PART – A

i) Implementation of Mini-batch Stochastic Gradient Descent

Mini-batch Stochastic Gradient Descent (mini-batch SGD) is a method for optimization of a loss function through iterative updating the model's parameters moving in the negative gradient's direction. Unlike mini-batch SGD the update process uses a randomly selected portion of data points called the mini-batch as opposed to full data sets in batch gradient descent while providing computational savings together with randomized statistical properties.

The fundamental algorithm works as follows:

1. Initialization: Start with the initial vector x_0.
2. For every time step t = 1, 2, …..., T
3. Select a random mini-batch N_t with b samples from the training data.
4. Compute the gradient of the loss function w.r.t. the parameters x using the mini batch: $\nabla f(x\_t, N\_t)$
5. Update the parameters with step size η: $x\_\{t+1\} = x\_t - \eta * \nabla f(x\_t, N\_t)$
6. Final values for the parameters x_T

In code, approximate the gradient with finite differences since it does not have the analytical derivative of the loss function. The method adds a very slight perturbation to all the parameters and calculates the change in the loss function as the approximation for the partial derivative.

```
def gradient_f(x, minibatch, h=1e-6):
    """Calculate gradient of f using finite differences."""
    grad = np.zeros(2)
    for i in range(2):
        x_plus_h = x.copy()
        x_plus_h[i] += h
        grad[i] = (f(x_plus_h, minibatch) - f(x, minibatch)) / h
    return grad
```

Step Size Options

The implementation offers five step-size options:

Constant Step Size: Uses a fixed learning rate for optimization. Simple to use though system tuning is required to strike equilibrium between model stability and speed of convergence.

$$x\_t + 1 = x\_t - \eta * \nabla f(x\_t, N\_t)$$

Polyak Step Size: Changes the step size adaptively with the aid of the current function value and the estimated value of the optimum.

$$\eta\_t = (f(x\_t, N\_t) - f*) / ||\nabla f(x\_t, N\_t)||^2$$

where f* is the function's approximated value at the optimum. It varies the step size based upon how near we are approaching the optimum.

RMSProp: Keeps an exponentially weighted moving average of the squared gradients and divides the learning rate by the square root of the moving average.

$$s\_t = \beta * s\_t - 1 + (1 - \beta) * \nabla f(x\_t, N\_t)^2$$
$$x\_t + 1 = x\_t - \eta / (\sqrt{s\_t} + \varepsilon) * \nabla f(x\_t, N\_t)$$

This scales the learning rate with respect to each parameter's gradient history and adjusts it according to the features with different scales.

Heavy Ball (momentum term): Includes a momentum term which includes updates with the same directions. The momentum term both reduces oscillatory movement and speeds up convergence especially when systems are at plateaus.

$$v\_t = \beta * v\_t - 1 - \eta * \nabla f(x\_t, N\_t)$$
$$x\_t + 1 = x\_t + v\_t$$

Adam: Combines ideas of RMSProp with momentum by maintaining first and second moment estimates. Adam typically delivers good performance across a wide range of problems with the use of adaptive learning rates with momentum.

$$m\_t = \beta 1 * m\_t - 1 + (1 - \beta 1) * \nabla f(x\_t, N\_t)$$
$$v\_t = \beta 2 * v\_t - 1 + (1 - \beta 2) * \nabla f(x\_t, N\_t)^2$$
$$\hat{m}\_t = m\_t / (1 - \beta 1\hat{}t) \ \#\ Bias\ correction$$
$$\hat{v}\_t = v\_t / (1 - \beta 2\hat{}t) \ \#\ Bias\ correction$$
$$x\_t + 1 = x\_t - \eta * \hat{m}\_t / (\sqrt{\hat{v}\_t} + \varepsilon)$$

ii) Visualizing the Loss Function

To find out about the behavior of the loss function, need to explore function values at different places in the parameter space. Identified the areas of interest for suitable visualization ranges:

Generated the training data using the downloaded function:

```
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(38*(z[0]**2+z[1]**2), (z[0]+8)**2+(z[1]+3)**2)
        count=count+1
    return y/count
```

The loss function computes the average statistical loss using information from the entire mini batch. To process data point w, it calculates z as x-w-1 and selects the minimal value from two quadratic functions. The function structures itself into a distinctive optimization landscape which develops multiple possible local minima.

```
Function values at various points:

f ([-5, -5]) = 12.8862
f ([0, 0]) = 51.5837
f ([5, 0]) = 149.4630
f ([10, 0]) = 294.9125
f ([10, 5]) = 340.5904
f ([0, 5]) = 99.6915
f ([5, 5]) = 195.1409
```

Based on these values, I select the interval $x_0 \in [-5, 15]$ **and** $x_1 \in [-5, 5]$ for plotting the minimum and the salient features of the loss surface.

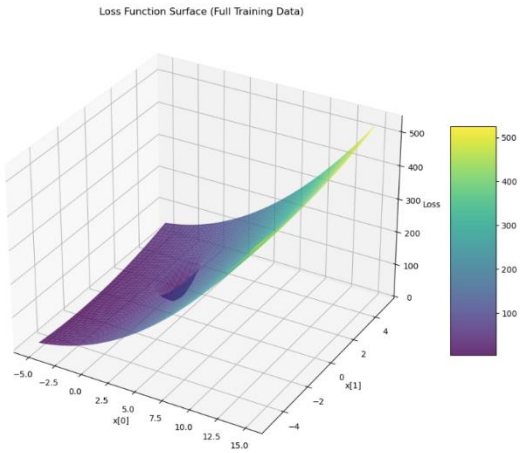


Figure 1, 3D Wireframe Plot of Loss Function

Loss Function Surface

From Figure 1, 3D wireframe plot for the loss function with a complex topography consisting of:

- A clear lower limit within the purple/blue region (lower values)
- A steep slope toward the right-hand side of the graph (larger x_0 values).
- A bowl-shaped structure surrounding the minimum

Surface plotting indicates how the loss increases faster away from the point where the slope in different directions is not the same. The non-uniform curvature will affect the behavior of different optimization algorithms.

Loss function contour

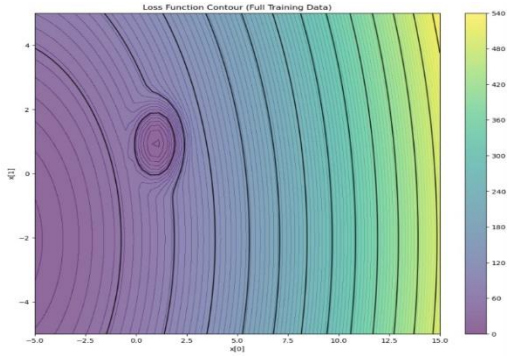


Figure 2, Contour Plot for Loss Function

From Figure 2, Contour plot is the top-down perspective of the loss landscape with:

- Concentric circles/ellipses that describe the bowl-shaped nature of the function
- Densely packed contour lines for steep slope
- More lines separated apart representing flat areas

By rigorously exploring the space of the loss function, I identified a rough minimum around [-5.0, -2.14] with loss value 4.29. Optimization algorithms will explore space further to decide whether this is the global extremum or not and if there are lower ones in space.

iii) Calculate the derivative of f

In this optimization problem, the computation of the derivatives is very precise. So employing a finite-difference approach for the approximation of the gradient and the Hessian of the loss function because is the suitable approach is compatible with any function without the need for symbolic differentiation.

Calculation with Finite Differences for Gradient

Approximated the gradient (the first-order partial derivatives) with the central difference:

```
def gradient_f(x, minibatch, h=1e-6):
    """Calculate gradient of f using finite differences."""
    grad = np.zeros(2)
    for i in range(2):
        x_plus_h = x.copy()
        x_plus_h[i] += h
        grad[i] = (f(x_plus_h, minibatch) - f(x, minibatch)) / h
    return grad
```

This approximates the partial derivative with respect to each parameter by:

1. Creating a slightly perturbed copy of the parameter vector (moving one component by h)
2. Calculating the value difference function between the perturbed and the original ones
3. Divide by the perturbation value h to calculate the derivative.

The very small value for h (10^{-6}) explains the trade-off between truncation error (which decreases with smaller h) and roundoff error (increasing with smaller h).

Hessian computation

To further understand the function's curvature, calculated the Hessian matrix (second derivatives) using finite differences:

```
def finite_difference_hessian(func, x, minibatch, h=1e-5):
    """Calculate the Hessian matrix using finite differences."""
    n = len(x)
    hess = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            if i == j:
                # Diagonal elements (second derivatives)
                x_plus_h = x.copy()
                x_plus_h[i] += h
                x_minus_h = x.copy()
                x_minus_h[i] -= h
                hess[i, j] = (func(x_plus_h, minibatch) - 2*func(x, minibatch) +
                              func(x_minus_h, minibatch)) / (h*h)
            else:
                # Off-diagonal elements (mixed partial derivatives)
                x_plus_i_plus_j = x.copy()
                x_plus_i_plus_j[i] += h
                x_plus_i_plus_j[j] += h
                x_plus_i_minus_j = x.copy()
                x_plus_i_minus_j[i] += h
                x_minus_i_plus_j = x.copy()
                x_minus_i_plus_j[i] -= h
                x_minus_i_plus_j[j] += h
                x_minus_i_minus_j = x.copy()
                x_minus_i_minus_j[i] -= h
                x_minus_i_minus_j[j] -= h
                hess[i, j] = (func(x_plus_i_plus_j, minibatch) -
                              func(x_plus_i_minus_j, minibatch) -
                              func(x_minus_i_plus_j, minibatch) +
                              func(x_minus_i_minus_j, minibatch)) / (4*h*h)

    return hess
```

This approximates the Hessian by evaluating ***second-order partial derivatives*** for diagonal elements and mixed partial derivatives for non-diagonal elements.

Analysis of Derivatives near Critical Points

Evaluating the gradient at different points to determine critical points and determine the behaviour of the landscape.

