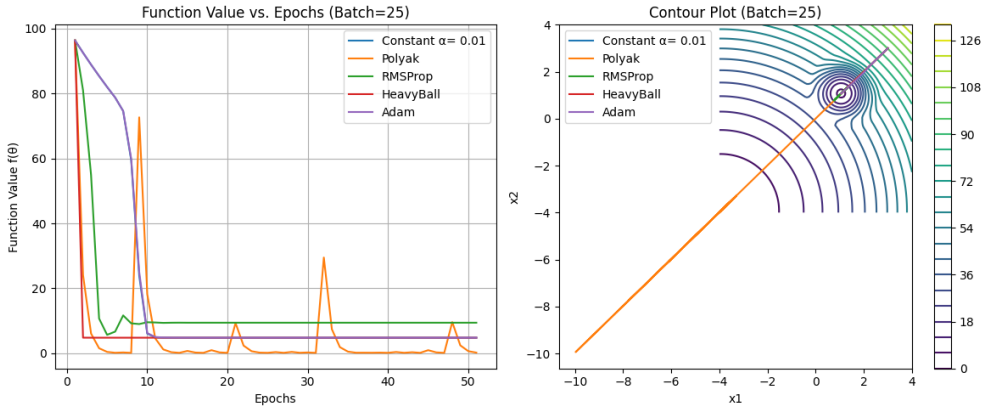


Figure 14: Comparison of algorithms - Batch Size = 25

Finally figure 14, using the largest batch size of 25 - the entire dataset, provides some interesting insights. First of all, polyak performs the same because it focuses on using F_star, which I have set to 0, so it still jumps towards the minimum away from the valley. This time, it has no noise associated because it is using the full dataset to calculate the exact derivative. However, for rmsprop, heavyball, and adam, each of them ended up stuck in the valley. We can see this from the function plot where they all flatline around 8, which is the smallest point in the valley. This is because we're using the entire dataset so there is no noise introduced when calculating the approximate derivative, meaning each algorithm can't find its way back out of the valley. The noise introduced in the other batches allowed each of the algorithms to jump out.

Code

mini_batch_sgd.py

```
from matplotlib import pyplot as plt
from loss_functions import *
from step_sizes import *
import sympy as sp

F_STAR = 0
x1, x2 = sp.symbols('x1 x2')
x0 = [3, 3]  # Initial point
X = generate_trainingdata(m=25)
"""
Part A
"""


def approx_gradient(f, x, minibatch):
    grad_x1_total = 0
    grad_x2_total = 0
    for w in minibatch:
        # Compute symbolic derivatives
        grad_x1, grad_x2 = compute_symbolic_derivative([x1, x2], [w])

        # Compute exact gradient at this point
        grad_x1_value, grad_x2_value = numerical_derivative(x[0], x[1], grad_x1,
            grad_x2)

        grad_x1_total += grad_x1_value
        grad_x2_total += grad_x2_value

    # Average the gradients over the mini-batch
    grad_x1_avg = grad_x1_total / len(minibatch)
    grad_x2_avg = grad_x2_total / len(minibatch)

    return np.array([grad_x1_avg, grad_x2_avg])


"""
(i)
"""


def mini_batch_SGD(X, initial_x0, num_iters, batch_size, step_size_func, alpha0,
    beta0, beta1, epsilon=1e-8):
    m, n = X.shape
    theta = np.array(initial_x0)

    moving_avg = np.zeros_like(theta)  # RMSPROP
    z_t = np.zeros_like(theta)  # Heavy Ball
    m_t = np.zeros_like(theta)  # Adam - First momentum term
    v_t = np.zeros_like(theta)  # Adam - Second momentum term
    t = 1  # Adam - time step
```

```
    theta_history = [theta.copy()]
    f_values = [f(theta, X)]

    epoch_count = 0

    for iter_num in range(num_iters):
        indices = np.random.permutation(m)
        X_shuffled = X[indices]

        epoch_count += 1
        for i in range(0, m, batch_size):
            minibatch = X_shuffled[i:i + batch_size]

            # Calculate the approximate gradient for the current mini-batch
            grad = approx_gradient(f, theta, minibatch)
            if step_size_func == polyak_step_size:
                f_x = f(theta, minibatch)
                alpha = step_size_func(f_x, F_STAR, grad)
            elif step_size_func == rmsprop_step:
                alpha, average = step_size_func(grad, moving_avg, alpha0, beta0)
                moving_avg = average
            elif step_size_func == heavy_ball_step:
                z = heavy_ball_step(z_t, grad, alpha0, beta0)
                alpha = z
            elif step_size_func == adam_step:
                m_t, v_t, alpha = step_size_func(m_t, v_t, grad, alpha0, beta0,
                    beta1, epsilon, t)
                t += 1
            else:
                alpha = alpha0

            theta = theta - (alpha * grad)
            theta_history.append(theta.copy())
            f_values.append(f(theta, X))

    return np.array(theta_history), np.array(f_values), epoch_count


"""
(iii)
"""


def symbolic_f(x, minibatch):
    y = 0
    count = 0
    for w in minibatch:
        z1 = x[0] - w[0] - 1  # x[0] - w[0] - 1
        z2 = x[1] - w[1] - 1  # x[1] - w[1] - 1
        term1 = 37 * (z1 ** 2 + z2 ** 2)
        term2 = (z1 + 5) ** 2 + (z2 + 5) ** 2
        y += sp.Min(term1, term2)
```

```
            count += 1
    return y / count


def compute_symbolic_derivative(x, minibatch):
    grad_x1 = sp.diff(symbolic_f(x, minibatch), x[0])
    grad_x2 = sp.diff(symbolic_f(x, minibatch), x[1])
    return grad_x1, grad_x2


def numerical_derivative(x1_val, x2_val, grad_x1, grad_x2):
    grad_x1_value = grad_x1.subs({x1: x1_val, x2: x2_val})
    grad_x2_value = grad_x2.subs({x1: x1_val, x2: x2_val})
    return grad_x1_value, grad_x2_value


"""
(ii)
"""


def a_part_two():
    training_data = generate_trainingdata(m=25)
    X = training_data

    x1_range = np.linspace(-1.5, 3, 100)
    x2_range = np.linspace(-1.5, 3, 100)
    x1, x2 = np.meshgrid(x1_range, x2_range)

    f_values = np.zeros_like(x1)

    for i in range(x1.shape[0]):
        for j in range(x1.shape[1]):
            x = np.array([x1[i, j], x2[i, j]])
            f_values[i, j] = f(x, X)

    fig = plt.figure(figsize=(12, 6))

    ax = fig.add_subplot(121, projection='3d')
    ax.plot_surface(x1, x2, f_values, cmap='viridis')
    ax.set_title('Wireframe Plot of f(x, T)')
    ax.set_xlabel('x1')
    ax.set_ylabel('x2')
    ax.set_zlabel('f(x, T)')

    ax2 = fig.add_subplot(122)
    contour = ax2.contour(x1, x2, f_values, 20, cmap='viridis')
    ax2.set_title('Contour Plot of f(x, T)')
    ax2.set_xlabel('x1')
    ax2.set_ylabel('x2')
    fig.colorbar(contour)

    plt.tight_layout()
```

```
    plt.savefig("images/wireframe_contour_all_N.png")


"""
Part B
"""

"""
(i)
"""


def gradient_descent(x0, minibatch, alpha, num_iterations):
    X = np.array([x0])
    symbolic_grad = compute_symbolic_derivative([x1, x2], minibatch)

    x1_value = X[-1][0]
    x2_value = X[-1][1]

    for i in range(num_iterations):
        grad_x1_value, grad_x2_value = numerical_derivative(x1_value, x2_value,
            symbolic_grad[0], symbolic_grad[1])
        step = alpha * np.array([grad_x1_value, grad_x2_value])
        x0 = X[-1] - step
        X = np.append(X, [x0], axis=0)

        x1_value = x0[0]
        x2_value = x0[1]

    return X


def part_b_one():
    alpha = 0.01
    num_iterations = 100

    # Run gradient descent
    X_history = gradient_descent(x0, X, alpha, num_iterations)
    x1_range = np.linspace(-4, 4, 100)
    x2_range = np.linspace(-4, 4, 100)
    x1_feature, x2_feature = np.meshgrid(x1_range, x2_range)

    f_values = np.zeros_like(x1_feature)

    for i in range(x1_feature.shape[0]):
        for j in range(x1_feature.shape[1]):
            x = np.array([x1_feature[i, j], x2_feature[i, j]])  # Current point (x1,
                x2)
            f_values[i, j] = f(x, X)  # Calculate the loss for this point

    # Create the contour plot
    plt.figure(figsize=(8, 6))
```

```
    # Plot the contour of the loss function
    contour = plt.contour(x1, x2, f_values, 20, cmap='viridis')
    plt.title(f'Contour Plot with Gradient Descent Path with alpha={alpha}')
    plt.xlabel('x1')
    plt.ylabel('x2')

    # Plot the path of gradient descent (as a line)
    plt.plot(X_history[:, 0], X_history[:, 1], 'ro-', label='Gradient Descent Path')
    plt.colorbar(contour)
    plt.legend()
    plt.tight_layout()
    plt.savefig(f"images/grad_descent_alpha_{alpha}.png")


"""
(ii)
"""