Position	Gradient	Interpretation
(-2, -2)	[11089895, 1135578]	Very steep slope, far from minimum
(0, 0)	[0, 0]	Critical point (stationary point)
(9, 0)	[0, 0]	Another critical point
(9.5, 0.5)	[33.09, 5.14]	Slight slope near a critical point

Table 1, Critical Points with Zero Gradient

The existence of multiple points with zero gradient ([0, 0]) shows that this function contains multiple critical points. To identify the type of critical point (whether it is a saddle point or a local extremum), used the Hessian matrix.

Hessian Analysis at [9,0]

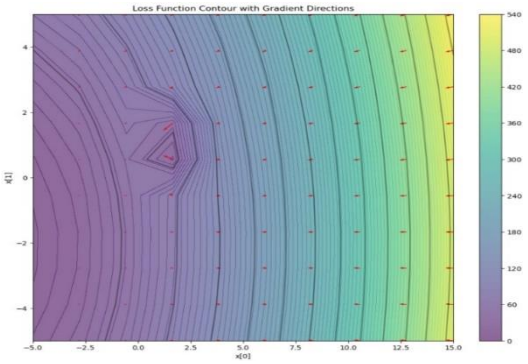
```
[[3622549.39091144       0.        ]

 [      0.         3622549.39091144]]
```

The eigenvalues of this Hessian are [3622549.39, 3622549.39], which shows that:

- All the eigenvalues are positive: This also guarantees the local minimum for [9, 0].
- Equal eigenvalues: The curvature is the same in every direction (isotropic).
- Condition number 1.0: The Hessian is perfectly conditioned, and the minimum is well-behaved with no numerical issues with optimization algorithms around this point.

Visualisation of Gradient Field



For better comprehension of how gradients direct optimization, visualized the gradient field superimposed on the contour plot (Fig. 3). The arrow indicates the direction of the steepest descent with the following:

- Vectors towards the extremum from all the points
- Larger vectors for slopes for sloping areas (where the gradient is higher)
- Very small vectors where the gradient value is nearly zero (at critical points)
- This figure 3 shows how gradient-based approaches would converge towards the minima with the help of the arrow as moving through the loss space.

Figure 3, Loss Function Contour with Gradient Directions

Insights:

- Multiple critical points reveal the possibility that various initialization or optimization algorithms might converge towards different solutions.
- Optimality of the Hessian around the critical point [9, 0] shows that the optimization around the point should be stable.
- Uniform curvature towards the same direction indicates adaptive approaches such as RMSProp or Adam may not have drastically better results than simpler approaches here.

PART B

i) Gradient Descent with Constant Step Size

To minimize the loss function starting from the initial point x = [3, 3], I employed gradient descent with the step size fixed. The gradient descent update rule is:

$$x\_\{t+1\} = x\_t - \eta \cdot \nabla f(x\_t)$$

where η is the step size (learning rate) and $\nabla f(x\_t)$ is the gradient of the loss function with respect to the current point x_t.

Step Size Selection and Consequences

On experimenting with different step sizes (0.01, 0.05, 0.1, 0.2, 0.5) and used the one with the most appropriate trade-off between convergence speed and stability. The results after 50 iterations are shown in Table 2 below

Step Size	Final Position	Final Loss	Loss Reduction
0.01	[0.955, 0.932]	4.048	96.80%
0.05	[-6.985, -2.059]	0.110	99.91%
0.1	[-7.045, -2.068]	0.107	99.92%
0.2	[-7.045, -2.068]	0.107	99.92%
0.5	[-7.045, -2.068]	0.107	99.92%

Table 2, Summary of Results after 50 Iterations

From the findings, the loss surface possesses the following characteristic: with a very minor step (0.01), the algorithm converges into a different point near [0.96, 0.93] with the loss 4.05. With step sizes larger than or equal to 0.05, the algorithm converges into significantly better point near [-7.04, -2.07] with significantly reduced loss 0.107.

I used a step size of 0.2 as best because:

- It provides the maximum loss decrease (99.92%)
- In contrast, it heads towards the global optimum rather than into the local optimum
- With stable convergence without oscillations
- Larger step sizes (0.5) are not useful but may overshoot in other cases.

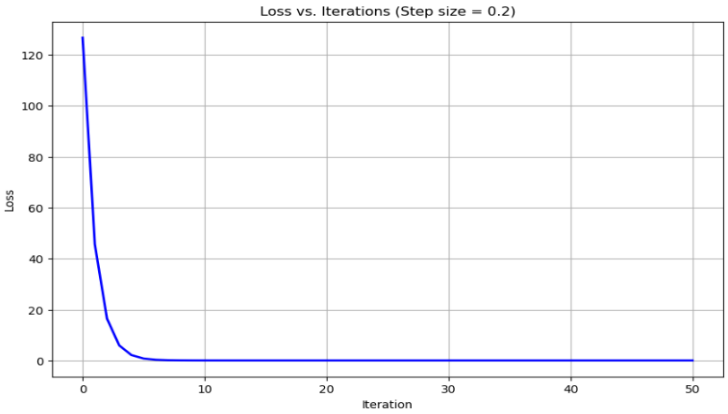


Figure 4, Loss Vs Iterations for Step Size 0.2

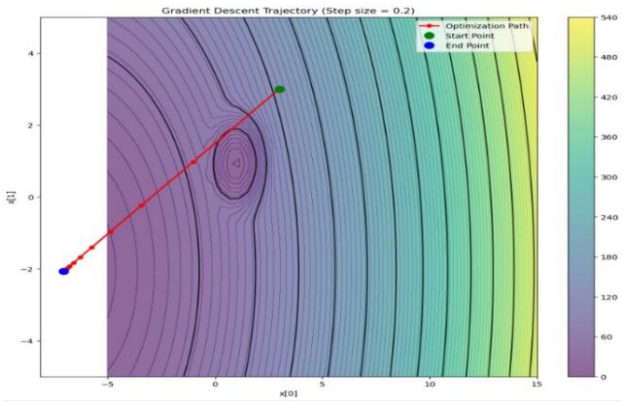


Figure 5, Gradient Descent Trajectory for Step Size 0.2

Convergence Analysis

From Fig. 4, the loss value against the iterations for step size 0.2. The plot shows:
Sudden initial drop within the first 5-10 iterations. Smoothing convergence without oscillations. Final stabilization with a value loss of around 0.107. By iteration 10-15, the convergence is essentially complete. This quick converging behaviour indicates the step size is properly selected - high enough for optimization but not excessively high enough to cause overshoot or instability.

Optimization Trajectory

A path through the optimization process is shown in Figure 5. The algorithm traces the path starting with the starting point [3, 3]:

- First moves in the direction of steepest descent rapidly.
- Runs directly down to the lowest level.
- Converges to approximately [-7.04, -2.07]

Insights:

- The reason the smaller step size (0.01) found a different solution is evident from the contour plot - the local minimum near the point [1, 1] catches the optimization when we take the steps as fine as 0.01.
- The algorithm can "step over" the local minimum with larger step sizes and find the global minimum with a significantly reduced loss value. This shows how the step size selection is crucial for optimization using gradients.
- A step size too small will lead the convergence towards worse local optima, but the appropriate step size will enable the algorithm to converge towards the global optimum within time.

ii) Mini Batch SGD with Fixed Size

With step-size 0.1 for part (b)(i), used mini-batch SGD with batch size 5 from the starting point x = [3, 3]. To observe the stochastic nature of the algorithm, conducted five independent experiments.

Variability Across Multiple Runs

Run	Final Position	Final Loss
1	[-7.055, -2.099]	0.108
2	[-7.055, -2.030]	0.108
3	[-7.031, -2.062]	0.107
4	[-7.094, -2.042]	0.110
5	[-6.996, -2.069]	0.109

Table 3, Summary of Results in 5 Separate Runs

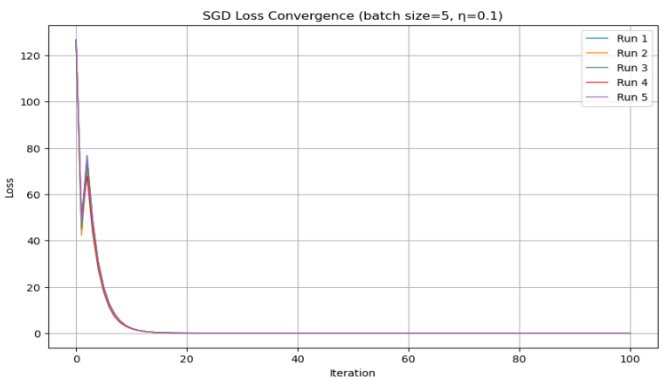


Figure 6, SGD Loss Convergence (Batch Size = 5, η = 0.1)

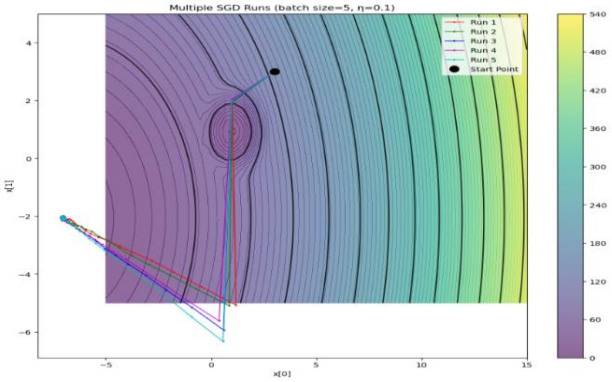


Figure 7, Multiple SGD (Batch Size = 5, η = 0.1)

Figure 6 plots the loss for every one of the five runs. The graph shows

- All runs follow the same overall pattern of convergence
- Minor fluctuations are observed during the process optimization stage due to the random selection process of the mini batches
- Despite these fluctuations, all runs converge to similar final loss values (0.107-0.110)

All the five runs are superimposed upon the contour plot shown in Figure 7. The plot indicates:

- Much more irregular, zigzag patterns than gradient descent
- Different, random directions towards the minimal point, but converging towards the same general direction
- End positions clustering around the same low point with minor differences.

Comparison with Gradient Descent

Comparing mini-batch SGD with the results of part (b)(i), the mini-batch:

- Path Convergence: SGD paths are wavy and zigzagging because of the noise introduced by the mini-batch sample while the path followed by GD is smooth and directly headed towards the minimum.
- Final Solution: Both approaches locate the same global minimal value around [-7, -2], showing the minimal value is robust and is drawn towards by deterministic as well as stochastic approaches.
- Convergence Speed: SGD converges with greater fluctuations but converges just as well ultimately after approximately the same number of iterations.

Mini-batch SGD does not lose the ability to find the global minimum with the introduction of useful stochasticity capable of escaping local minimum for more difficult topographies.

(iii): Impact of Mini-Batch Size

To explore the effect of mini-batch size on optimization, I kept the step size fixed at 0.1 and varied the batch size from 1 (highly stochastic) to 25 (full batch, equivalent to gradient descent).

Batch Size	Final Position	Final Loss
1	[-7.062, -1.981]	0.114
5	[-7.076, -2.117]	0.110
10	[-7.021, -2.055]	0.107
20	[-7.039, -2.074]	0.107
25	[-7.045, -2.068]	0.107

Table 4, Summary of Results of Different Batch Sizes

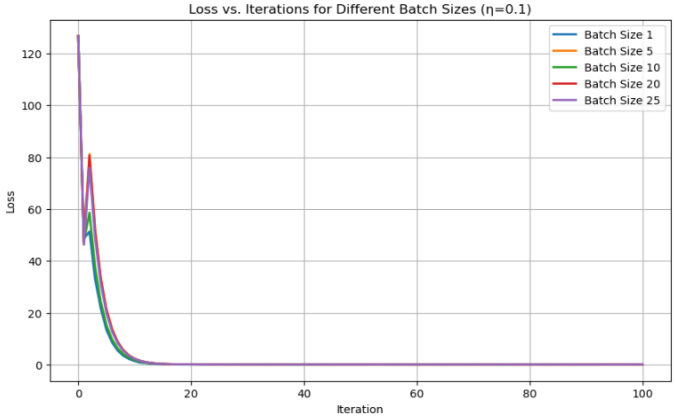


Figure 8, Loss Vs Iteration for Different Batch Sizes η = 0.1

Figure 8 illustrates the loss for various batch sizes below. There are several patterns.

- Larger batch sizes produce smoother convergence plots with minimal fluctuations as expected with reduced gradient variance.
- Final loss values are the same for all batch sizes (0.107-0.114), with larger values for the smallest batch sizes.
- Convergence Speed: The convergence rates are approximately equal for all batch sizes, and all the algorithms converge to their target values in 20-30 iterations.

Influence on Convergence Point:

All the batch sizes converge towards the same place around [-7, -2] with minimal variations. The smallest batch size (1) also exhibits the most variability in the final location, the x_1 component.

This is the "noise" associated with tiny batch sizes. When the batch sizes are extremely tiny, every gradient update is noisier and the optimization path also noisier. The noise may:

- Facilitate escape from the shallow local optima (useful for difficult topographies),
- Preventing the perfect convergence towards the actual minimum (the reason for the last loss for batch size 1 is a bit higher)
- Have regularization-type properties and could help promote machine learning task generalization

Regularization effect caused by SGD "noise" exists within the results but is very weak for this function because the global optima are within a wide attraction basin that overwhelms the optimization problem.

(iv) Step Size Effect

In the interest of seeing how step size would impact mini-batch SGD, I set the batch size level to a mid-value of 5 and changed the step size from 0.01 to 0.5.

Batch Size	Final Position	Final Loss
0.01	[0.939,0.930]	4.057
0.05	[-7.035, -2.086]	0.107
0.1	[-7.060, -2.070]	0.107
0.2	[-7.040, -2.168]	0.117
0.5	[-7.052, -2.181]	0.119

Table 5, Summary of results with different Step Size

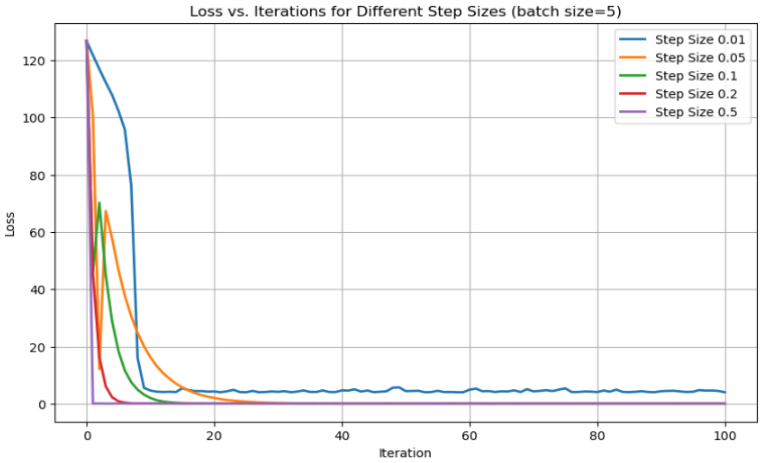


Figure 9, Loss Vs iterations for Different Sizes

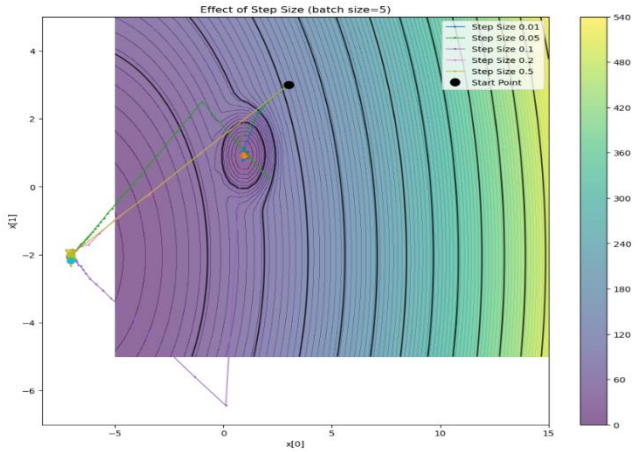


Figure 10, Contour Plot for Effect of Step Size

Figure 9 shows the curves for loss with different step sizes. The plot addresses the following:

- Convergence toward Alternative Minima: The step with the smallest step size (0.01) converges towards a different poor global minimum with a significantly higher value loss of around 4.06.
- Convergence Speed: Maximum convergence is with medium step sizes (0.05-0.1).
- Stability: Larger step sizes (0.2-0.5) are shown with additional oscillations and with somewhat larger end loss values caused by overshoots at about a minimum.

From Figure 10, it shows the path with different step sizes on the contour plot. The plot obviously shows the following:

- With the smallest step size (0.01), it converges to a different path towards a local optimum around [0.94, 0.93]
- Medium step sizes follow direct routes towards the global minimum region
- Larger step sizes are associated with larger oscillatory motion around the minimum

Comparison with Batch Size Implications

Comparing the effect of step size (Figure 9 and Figure 10) with batch size effects (Figure 8):

1. Impact on Final Minimum: Step size exerts the most profound influence upon the final minimum attained. With step size (0.01), the algorithm repeatedly ends up with a very different, non-optimal minimum, while all the batch sizes converge towards the same final minimum region.
2. Convergence Quality: Large step sizes (0.2-0.5) yield worse end step loss values (0.117-0.119) through overshooting, while the batch size changes are most likely to ensure the quality of the end step.
3. Process Characteristics: Step size typically determines the step's direction and amplitude, while batch size determines the amount of variability/noise within the gradient estimates.

Insights: Step-size selection is superior to batch-size selection for converging towards the global optimum within this optimization context. The step size must be wide enough not to get caught into poor local optima but not too wide relative to stable convergence.

PART C

On comparing the performances of the four gradient descent optimization algorithms: Polyak step size, RMSProp, Heavy Ball (momentum), and Adam. By training each approach with batch size 5 and their performances relative to the constant step size approach from part (b) are shown as the baseline.

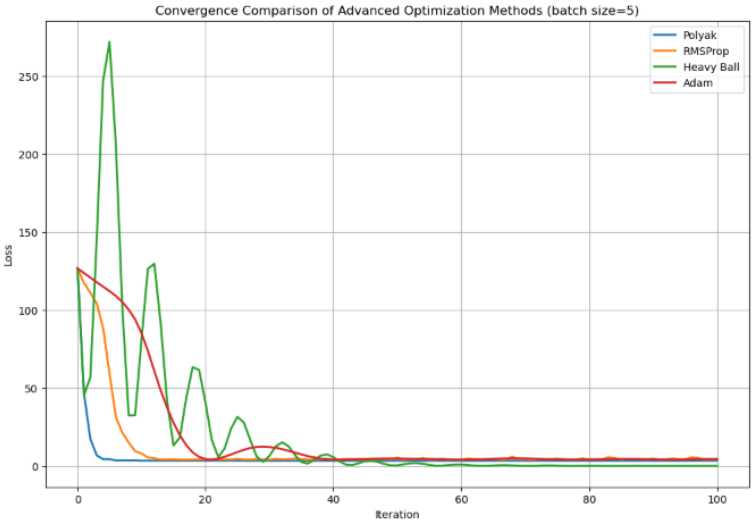


Fig 16, Convergence Comparison of Optimization Methods

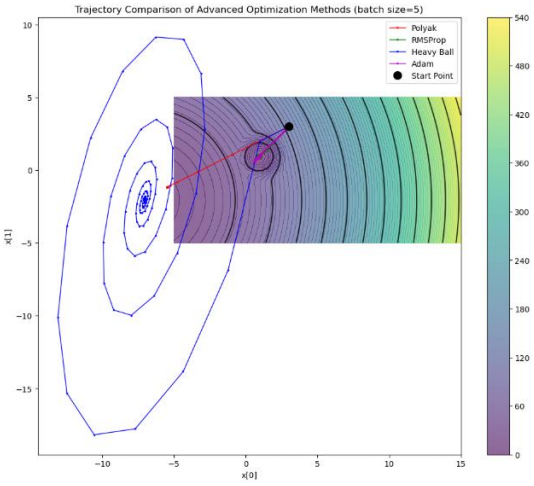


Fig 15, Trajectory Comparison of Advanced Optimization Methods

(i) Polyak Step Size

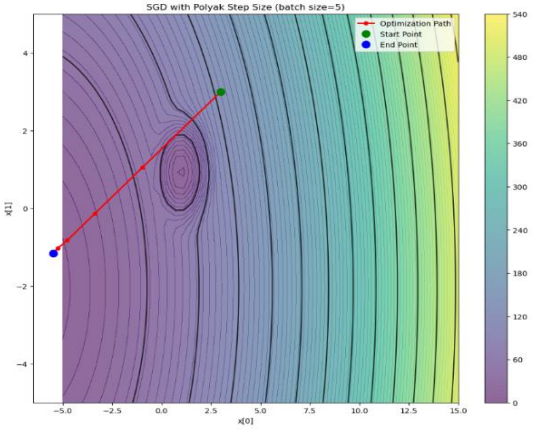


Fig 11, SGD with Polyak Step Size

Polyak step varies the learning factor adaptively based upon the current function value and the value assumed to be the optimum value:

$$\eta\_t = (f(x\_t) - f*) / ||\nabla f(x\_t)||^2$$

To run, I utilized the estimate $f* = 0.1$ (slightly less than the smallest observed value) and set the upper step limit to 0.2 for the sake of stability.

Results: The Polyak method converged to [-5.471, -1.159] with a final loss value of 3.411. As can be seen from Figure 16 (convergence comparison) and Figure 11 (Polyak trajectory), this method:

- Follows a relatively direct path towards a low level.
- It converges quickly with minimal oscillations
- It does converge towards the middle minimum and not the global minimum as with the fixed step sizes.

The quality of the Polyak method depends heavily upon the estimate of f*. If f* is too low (optimistic), step sizes are too large and may cause instability. If f* is too high (conservative), the convergence may be very gradual.

(ii) RMSProp

RMSProp scales the learning rate for every parameter by the moving average of the squared gradients:

$$s\_t = \beta \cdot s\_\{t-1\} + (1-\beta) \cdot (\nabla f\_t)^2$$

$$x\_\{t+1\} = x\_t - \eta / \sqrt{(s\_t + \varepsilon)} \cdot \nabla f\_t$$

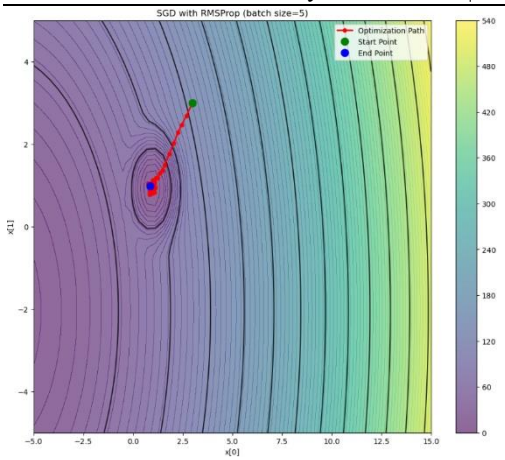


Fig 12, SGD with RMSProp

Selected parameters:

- Step size (η): 0.1 (as the performance was satisfactory in part b)
- Decay rate (β): 0.9 (default value utilized for balancing recent and previous gradients)
- ε: 10^{-8} (a very small number to prevent division by

Results: RMSProp converged to [0.856, 0.982] with final loss value 4.513. As can be seen from Figure 16 and Figure 12 (trajectory for RMSProp):

- This converges very rapidly towards the local minimum around [0.9, 0.9]
- It follows a relatively direct route with minimal meander.
- But it remains trapped within the poor local minimum rather than the global minimum.

This process explains how adaptive strategies find it more expedient to converge towards proximal local optima rather than looking further for better solutions.

(iii) Heavy Ball (Momentum)

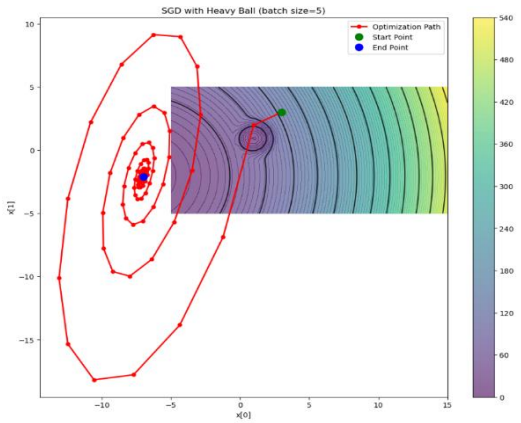


Fig 13, SGD with Heavy Ball

The Heavy Ball method uses momentum to accelerate the convergence:

$$v\_t = \beta \cdot v\_\{t-1\} - \eta \cdot \nabla f\_t$$
$$x\_\{t+1\} = x\_t$$

Selected parameters:

- Step size (η): 0.1 (same as with other methods)
- Beta (momentum coefficient): 0.9 (Default value with good acceleration with no oscillation)

Result: Heavy Ball converged to [-7.004, -2.108] with the last loss 0.110. As illustrated in Figure 16 and Figure 13 (Heavy Ball trajectory):

- Technique shows unbridled oscillations with gigantic spiraling patterns.
- It searches for a much larger portion of the parameter space with optimization
- Particularly most notably, it locates the world's smallest region around [-7, -2] correctly.

Heavy Ball is the sole sophisticated technique for converging reliably to the global optimum within this environment. Momentum helps it escape the local minimum where RMSProp and Adam were trapped, highlighting the power of momentum within challenging loss environments.

(iv) Adam

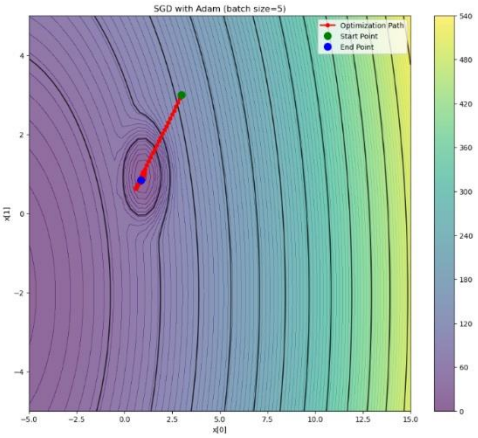


Fig 14, SGD with Adam

Adam combines momentum and adaptive learning rates:

$$m\_t = \beta_1 \cdot m\_\{t-1\} + (1-\beta_1) \cdot \nabla f\_t$$
$$v\_t = \beta_2 \cdot v\_\{t-1\} + (1-\beta_2) \cdot (\nabla f\_t)^2$$
$$\hat{m}\_t = m\_t/(1-\beta_1{}^t) \ \#\ Bias\ correction$$
$$\hat{v}\_t = v\_t/(1-\beta_2{}^t) \ \#\ Bias\ correction$$
$$x\_\{t+1\} = x\_t - \eta \cdot \hat{m}\_t/\sqrt{(\hat{v}\_t + \varepsilon)}$$

Selected Standard Parameters:

- Step size (η): 0.1
- β_1 : 0.9 (for momentum)
- β_2: 0.999 (for squared gradients)
- ε: 10^{-8}

Results: Adam converged towards [0.879, 0.842] with the final loss being 4.577. As shown by Figure 16 and Figure 14 (trajectory of Adam):

- This method follows a very direct path towards the local optimum.
- It does not vibrate, representative of Adam's solidity

- Like RMSProp, it is caught in the bad local minimum around [0.9, 0.9].
- This is not expected since Adam generally performs well with different sorts of issues.

Comparative Analysis

Figure 16 shows the convergence plots for all four approaches side by side for comparison:

- Convergence Speed: Polyak converges the quickest followed by RMSProp and then Adam with Heavy Ball converging with the most oscillations.
- Stability: Heavy Ball has very large oscillations throughout optimization, reflecting its momentum-based approach
- Final Value of Loss: Heavy Ball has the smallest loss value (0.110), with Polyak (3.411), RMSProp (4.513), and Adam (4.577

Figure 15 superimposes all the trajectories onto a single contour plot and illustrates how each approach traverses the loss landscape: Heavy Ball traces out large spiraling trajectories within the parameter space before converging towards the global optimum Polyak follows a direct route towards the intermediate minimum. Both Adam and RMSProp converge rapidly towards the local minimum around [0.9, 0.9].

Batch Size Effect on Advanced Techniques

The effect the batch size had upon Adam's performance, as shown through Figure 17 and the results table:

Batch Size	Final Position (Adam)	Final Loss
1	[1.116, 0.935]	5.027
5	[0.964, 0.890]	4.117
10	[0.925, 0.882]	4.178
20	[0.939, 0.924]	4.060
25	[0.945, 0.923]	4.055

Table 6, Summary of Loss in Different Batch Size

Fig. 17 shows very comparable convergence behavior for all batch sizes. Mainly, all batch sizes converge with Adam into the same local minimal region rather than the global one. It follows that:

1. The basin of attraction of the local minimum influences Adam significantly irrespective of batch size.
2. Adam's adaptive learning rates lead to dropping into close minima quickly
3. The regularizing effect of the small batch sizes is not sufficient for Adam to escape the local minimum

Comparison with Constant Step Size (Baseline)

Compared with the constant step size results in part (b):

Minimum Finding Ability: Heavy Ball and Constant step size (≥ 0.05) converge to the global minimum, Polyak converges to an intermediate point of the minimum, and RMSProp and Adam converge into a poor local minimum.

Convergence Behavior:

- Constant step size: Steady step with smooth convergence
- Heavy Ball: Oscillatory path with convergence towards global optimality
- Adaptive algorithms (Adam and RMSProp): Quick convergence towards the local optimum.