def part_b_two():
    alpha = 0.01
    num_iters = 100
    batch_size = 5

    # Run mini-batch SGD with constant step size
    theta_final, func_values, epochs = mini_batch_SGD(X, x0, num_iters, batch_size,
        None, alpha, None, None)

    x1_range = np.linspace(-4, 4, 100)
    x2_range = np.linspace(-4, 4, 100)
    x1_feature, x2_feature = np.meshgrid(x1_range, x2_range)

    f_values = np.zeros_like(x1_feature)

    for i in range(x1_feature.shape[0]):
        for j in range(x1_feature.shape[1]):
            x = np.array([x1_feature[i, j], x2_feature[i, j]])  # Current point (x1,
                x2)
            f_values[i, j] = f(x, X)  # Calculate the loss for this point

    # Create the contour plot
    plt.figure(figsize=(8, 6))

    # Plot the contour of the loss function
    contour = plt.contour(x1_feature, x2_feature, f_values, 20, cmap='viridis')
    plt.title(f'Contour Plot with Mini-Batch SGD with alpha={alpha}')
    plt.xlabel('x1')
    plt.ylabel('x2')

    theta_final = np.array(theta_final)
    # Plot the path of gradient descent (as a line)
    plt.plot(theta_final[:, 0], theta_final[:, 1], 'ro-', label='Mini-Batch SGD Path
        ')
```

```
    plt.xlim(-4, 4)
    plt.ylim(-4, 4)
    plt.colorbar(contour)
    plt.legend()
    plt.tight_layout()
    plt.savefig(f"images/mini_batch_sgd_alpha_{alpha}.png")


"""
Plot epochs vs function value
"""


def part_b_epochs():
    alpha = 0.01
    num_iters = 25
    batch_size = 5

    num_runs = 5  # Run SGD multiple times
    plt.figure(figsize=(8, 6))

    for run in range(num_runs):
        theta_final, f_values, num_epochs = mini_batch_SGD(X, x0, num_iters,
            batch_size, None, alpha, None, None)
        f_values_per_epoch = f_values[batch_size - 1::batch_size]  # Every
            batch_size-th value corresponds to an epoch

        # Plot function values at each epoch
        plt.plot(range(1, num_epochs + 1), f_values_per_epoch, label=f'Run {run + 1}
            ')

    plt.yscale('log')
    plt.xlabel('Epochs')
    plt.ylabel('Function Value f(θ)')
    plt.title('Function Value vs. Epochs for Mini-Batch SGD')
    plt.legend()
    plt.grid()
    plt.savefig(f"images/runs_epoch_vs_function_{alpha}_log.png")


"""
(iii)
"""


def part_b_three():
    alpha = 0.01
    num_iters = 100
    batch_sizes = [5, 15, 20, 25]

    x1_range = np.linspace(-4, 4, 100)
    x2_range = np.linspace(-4, 4, 100)
    x1_feature, x2_feature = np.meshgrid(x1_range, x2_range)
```

```
    f_values = np.zeros_like(x1_feature)

    for i in range(x1_feature.shape[0]):
        for j in range(x1_feature.shape[1]):
            x = np.array([x1_feature[i, j], x2_feature[i, j]])  # Current point (x1,
                x2)
            f_values[i, j] = f(x, X)  # Calculate the loss for this point

    for batch_size in batch_sizes:
        print(f"Running for batch size: {batch_size}")
        theta_final, f_vals, num_epochs = mini_batch_SGD(X, x0, num_iters,
            batch_size, None, alpha, None, None)

        fig, axes = plt.subplots(1, 2, figsize=(12, 5))

        ax1 = axes[1]
        contour = ax1.contour(x1_feature, x2_feature, f_values, 20, cmap='viridis')
        ax1.plot(theta_final[:, 0], theta_final[:, 1], 'ro-', label='Mini-Batch SGD
            Path')
        ax1.set_title(f'Contour Plot (Batch={batch_size})')
        ax1.set_xlabel('x1')
        ax1.set_ylabel('x2')
        ax1.legend()
        fig.colorbar(contour, ax=ax1)

        ax2 = axes[0]
        num_updates_per_epoch = np.ceil(len(X) / batch_size).astype(int)  # How many
             updates per epoch?