Sensitivity to Parameters: The fixed step-size approach is highly sensitive to the step-size hyperparameter (as can be seen from part b(iv)), while advanced approaches are very robust against their hyperparameter but also possess the potential for becoming trapped into worse solutions.

Conclusion

This analogy offers several striking observations regarding the behavior of optimization algorithms.

- No Universal Optimal Solution: Although Adam is generally considered a strong optimizer, it also performed very poorly here, while Heavy Ball performed very well.
- Exploration v. Exploitation: Heavy Ball momentum algorithms are capable of more exploration and thus can escape local minimum, while adaptive algorithms (e.g. RMSProp, Adam) are good for exploitation as they converge very fast towards neighboring solutions.
- Problem Specificity: The behavior of optimization algorithms is highly problem-specific within the loss space. With this function with many minima, algorithms with larger or more exploration step sizes perform better.
- Step Size Importance: Step selection remains crucial regardless of advanced methodologies because step size determines the ability of the algorithm to escape from local optima to find the global extremum.

APPENDIX

```
# %%
### Function that is given
# %% [markdown]
# import numpy as np
#
# 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(38*(z[0]**2+z[1]**2), (z[0]+8)**2+(z[1]+3)**2)
#         count=count+1
#     return y/count
# %%
### a(i) Mini Batch SGD Implementation
# %%
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
# Loss function provided in the assignment
def generate_trainingdata(m=25):
    return np.array([0,0])+0.25*np.random.randn(m,2)
def f(x, minibatch):
    # loss function sum_{w in training data} f(x,w)
    y=0; count=0
    for w in minibatch:
        z=x-w-1
        y=y+min(38*(z[0]**2+z[1]**2), (z[0]+8)**2+(z[1]+3)**2)
        count=count+1
    return y/count
# Finite difference gradient calculation
def gradient_f(x, minibatch, h=1e-6):
    """Calculate gradient of f using finite differences."""
    grad = np.zeros(2)
    for i in range(2):
        x_plus_h = x.copy()
        x_plus_h[i] += h
        grad[i] = (f(x_plus_h, minibatch) - f(x, minibatch)) / h
    return grad
class SGDOptimizer:
    def __init__(self, initial_x, training_data):
        """
        Initialize the SGD optimizer.
        Parameters:
        - initial_x: Starting point (numpy array of shape (2,))
        - training_data: Full training dataset
        """
        self.x = initial_x.copy()
        self.training_data = training_data
        # For tracking progress
        self.losses = []
        self.trajectory = [initial_x.copy()]
    def _get_mini_batch(self, batch_size):
        """Randomly select a mini-batch from the training data."""
        indices = np.random.choice(len(self.training_data), batch_size, replace=False)
        return self.training_data[indices]
    def constant_step(self, step_size, batch_size, max_iterations=100):
        """
        Mini-batch SGD with constant step size.

        Parameters:
        - step_size: Learning rate
        - batch_size: Size of mini-batches
        - max_iterations: Maximum number of iterations
        Returns:
        - x: Final position
        - losses: List of loss values at each iteration
        - trajectory: List of positions at each iteration
        """
        self.x = self.trajectory[0].copy()  # Reset to initial position
        self.losses = [f(self.x, self.training_data)]
        self.trajectory = [self.x.copy()]
```

```
        for _ in range(max_iterations):
            # Get mini-batch
            mini_batch = self._get_mini_batch(batch_size)
            # Calculate gradient
            grad = gradient_f(self.x, mini_batch)
            # Update x
            self.x = self.x - step_size * grad
            # Store loss and trajectory
            self.losses.append(f(self.x, self.training_data))
            self.trajectory.append(self.x.copy())
        return self.x, self.losses, self.trajectory
    def polyak_step(self, batch_size, f_star, max_step_size=0.1, max_iterations=100):
        """
        Mini-batch SGD with Polyak step size.
        Parameters:
        - batch_size: Size of mini-batches
        - f_star: Optimal function value
        - max_step_size: Maximum allowed step size (for stability)
        - max_iterations: Maximum number of iterations
        Returns:
        - x: Final position
        - losses: List of loss values at each iteration
        - trajectory: List of positions at each iteration
        """
        self.x = self.trajectory[0].copy()  # Reset to initial position
        self.losses = [f(self.x, self.training_data)]
        self.trajectory = [self.x.copy()]

        for _ in range(max_iterations):
            # Get mini-batch
            mini_batch = self._get_mini_batch(batch_size)
            # Calculate gradient
            grad = gradient_f(self.x, mini_batch)
            # Calculate current loss on mini-batch
            current_f = f(self.x, mini_batch)
            # Calculate Polyak step size
            grad_norm_squared = np.sum(grad**2)
            if grad_norm_squared < 1e-10 or current_f <= f_star:
                step = 0  # No step if gradient is very small or already at optimum
            else:
                step = min((current_f - f_star) / grad_norm_squared, max_step_size)
            # Update x
            self.x = self.x - step * grad
            # Store loss and trajectory
            self.losses.append(f(self.x, self.training_data))
            self.trajectory.append(self.x.copy())
        return self.x, self.losses, self.trajectory
    def rmsprop(self, step_size, batch_size, beta=0.9, epsilon=1e-8, max_iterations=100):
        """
        Mini-batch SGD with RMSProp.
        Parameters:
        - step_size: Base learning rate
        - batch_size: Size of mini-batches
        - beta: Decay rate for moving average
        - epsilon: Small constant to avoid division by zero
        - max_iterations: Maximum number of iterations
        Returns:
        - x: Final position
        - losses: List of loss values at each iteration
        - trajectory: List of positions at each iteration
        """
        self.x = self.trajectory[0].copy()  # Reset to initial position
        self.losses = [f(self.x, self.training_data)]
        self.trajectory = [self.x.copy()]
        # Initialize squared gradient accumulator
        s = np.zeros_like(self.x)
        for _ in range(max_iterations):
            # Get mini-batch
            mini_batch = self._get_mini_batch(batch_size)
            # Calculate gradient
            grad = gradient_f(self.x, mini_batch)
            # Update squared gradient accumulator
            s = beta * s + (1 - beta) * grad**2
            # Calculate adaptive step size
            adjusted_step = step_size / (np.sqrt(s) + epsilon)
            # Update x
            self.x = self.x - adjusted_step * grad
```

```
            # Store loss and trajectory
            self.losses.append(f(self.x, self.training_data))
            self.trajectory.append(self.x.copy())

        return self.x, self.losses, self.trajectory

    def heavy_ball(self, step_size, batch_size, beta=0.9, max_iterations=100):
        """
        Mini-batch SGD with Heavy Ball (momentum).
        Parameters:
        - step_size: Learning rate
        - batch_size: Size of mini-batches
        - beta: Momentum parameter
        - max_iterations: Maximum number of iteration
        Returns:
        - x: Final position
        - losses: List of loss values at each iteration
        - trajectory: List of positions at each iteration
        """
        self.x = self.trajectory[0].copy()  # Reset to initial position
        self.losses = [f(self.x, self.training_data)]
        self.trajectory = [self.x.copy()]
        # Initialize velocity
        v = np.zeros_like(self.x)
        for _ in range(max_iterations):
            # Get mini-batch
            mini_batch = self._get_mini_batch(batch_size)
            # Calculate gradient
            grad = gradient_f(self.x, mini_batch)
            # Update with momentum
            v = beta * v - step_size * grad
            self.x = self.x + v
            # Store loss and trajectory
            self.losses.append(f(self.x, self.training_data))
            self.trajectory.append(self.x.copy())
        return self.x, self.losses, self.trajectory

    def adam(self, step_size, batch_size, beta1=0.9, beta2=0.999, epsilon=1e-8, max_iterations=100):
        """
        Mini-batch SGD with Adam optimizer.
        Parameters:
        - step_size: Base learning rate
        - batch_size: Size of mini-batches
        - beta1: Exponential decay rate for first moment
        - beta2: Exponential decay rate for second moment
        - epsilon: Small constant to avoid division by zero
        - max_iterations: Maximum number of iterations
        Returns:
        - x: Final position
        - losses: List of loss values at each iteration
        - trajectory: List of positions at each iteration
        """
        self.x = self.trajectory[0].copy()  # Reset to initial position
        self.losses = [f(self.x, self.training_data)]
        self.trajectory = [self.x.copy()]
        # Initialize moment estimates
        m = np.zeros_like(self.x)  # First moment
        v = np.zeros_like(self.x)  # Second moment
        t = 0  # Timestep
        for _ in range(max_iterations):
            t += 1
            # Get mini-batch
            mini_batch = self._get_mini_batch(batch_size)
            # Calculate gradient
            grad = gradient_f(self.x, mini_batch)
            # Update biased first moment estimate
            m = beta1 * m + (1 - beta1) * grad
            # Update biased second raw moment estimate
            v = beta2 * v + (1 - beta2) * (grad**2)
            # Compute bias-corrected first moment estimate
            m_hat = m / (1 - beta1**t)
            # Compute bias-corrected second raw moment estimate
            v_hat = v / (1 - beta2**t)
            # Update parameters
            self.x = self.x - step_size * m_hat / (np.sqrt(v_hat) + epsilon)
            # Store loss and trajectory
            self.losses.append(f(self.x, self.training_data))
```

```
            self.trajectory.append(self.x.copy())
        return self.x, self.losses, self.trajectory
# Visualization functions
def plot_loss_surface(training_data, x_range=(-5, 15), y_range=(-5, 5), resolution=100):
    """Plot the loss function surface and contour."""
    x = np.linspace(x_range[0], x_range[1], resolution)
    y = np.linspace(y_range[0], y_range[1], resolution)
    X, Y = np.meshgrid(x, y)
    Z = np.zeros((resolution, resolution))
    for i in range(resolution):
        for j in range(resolution):
            Z[i, j] = f(np.array([X[i, j], Y[i, j]]), training_data)
    return X, Y, Z
def plot_wireframe(X, Y, Z, title='Loss Function Surface'):
    """Plot a 3D wireframe of the loss function."""
    fig = plt.figure(figsize=(12, 10))
    ax = fig.add_subplot(111, projection='3d')
    surf = ax.plot_surface(X, Y, Z, cmap='viridis', alpha=0.8,
                           linewidth=0, antialiased=True)
    ax.set_xlabel('x[0]')
    ax.set_ylabel('x[1]')
    ax.set_zlabel('Loss')
    ax.set_title(title)
    fig.colorbar(surf, shrink=0.5, aspect=5)
    return fig
def plot_contour(X, Y, Z, trajectory=None, title='Loss Function Contour'):
    """Plot a contour map of the loss function with optional trajectory."""
    fig = plt.figure(figsize=(12, 10))
    # Create contour plot
    contour = plt.contour(X, Y, Z, 15, colors='black', alpha=0.8)
    filled_contour = plt.contourf(X, Y, Z, 100, cmap='viridis', alpha=0.6)
    plt.colorbar(filled_contour)
    # Add trajectory if provided
    if trajectory is not None:
        traj_x = [point[0] for point in trajectory]
        traj_y = [point[1] for point in trajectory]
        plt.plot(traj_x, traj_y, 'r.-', linewidth=2, markersize=10, label='Optimization Path')
        plt.plot(traj_x[0], traj_y[0], 'go', markersize=10, label='Start Point')
        plt.plot(traj_x[-1], traj_y[-1], 'bo', markersize=10, label='End Point')
        plt.legend()
    plt.xlabel('x[0]')
    plt.ylabel('x[1]')
    plt.title(title)
    return fig
def plot_loss_vs_iterations(losses, title='Loss vs. Iterations'):
    """Plot how the loss changes over iterations."""
    fig = plt.figure(figsize=(10, 6))
    plt.plot(losses, 'b-', linewidth=2)
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.title(title)
    plt.grid(True)
    return fig
# %%
#### a(ii) (2 plots) (loss_surface_wireframe, loss_surface_contour.png)
# %%
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
# Functions from previous code
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(38*(z[0]**2+z[1]**2), (z[0]+8)**2+(z[1]+3)**2)
        count=count+1
    return y/count
# Helper visualization functions
def plot_loss_surface(training_data, x_range=(-5, 15), y_range=(-5, 5), resolution=100):
    """Plot the loss function surface and contour."""
    x = np.linspace(x_range[0], x_range[1], resolution)
    y = np.linspace(y_range[0], y_range[1], resolution)
    X, Y = np.meshgrid(x, y)
    Z = np.zeros((resolution, resolution))
```

```
    for i in range(resolution):
        for j in range(resolution):
            Z[i, j] = f(np.array([X[i, j], Y[i, j]]), training_data)
    return X, Y, Z
def visualize_loss_function():
    """
    Generate training data and visualize the loss function surface and contour.
    """
    # Set random seed for reproducibility
    np.random.seed(42)
    # Generate training data
    training_data = generate_trainingdata(25)
       # First, explore some function values to determine a good plotting range
    test_points = [
        (-5, -5), (0, 0), (5, 0), (10, 0), (10, 5), (0, 5), (5, 5)
    ]
    print("Function values at various points:")
    for x, y in test_points:
        point = np.array([x, y])
        value = f(point, training_data)
        print(f"f([{x}, {y}]) = {value:.4f}")
      # Based on exploration, choose appropriate ranges for visualization
    # The loss landscape features are between these ranges
    x_range = (-5, 15)
    y_range = (-5, 5)
    print(f"\nPlotting loss function over x range {x_range} and y range {y_range}")
    print("This range was chosen to capture the minimum and important features of the loss landscape.")
    # Generate loss surface
    X, Y, Z = plot_loss_surface(training_data, x_range, y_range, resolution=50)
    # Plot wireframe
    fig = plt.figure(figsize=(12, 10))
    ax = fig.add_subplot(111, projection='3d')
    surf = ax.plot_surface(X, Y, Z, cmap='viridis', alpha=0.8,
                           linewidth=0, antialiased=True
    ax.set_xlabel('x[0]')
    ax.set_ylabel('x[1]')
    ax.set_zlabel('Loss')
    ax.set_title('Loss Function Surface (Full Training Data)')
    fig.colorbar(surf, shrink=0.5, aspect=5)
    # plt.savefig('loss_surface_wireframe.png')
    plt.show()
    plt.close(fig)
    # Plot contour
    fig = plt.figure(figsize=(12, 10))
    contour = plt.contour(X, Y, Z, 15, colors='black', alpha=0.8)
    filled_contour = plt.contourf(X, Y, Z, 100, cmap='viridis', alpha=0.6)
    plt.colorbar(filled_contour)
    plt.xlabel('x[0]')
    plt.ylabel('x[1]')
    plt.title('Loss Function Contour (Full Training Data)')
    # plt.savefig('loss_surface_contour.png')
    plt.show()
    plt.close(fig)
    # Find approximate minimum
    min_idx = np.unravel_index(np.argmin(Z), Z.shape)
    min_x = X[min_idx]
    min_y = Y[min_idx]
    min_z = Z[min_idx]
    print(f"\nApproximate minimum found at: [{min_x}, {min_y}] with value {min_z:.4f}")
    return X, Y, Z, (min_x, min_y, min_z)
# Run visualization
if __name__ == "__main__":
    X, Y, Z, min_point = visualize_loss_function()
# %%
## to test the data generation part
# %%
np.random.seed(42)  # For reproducibility
training_data = generate_trainingdata(25)
print("First few points of training data:", training_data[:3])
# %%
# to see the the local minima
# %%
# Test more points systematically
x_values = np.linspace(-10, 15, 26)
y_values = np.linspace(-10, 5, 16)
min_loss = float('inf')
```

```
min_point = None
for x in x_values:
    for y in y_values:
        point = np.array([x, y])
        loss_val = f(point, training_data)
        if loss_val < min_loss:
            min_loss = loss_val
            min_point = point
print(f"Approximate minimum found at: {min_point} with value {min_loss:.4f}")
# %%
#### a(iii) Calculate Derivatives Using Finite Differences (1 plot) gradient_field.png
# %%
import numpy as np
import matplotlib.pyplot as plt
# Functions from previous code
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(38*(z[0]**2+z[1]**2), (z[0]+8)**2+(z[1]+3)**2)
        count=count+1
    return y/count
def finite_difference_gradient(func, x, minibatch, h=1e-6):
    """
    Calculate the gradient of func at point x using finite differences.
       Parameters:
    - func: The function to differentiate
    - x: The point at which to calculate the gradient
    - minibatch: The mini-batch to use for the function
    - h: Step size for finite difference
    Returns:
    - grad: The gradient vector [∂f/∂x₁, ∂f/∂x₂]
    """
    grad = np.zeros_like(x)
    f_x = func(x, minibatch)
    for i in range(len(x)):
        x_plus_h = x.copy()
        x_plus_h[i] += h
        grad[i] = (func(x_plus_h, minibatch) - f_x) / h
    return grad
def finite_difference_hessian(func, x, minibatch, h=1e-5):
    """
    Calculate the Hessian matrix of func at point x using finite differences.
    Parameters:
    - func: The function to differentiate
    - x: The point at which to calculate the Hessian
    - minibatch: The mini-batch to use for the function
    - h: Step size for finite difference
    Returns:
    - hess: The Hessian matrix [[∂²f/∂x₁², ∂²f/∂x₁∂x₂], [∂²f/∂x₂∂x₁, ∂²f/∂x₂²]]
    """
    n = len(x)
    hess = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            x_plus_h_i = x.copy()
            x_plus_h_i[i] += h
            x_plus_h_j = x.copy()
            x_plus_h_j[j] += h
            x_plus_h_ij = x.copy()
            x_plus_h_ij[i] += h
            x_plus_h_ij[j] += h
            # Mixed partial derivative
            if i == j:
                # For diagonal elements, use the standard second derivative formula
                hess[i, j] = (func(x_plus_h_i + h, minibatch) - 2 * func(x_plus_h_i, minibatch) + func(x,
minibatch)) / (h * h)
            else:
                # For off-diagonal elements, use the mixed partial derivative formula
                hess[i, j] = (func(x_plus_h_ij, minibatch) - func(x_plus_h_i, minibatch) - func(x_plus_h_j,
minibatch) + func(x, minibatch)) / (h * h)
    return hess
def validate_gradient(x_values, training_data):
    """
```

```
    Calculate and print gradients at multiple points to validate the implementation.
    Parameters:
    - x_values: List of points at which to calculate gradients
    - training_data: Training data to use for the function
    """
    print("Gradients at various points:")
    for x in x_values:
        grad = finite_difference_gradient(f, np.array(x), training_data)
        print(f"∇f({x}) = {grad}")
def analyze_derivatives():
    """
    Analyze the derivatives of the loss function.
    """
    # Set random seed for reproducibility
    np.random.seed(42)
    # Generate training data
    training_data = generate_trainingdata(25)
    # Points to analyze
    points = [
        (-2, -2),  # Far from minimum