        epoch_indices = np.arange(num_updates_per_epoch - 1, len(f_vals),
            num_updates_per_epoch)
        f_values_per_epoch = f_vals[epoch_indices]
        ax2.plot(range(1, len(f_values_per_epoch) + 1), f_values_per_epoch, label=f'
            Batch={batch_size}')

        ax2.set_xlabel('Epochs')
        ax2.set_ylabel('Function Value f(θ)')
        ax2.set_title(f'Function Value vs. Epochs (Batch={batch_size})')
        ax2.legend()
        ax2.grid()

        plt.tight_layout()
        plt.savefig(f"images/batch_sizes/sgd_subplots_alpha_{alpha}_batch_{
            batch_size}.png")
        plt.clf()


"""
(iv)
"""


def part_b_four():
```

```
alpha_values = [0.0001, 0.01, 0.1, 0.5]
num_iters = 100
batch_size = 5

x1_range = np.linspace(-4, 4, 100)
x2_range = np.linspace(-4, 4, 100)
x1_feature, x2_feature = np.meshgrid(x1_range, x2_range)

f_values = np.zeros_like(x1_feature)

for i in range(x1_feature.shape[0]):
    for j in range(x1_feature.shape[1]):
        x = np.array([x1_feature[i, j], x2_feature[i, j]])  # Current point (x1,
            x2)
        f_values[i, j] = f(x, X)  # Calculate the loss for this point

"""Subplots"""
for alpha in alpha_values:
    print(f"Running for alpha: {alpha}")
    theta_final, f_vals, num_epochs = mini_batch_SGD(X, x0, num_iters,
        batch_size, None, alpha, None, None)

    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    ax1 = axes[1]
    contour = ax1.contour(x1_feature, x2_feature, f_values, 20, cmap='viridis')
    ax1.plot(theta_final[:, 0], theta_final[:, 1], 'ro-', label='Mini-Batch SGD
        Path')
    ax1.set_title(f'Contour Plot (Alpha={alpha})')
    ax1.set_xlabel('x1')
    ax1.set_ylabel('x2')
    ax1.legend()
    fig.colorbar(contour, ax=ax1)

    ax2 = axes[0]
    num_updates_per_epoch = np.ceil(len(X) / batch_size).astype(int)
    epoch_indices = np.arange(num_updates_per_epoch - 1, len(f_vals),
        num_updates_per_epoch)
    f_values_per_epoch = f_vals[epoch_indices]
    ax2.plot(range(1, len(f_values_per_epoch) + 1), f_values_per_epoch, label=f'
        Alpha={alpha}')
    ax2.set_xlabel('Epochs')
    ax2.set_ylabel('Function Value f(θ)')
    ax2.set_title(f'Function Value vs. Epochs (Alpha={alpha})')
    ax2.legend()
    ax2.grid()

    # Adjust layout and save the figure
    plt.tight_layout()
    plt.savefig(f"images/alpha/sgd_subplots_alpha_{alpha}.png")
    plt.clf()
```

```
"""
Part C
"""


def polyak_sgd_contour():
    alpha_values = [0.0001, 0.01, 0.1, 0.5]
    num_iters = 50
    batch_size = 5

    x1_range = np.linspace(-4, 4, 100)
    x2_range = np.linspace(-4, 4, 100)
    x1_feature, x2_feature = np.meshgrid(x1_range, x2_range)

    f_values = np.zeros_like(x1_feature)

    for i in range(x1_feature.shape[0]):
        for j in range(x1_feature.shape[1]):
            x = np.array([x1_feature[i, j], x2_feature[i, j]])  # Current point (x1,
                x2)
            f_values[i, j] = f(x, X)  # Calculate the loss for this point

    # Create the contour plot
    plt.figure(figsize=(8, 6))

    # Plot the contour of the loss function
    contour = plt.contour(x1_feature, x2_feature, f_values, 20, cmap='viridis')
    plt.title(f'Contour Plot with Mini-Batch SGD with Polyak')
    plt.xlabel('x1')
    plt.ylabel('x2')
    for alpha in alpha_values:
        print(f"Running for alpha: {alpha}")
        theta_final, f_vals, num_epochs = mini_batch_SGD(X, x0, num_iters,