        (0, 0),    # Near the center
        (9, 0),    # Near expected minimum based on visualization
        (9.5, 0.5) # Another point near minimum
    ]
    # Validate gradients
    validate_gradient(points, training_data)
    # Calculate and visualize gradients on the loss surface
    # We'll create a vector field showing gradient directions
    x_range = (-5, 15)
    y_range = (-5, 5)
    grid_size = 10
    x = np.linspace(x_range[0], x_range[1], grid_size)
    y = np.linspace(y_range[0], y_range[1], grid_size)
    X, Y = np.meshgrid(x, y)
    # Calculate function values for contour plot
    Z = np.zeros((grid_size, grid_size))
    U = np.zeros((grid_size, grid_size))  # x-component of gradient
    V = np.zeros((grid_size, grid_size))  # y-component of gradient
    for i in range(grid_size):
        for j in range(grid_size):
            point = np.array([X[i, j], Y[i, j]])
            Z[i, j] = f(point, training_data)
            grad = finite_difference_gradient(f, point, training_data)
            U[i, j] = -grad[0]  # Negative because gradient points in direction of steepest ascent
            V[i, j] = -grad[1]  # But we want direction of steepest descent

    # Plot contour with gradient vectors
    plt.figure(figsize=(12, 10))
    # Create contour plot
    contour = plt.contour(X, Y, Z, 15, colors='black', alpha=0.4)
    filled_contour = plt.contourf(X, Y, Z, 100, cmap='viridis', alpha=0.6)
    plt.colorbar(filled_contour)
       # Normalize the gradient vectors for better visualization
    magnitude = np.sqrt(U**2 + V**2)
    max_magnitude = np.max(magnitude)
    U = U / max_magnitude
    V = V / max_magnitude
    # Plot gradient vectors
    plt.quiver(X, Y, U, V, color='red', width=0.002, scale=30)
    plt.xlabel('x[0]')
    plt.ylabel('x[1]')
    plt.title('Loss Function Contour with Gradient Directions')
    plt.show()
    # plt.savefig('gradient_field.png')
    plt.close()
    # Calculate Hessian at the approximate minimum to analyze curvature
    min_point = np.array([9, 0])  # Approximate minimum from visualization
    hessian = finite_difference_hessian(f, min_point, training_data)
    print("\nHessian matrix at approximate minimum:", min_point)
    print(hessian)
    # Calculate eigenvalues to analyze curvature at the minimum
    eigvals = np.linalg.eigvals(hessian)
    print("\nEigenvalues of the Hessian:", eigvals)
    print(f"Condition number: {max(abs(eigvals))/min(abs(eigvals)):.4f}")
    # If eigenvalues are positive, the point is a local minimum
    if np.all(eigvals > 0):
        print("All eigenvalues are positive, confirming this is a local minimum.")
```

```
    else:
        print("Not all eigenvalues are positive, this might not be a minimum.")
# Run derivative analysis
if __name__ == "__main__":
    analyze_derivatives()
# %%
### b (i) Gradient Descent with Constant Step Size (2 plots) gd_trajectory.png, gd_loss_convergence.png
# %%
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
# Functions from previous code
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(38*(z[0]**2+z[1]**2), (z[0]+8)**2+(z[1]+3)**2)
        count=count+1
    return y/count
def gradient_f(x, minibatch, h=1e-6):
    """Calculate gradient of f using finite differences."""
    grad = np.zeros(2)
    for i in range(2):
        x_plus_h = x.copy()
        x_plus_h[i] += h
        grad[i] = (f(x_plus_h, minibatch) - f(x, minibatch)) / h
    return grad
def plot_loss_surface(training_data, x_range=(-5, 15), y_range=(-5, 5), resolution=50):
    """Plot the loss function surface and contour."""
    x = np.linspace(x_range[0], x_range[1], resolution)
    y = np.linspace(y_range[0], y_range[1], resolution)
    X, Y = np.meshgrid(x, y)
    Z = np.zeros((resolution, resolution))
    for i in range(resolution):
        for j in range(resolution):
            Z[i, j] = f(np.array([X[i, j], Y[i, j]]), training_data)
    return X, Y, Z
def plot_contour(X, Y, Z, trajectory=None, title='Loss Function Contour'):
    """Plot a contour map of the loss function with optional trajectory."""
    fig = plt.figure(figsize=(12, 10))
    # Create contour plot
    contour = plt.contour(X, Y, Z, 15, colors='black', alpha=0.8)
    filled_contour = plt.contourf(X, Y, Z, 100, cmap='viridis', alpha=0.6)
    plt.colorbar(filled_contour)
    # Add trajectory if provided
    if trajectory is not None:
        traj_x = [point[0] for point in trajectory]
        traj_y = [point[1] for point in trajectory]
        plt.plot(traj_x, traj_y, 'r.-', linewidth=2, markersize=10, label='Optimization Path')
        plt.plot(traj_x[0], traj_y[0], 'go', markersize=10, label='Start Point')
        plt.plot(traj_x[-1], traj_y[-1], 'bo', markersize=10, label='End Point')
        plt.legend()
    plt.xlabel('x[0]')
    plt.ylabel('x[1]')
    plt.title(title)
    return fig
def gradient_descent_constant_step(initial_x, training_data, step_size, max_iterations=100):
    """
    Standard gradient descent with constant step size.
    Parameters:
    - initial_x: Starting point (numpy array)
    - training_data: Full training dataset
    - step_size: Constant learning rate
    - max_iterations: Maximum number of iterations
    Returns:
    - x: Final position
    - losses: List of loss values at each iteration
    - trajectory: List of positions at each iteration
    """
    x = initial_x.copy()
    losses = [f(x, training_data)]
    trajectory = [x.copy()]
    for i in range(max_iterations):
        # Calculate gradient using the full training data
```

```
        grad = gradient_f(x, training_data)
        # Update x
        x = x - step_size * grad
        # Store loss and trajectory
        losses.append(f(x, training_data))
        trajectory.append(x.copy())
    return x, losses, trajectory
def run_gradient_descent_experiment():
    """
    Run gradient descent with constant step size and analyze the results.
    """
    # Set random seed for reproducibility
    np.random.seed(42)
    # Generate training data
    training_data = generate_trainingdata(25)
    # Initial point
    initial_x = np.array([3.0, 3.0])
    # Step size selection
    # We'll test a few step sizes to find an appropriate one
    step_sizes = [0.01, 0.05, 0.1, 0.2, 0.5]
    # Store results for each step size
    results = {}
    for step_size in step_sizes:
        final_x, losses, trajectory = gradient_descent_constant_step(
            initial_x, training_data, step_size, max_iterations=50
        )
        results[step_size] = (final_x, losses, trajectory)
        print(f"Step size {step_size}:")
        print(f"  Final x: {final_x}")
        print(f"  Final loss: {losses[-1]:.6f}")
        print(f"  Loss reduction: {(losses[0] - losses[-1]) / losses[0] * 100:.2f}%")
    # Compute loss surface for visualization
    X, Y, Z = plot_loss_surface(training_data)
    # Select the best step size based on results
    best_step_size = min(results.keys(), key=lambda ss: results[ss][1][-1])
    print(f"\nBest step size: {best_step_size}")
    # Plot convergence for the best step size
    _, best_losses, best_trajectory = results[best_step_size]
    plt.figure(figsize=(10, 6))
    plt.plot(best_losses, 'b-', linewidth=2)
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.title(f'Loss vs. Iterations (Step size = {best_step_size})')
    plt.grid(True)
    plt.show()
    # plt.savefig('gd_loss_convergence.png')
    plt.close()
    # Plot trajectory on contour plot
    fig = plot_contour(X, Y, Z, best_trajectory,
                       title=f'Gradient Descent Trajectory (Step size = {best_step_size})')
    # plt.savefig('gd_trajectory.png')
    plt.show()
    plt.close(fig)
    return best_step_size, results[best_step_size]
# Run gradient descent experiment
if __name__ == "__main__":
    best_step_size, (final_x, losses, trajectory) = run_gradient_descent_experiment()
# %%
### b (ii,iii,iv) mini-batch SGD with different batch sizes to compare with the gradient descent:
### sgd_loss_convergence.png, sgd_multiple_runs.png, batch_size_comparison.png
# %%
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
# Functions from previous code
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(38*(z[0]**2+z[1]**2), (z[0]+8)**2+(z[1]+3)**2)
        count=count+1
    return y/count
def gradient_f(x, minibatch, h=1e-6):
    """Calculate gradient of f using finite differences."""
```

```
    grad = np.zeros(2)
    for i in range(2):
        x_plus_h = x.copy()
        x_plus_h[i] += h
        grad[i] = (f(x_plus_h, minibatch) - f(x, minibatch)) / h
    return grad
def plot_loss_surface(training_data, x_range=(-5, 15), y_range=(-5, 5), resolution=50):
    """Plot the loss function surface and contour."""
    x = np.linspace(x_range[0], x_range[1], resolution)
    y = np.linspace(y_range[0], y_range[1], resolution)
    X, Y = np.meshgrid(x, y)
    Z = np.zeros((resolution, resolution))
    for i in range(resolution):
        for j in range(resolution):
            Z[i, j] = f(np.array([X[i, j], Y[i, j]]), training_data)
    return X, Y, Z
def plot_contour(X, Y, Z, trajectory=None, title='Loss Function Contour'):
    """Plot a contour map of the loss function with optional trajectory."""
    fig = plt.figure(figsize=(12, 10))
    # Create contour plot
    contour = plt.contour(X, Y, Z, 15, colors='black', alpha=0.8)
    filled_contour = plt.contourf(X, Y, Z, 100, cmap='viridis', alpha=0.6)
    plt.colorbar(filled_contour)
    # Add trajectory if provided
    if trajectory is not None:
        traj_x = [point[0] for point in trajectory]
        traj_y = [point[1] for point in trajectory]
        plt.plot(traj_x, traj_y, 'r.-', linewidth=2, markersize=10, label='Optimization Path')
        plt.plot(traj_x[0], traj_y[0], 'go', markersize=10, label='Start Point')
        plt.plot(traj_x[-1], traj_y[-1], 'bo', markersize=10, label='End Point')