            batch_size, None, alpha, None, None)

        plt.plot(theta_final[:, 0], theta_final[:, 1], label=f'Alpha = {alpha}')

    plt.colorbar(contour)
    plt.legend()
    plt.tight_layout()
    plt.savefig(f"images/polyak/contour.png")


def rms_sgd_contour():
    alpha_values = [0.0001, 0.01, 0.1, 0.5]
    num_iters = 50
    batch_size = 5

    x1_range = np.linspace(-4, 4, 100)
    x2_range = np.linspace(-4, 4, 100)
    x1_feature, x2_feature = np.meshgrid(x1_range, x2_range)

    f_values = np.zeros_like(x1_feature)
```

```
    for i in range(x1_feature.shape[0]):
        for j in range(x1_feature.shape[1]):
            x = np.array([x1_feature[i, j], x2_feature[i, j]])  # Current point (x1,
                x2)
            f_values[i, j] = f(x, X)  # Calculate the loss for this point

    # Create the contour plot
    plt.figure(figsize=(8, 6))

    # Plot the contour of the loss function
    contour = plt.contour(x1_feature, x2_feature, f_values, 20, cmap='viridis')
    plt.title(f'Contour Plot with Mini-Batch SGD with RMSProp')
    plt.xlabel('x1')
    plt.ylabel('x2')

    beta_values = [0.25, 0.9]

    for beta in beta_values:
        for alpha in alpha_values:
            theta_final, f_values, num_epochs = mini_batch_SGD(X, x0, num_iters,
                batch_size, polyak_step_size, alpha,
                                                               beta, None)
            plt.plot(theta_final[:, 0], theta_final[:, 1], label=f'Alpha = {alpha},
                Beta = {beta}')

    plt.colorbar(contour)
    plt.legend()
    plt.tight_layout()
    plt.savefig(f"images/rmsprop/contour.png")


def rms_sgd():
    alpha_values = [0.0001, 0.01, 0.1, 0.5]
    beta_values = [0.25, 0.9]
    num_iters = 50
    batch_size = 5

    for beta in beta_values:
        plt.clf()
        for alpha in alpha_values:
            theta_final, f_values, num_epochs = mini_batch_SGD(X, x0, num_iters,
                batch_size, rmsprop_step, alpha,
                                                               beta, None)
            f_values_per_epoch = f_values[
                                 batch_size - 1::batch_size]  # Every batch_size-th
                                     value corresponds to an epoch

            # Plot function values at each epoch
            plt.plot(range(1, num_epochs + 1), f_values_per_epoch, label=f'Alpha: {
                alpha}')

        plt.xlabel('Epochs')
```

```
        plt.ylabel('Function Value f(θ)')
        plt.title(f'RMSProp: Function Value vs. Epochs - Beta = {beta} for varying
            Alpha values')
        plt.legend()
        plt.grid()
        plt.savefig(f"images/rmsprop/beta_{beta}_no_log.png")


def heavyball_sgd():
    alpha_values = [0.0001, 0.01, 0.1, 0.5]
    beta_values = [0.25, 0.9]
    num_iters = 50
    batch_size = 5

    for beta in beta_values:
        plt.clf()
        for alpha in alpha_values:
            theta_final, f_values, num_epochs = mini_batch_SGD(X, x0, num_iters,
                batch_size, heavy_ball_step, alpha,
                                                               beta, None)
            f_values_per_epoch = f_values[
                                 batch_size - 1::batch_size]  # Every batch_size-th
                                     value corresponds to an epoch

            safe_values = np.where(f_values_per_epoch > 0, f_values_per_epoch, np.
                nan)
            plt.plot(range(1, num_epochs + 1), safe_values, label=f'Alpha: {alpha}')

        plt.yscale('log')
        plt.xlabel('Epochs')
        plt.ylabel('Function Value f(θ)')
        plt.title(f'HeavyBall: Function Value vs. Epochs - Beta = {beta} for varying
             Alpha values')
        plt.legend()
        plt.grid()
        plt.savefig(f"images/heavyball/beta_{beta}.png")


def adam_sgd():
    alpha_values = [0.0001, 0.01, 0.1, 0.5]
    beta_values = [0.25, 0.9]
    beta1_values = [0.25, 0.99]
    num_iters = 50
    batch_size = 5

    for alpha in alpha_values:
        plt.clf()
        fig, axes = plt.subplots(2, 2, figsize=(10, 8))
        fig.suptitle(f'Adam: Function Value vs. Epochs for Different Beta Values',
            fontsize=14)
        for i, beta0 in enumerate(beta_values):
            for j, beta1 in enumerate(beta1_values):
                ax = axes[i, j]
```

```
                theta_final, f_values, num_epochs = mini_batch_SGD(X, x0, num_iters,
                    batch_size, adam_step,
                                                                   alpha,
                                                                   beta0, beta1)
                f_values_per_epoch = f_values[batch_size - 1::batch_size]  # Sample
                    every batch_size-th step

                ax.plot(range(1, num_epochs + 1), f_values_per_epoch, label=f'Alpha:
                    {alpha}')
                ax.set_yscale('log')
                ax.set_xlabel('Epochs')
                ax.set_ylabel('Function Value f(θ)')
                ax.set_title(f'Beta0 = {beta0}, Beta1 = {beta1}')
                ax.grid()

        plt.tight_layout()  # Adjust layout to fit title
        plt.savefig(f"images/adam_beta_comparison_{alpha}.png")


def adam_sgd_alpha():
    alpha_values = [0.0001, 0.01, 0.1, 0.5]
    beta_values = [0.25, 0.9]
    beta1_values = [0.25, 0.99]
    num_iters = 50
    batch_size = 5

    fig, axes = plt.subplots(2, 2, figsize=(10, 8))
    fig.suptitle('Adam: Function Value vs. Epochs for Different Alpha and Beta
        Values', fontsize=14)

    for idx, alpha in enumerate(alpha_values):
        ax = axes[idx // 2, idx % 2]

        for beta0 in beta_values:
            for beta1 in beta1_values:
                theta_final, f_values, num_epochs = mini_batch_SGD(X, x0, num_iters,
                    batch_size, adam_step,
                                                                   alpha, beta0,
                                                                      beta1)
                f_values_per_epoch = f_values[batch_size - 1::batch_size]  # Sample
                    every batch_size-th step
                ax.plot(range(1, num_epochs + 1), f_values_per_epoch, label=f'Beta0
                    ={beta0}, Beta1={beta1}')

        ax.set_yscale('log')
        ax.set_xlabel('Epochs')
        ax.set_ylabel('Function Value f(θ)')
        ax.set_title(f'Alpha = {alpha}')
        ax.grid()
        ax.legend()

    plt.tight_layout()
```

```
    plt.subplots_adjust(top=0.92)
    plt.savefig(f"images/adam/adam_alpha_comparison.png")
    plt.show()


"""
Part C
"""


def run_comparison():
    num_iters = 50
    batch_sizes = [5, 15, 25]
    alpha = 0.01

    x1_range = np.linspace(-4, 4, 100)
    x2_range = np.linspace(-4, 4, 100)
    x1_feature, x2_feature = np.meshgrid(x1_range, x2_range)

    f_values = np.zeros_like(x1_feature)

    for i in range(x1_feature.shape[0]):
        for j in range(x1_feature.shape[1]):
            x = np.array([x1_feature[i, j], x2_feature[i, j]])  # Current point (x1,
                x2)
            f_values[i, j] = f(x, X)  # Calculate the loss for this point

    for batch_size in batch_sizes:
        print(f"Running for batch size: {batch_size}")
        # Baseline
        theta_final, f_vals, num_epochs = mini_batch_SGD(X, x0, num_iters,
            batch_size, None, alpha, None, None)
        # Polyak
        polyak_final, polyak_vals, _ = mini_batch_SGD(X, x0, num_iters, batch_size,
            polyak_step_size, alpha, None, None)
        # RMSProp
        rmsprop_final, rmsprop_vals, _ = mini_batch_SGD(X, x0, num_iters, batch_size
            , rmsprop_step, 0.5, 0.25, None)
        # HeavyBall
        heavyball_final, heavyball_vals, _ = mini_batch_SGD(X, x0, num_iters,
            batch_size, heavy_ball_step, 0.01, 0.9,
                                                            None)
        # Adam
        adam_final, adam_vals, _ = mini_batch_SGD(X, x0, num_iters, batch_size,
            adam_step, 0.01, 0.25,
                                                  0.25)

        # Create a new figure with 1 row, 2 columns
        fig, axes = plt.subplots(1, 2, figsize=(12, 5))

        # Contour Plot + SGD Path (Left subplot)
        ax1 = axes[1]
        contour = ax1.contour(x1_feature, x2_feature, f_values, 20, cmap='viridis')
```

```
ax1.plot(theta_final[:, 0], theta_final[:, 1], label='Constant α= 0.01')
ax1.plot(polyak_final[:, 0], polyak_final[:, 1], label='Polyak')
ax1.plot(rmsprop_final[:, 0], rmsprop_final[:, 1], label='RMSProp')
ax1.plot(heavyball_final[:, 0], heavyball_final[:, 1], label='HeavyBall')
ax1.plot(adam_final[:, 0], adam_final[:, 1], label='Adam')

ax1.set_title(f'Contour Plot (Batch={batch_size})')
ax1.set_xlabel('x1')
ax1.set_ylabel('x2')
ax1.legend()
fig.colorbar(contour, ax=ax1)

ax2 = axes[0]
num_updates_per_epoch = np.ceil(len(X) / batch_size).astype(int)
epoch_indices = np.arange(num_updates_per_epoch - 1, len(f_vals),
    num_updates_per_epoch)
f_values_per_epoch = f_vals[epoch_indices]
polyak_values_per_epoch = polyak_vals[epoch_indices]
rmsprop_values_per_epoch = rmsprop_vals[epoch_indices]
heavyball_values_per_epoch = heavyball_vals[epoch_indices]
adam_values_per_epoch = adam_vals[epoch_indices]
ax2.plot(range(1, len(f_values_per_epoch) + 1), f_values_per_epoch, label=f'
    Constant α= 0.01')
ax2.plot(range(1, len(f_values_per_epoch) + 1), polyak_values_per_epoch,
    label=f'Polyak')
ax2.plot(range(1, len(f_values_per_epoch) + 1), rmsprop_values_per_epoch,
    label=f'RMSProp')
ax2.plot(range(1, len(f_values_per_epoch) + 1), heavyball_values_per_epoch,
    label=f'HeavyBall')
ax2.plot(range(1, len(f_values_per_epoch) + 1), adam_values_per_epoch, label
    =f'Adam')

ax2.set_xlabel('Epochs')
ax2.set_ylabel('Function Value f(θ)')
ax2.set_title(f'Function Value vs. Epochs (Batch={batch_size})')
ax2.legend()
ax2.grid()

plt.tight_layout()
plt.savefig(f"images/partC/sgd_comparison_batch_{batch_size}.png")
plt.clf()
```