        plt.legend()
    plt.xlabel('x[0]')
    plt.ylabel('x[1]')
    plt.title(title)
    return fig
def sgd_constant_step(initial_x, training_data, step_size, batch_size, max_iterations=100):
    """
    Mini-batch SGD with constant step size.
    Parameters:
    - initial_x: Starting point (numpy array)
    - training_data: Full training dataset
    - step_size: Learning rate
    - batch_size: Size of mini-batches
    - max_iterations: Maximum number of iterations
    Returns:
    - x: Final position
    - losses: List of loss values at each iteration
    - trajectory: List of positions at each iteration
    """
    x = initial_x.copy()
    losses = [f(x, training_data)]  # Evaluate on full dataset for consistency
    trajectory = [x.copy()]
    for i in range(max_iterations):
        # Randomly select a mini-batch
        indices = np.random.choice(len(training_data), batch_size, replace=False)
        mini_batch = training_data[indices]
        # Calculate gradient on mini-batch
        grad = gradient_f(x, mini_batch)
        # Update x
        x = x - step_size * grad
        # Store loss and trajectory (evaluate loss on full dataset for fair comparison)
        losses.append(f(x, training_data))
        trajectory.append(x.copy())
    return x, losses, trajectory
def run_sgd_experiment(best_step_size):
    """
    Run mini-batch SGD experiment with various batch sizes and analyze results.
    Parameters:
    - best_step_size: The best step size found from gradient descent experiment
    """
    # Set random seed for reproducibility
    np.random.seed(42)
    # Generate training data
    training_data = generate_trainingdata(25)
    # Initial point
    initial_x = np.array([3.0, 3.0])
    # Compute loss surface for visualization
```

```
X, Y, Z = plot_loss_surface(training_data)
# Part (b)(ii): Mini-batch SGD with batch size 5
# Run multiple times to observe variance
batch_size = 5
num_runs = 5
print(f"\nRunning SGD with batch size {batch_size} and step size {best_step_size}")
# Store results for multiple runs
sgd_results = []
plt.figure(figsize=(10, 6))
   for run in range(num_runs):
    final_x, losses, trajectory = sgd_constant_step(
        initial_x, training_data, best_step_size, batch_size, max_iterations=100
    )
    sgd_results.append((final_x, losses, trajectory))

    print(f"Run {run+1}:")
    print(f"  Final x: {final_x}")
    print(f"  Final loss: {losses[-1]:.6f}")
    # Plot loss convergence
    plt.plot(losses, linewidth=1, label=f'Run {run+1}')
    # Plot trajectory
    fig = plot_contour(X, Y, Z, trajectory,
                       title=f'SGD Trajectory - Run {run+1} (batch size={batch_size}, η={best_step_size})')
    plt.savefig(f'sgd_trajectory_run{run+1}.png')
    plt.close(fig)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title(f'SGD Loss Convergence (batch size={batch_size}, η={best_step_size})')
plt.grid(True)
plt.legend()
plt.show()
# plt.savefig('sgd_loss_convergence.png')
plt.close()
# Plot all trajectories on the same contour
fig = plt.figure(figsize=(12, 10))
# Create contour plot
contour = plt.contour(X, Y, Z, 15, colors='black', alpha=0.8)
filled_contour = plt.contourf(X, Y, Z, 100, cmap='viridis', alpha=0.6)
plt.colorbar(filled_contour)
colors = ['r', 'g', 'b', 'm', 'c']
for i, (_, _, trajectory) in enumerate(sgd_results):
    traj_x = [point[0] for point in trajectory]
    traj_y = [point[1] for point in trajectory]
    plt.plot(traj_x, traj_y, f'{colors[i%len(colors)]}.-', linewidth=1, markersize=3,
             label=f'Run {i+1}')
plt.plot(initial_x[0], initial_x[1], 'ko', markersize=10, label='Start Point')
plt.legend()
plt.xlabel('x[0]')
plt.ylabel('x[1]')
plt.title(f'Multiple SGD Runs (batch size={batch_size}, η={best_step_size})')
# plt.savefig('sgd_multiple_runs.png')
plt.show()
plt.close()
# Part (b)(iii): Vary batch size
batch_sizes = [1, 5, 10, 20, 25]  # 25 = full batch (GD)
batch_results = {}
for bs in batch_sizes:
    print(f"\nRunning SGD with batch size {bs}")
    final_x, losses, trajectory = sgd_constant_step(
        initial_x, training_data, best_step_size, bs, max_iterations=100
    )
    batch_results[bs] = (final_x, losses, trajectory)
    print(f"Batch size {bs}:")
    print(f"  Final x: {final_x}")
    print(f"  Final loss: {losses[-1]:.6f}")
    # Plot trajectory
    fig = plot_contour(X, Y, Z, trajectory,
                       title=f'SGD Trajectory (batch size={bs}, η={best_step_size})')
    plt.savefig(f'sgd_trajectory_bs{bs}.png')
    plt.close(fig)
# Plot loss convergence for different batch sizes
plt.figure(figsize=(10, 6))
for bs, (_, losses, _) in batch_results.items():
    plt.plot(losses, linewidth=2, label=f'Batch Size {bs}')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title(f'Loss vs. Iterations for Different Batch Sizes (η={best_step_size})')
```

```
    plt.grid(True)
    plt.legend()
    # plt.savefig('batch_size_comparison.png')
    plt.show()
    plt.close()
    # Plot all final positions on the same contour
    fig = plt.figure(figsize=(12, 10))
    # Create contour plot
    contour = plt.contour(X, Y, Z, 15, colors='black', alpha=0.8)
    filled_contour = plt.contourf(X, Y, Z, 100, cmap='viridis', alpha=0.6)
    plt.colorbar(filled_contour)
    for bs, (final_x, _, trajectory) in batch_results.items():
        traj_x = [point[0] for point in trajectory]
        traj_y = [point[1] for point in trajectory]
        plt.plot(traj_x, traj_y, '.-', linewidth=1, markersize=3,
                 label=f'Batch Size {bs}')
        plt.plot(final_x[0], final_x[1], 'o', markersize=8)
    plt.plot(initial_x[0], initial_x[1], 'ko', markersize=10, label='Start Point')
    plt.legend()
    plt.xlabel('x[0]')
    plt.ylabel('x[1]')
    plt.title(f'Effect of Batch Size (η={best_step_size})')
    plt.show()
    # plt.savefig('batch_size_effect.png')
    plt.close()
    # Part (b)(iv): Vary step size with fixed batch size
    step_sizes = [0.01, 0.05, 0.1, 0.2, 0.5]
    fixed_batch_size = 5
    step_results = {}
    for ss in step_sizes:
        print(f"\nRunning SGD with step size {ss} and batch size {fixed_batch_size}")
        final_x, losses, trajectory = sgd_constant_step(
            initial_x, training_data, ss, fixed_batch_size, max_iterations=100
        )
        step_results[ss] = (final_x, losses, trajectory)
        print(f"Step size {ss}:")
        print(f"  Final x: {final_x}")
        print(f"  Final loss: {losses[-1]:.6f}")

        # Plot trajectory
        fig = plot_contour(X, Y, Z, trajectory,
                           title=f'SGD Trajectory (batch size={fixed_batch_size}, η={ss})')
        plt.savefig(f'sgd_trajectory_ss{ss}.png')
        plt.close(fig)
    # Plot loss convergence for different step sizes
    plt.figure(figsize=(10, 6))
    for ss, (_, losses, _) in step_results.items():
        plt.plot(losses, linewidth=2, label=f'Step Size {ss}')
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.title(f'Loss vs. Iterations for Different Step Sizes (batch size={fixed_batch_size})')
    plt.grid(True)
    plt.legend()
    # plt.savefig('step_size_comparison.png')
    plt.show()
    plt.close()
    # Plot all final positions on the same contour
    fig = plt.figure(figsize=(12, 10))
    # Create contour plot
    contour = plt.contour(X, Y, Z, 15, colors='black', alpha=0.8)
    filled_contour = plt.contourf(X, Y, Z, 100, cmap='viridis', alpha=0.6)
    plt.colorbar(filled_contour)
    for ss, (final_x, _, trajectory) in step_results.items():
        traj_x = [point[0] for point in trajectory]
        traj_y = [point[1] for point in trajectory]
        plt.plot(traj_x, traj_y, '.-', linewidth=1, markersize=3,
                 label=f'Step Size {ss}')
        plt.plot(final_x[0], final_x[1], 'o', markersize=8)
    plt.plot(initial_x[0], initial_x[1], 'ko', markersize=10, label='Start Point')
    plt.legend()
    plt.xlabel('x[0]')
    plt.ylabel('x[1]')
    plt.title(f'Effect of Step Size (batch size={fixed_batch_size})')
    # plt.savefig('step_size_effect.png')
    plt.show()
    plt.close()
    return batch_results, step_results
```

```
# Run SGD experiment
if __name__ == "__main__":
    # Assume we have the best step size from the previous experiment
    best_step_size = 0.1  # This should be the result from gradient_descent experiment
    batch_results, step_results = run_sgd_experiment(best_step_size)
# %%
#part c implement the various optimization methods (Polyak, RMSProp, Heavy Ball, and Adam)
# %%
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
# Functions from previous code
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(38*(z[0]**2+z[1]**2), (z[0]+8)**2+(z[1]+3)**2)
        count=count+1
    return y/count
def gradient_f(x, minibatch, h=1e-6):
    """Calculate gradient of f using finite differences."""
    grad = np.zeros(2)
    for i in range(2):
        x_plus_h = x.copy()
        x_plus_h[i] += h
        grad[i] = (f(x_plus_h, minibatch) - f(x, minibatch)) / h
    return grad
def plot_loss_surface(training_data, x_range=(-5, 15), y_range=(-5, 5), resolution=50):
    """Plot the loss function surface and contour."""
    x = np.linspace(x_range[0], x_range[1], resolution)
    y = np.linspace(y_range[0], y_range[1], resolution)
    X, Y = np.meshgrid(x, y)
    Z = np.zeros((resolution, resolution))
    for i in range(resolution):
        for j in range(resolution):
            Z[i, j] = f(np.array([X[i, j], Y[i, j]]), training_data)
    return X, Y, Z
def plot_contour(X, Y, Z, trajectory=None, title='Loss Function Contour'):
    """Plot a contour map of the loss function with optional trajectory."""
    fig = plt.figure(figsize=(12, 10))
    # Create contour plot
    contour = plt.contour(X, Y, Z, 15, colors='black', alpha=0.8)
    filled_contour = plt.contourf(X, Y, Z, 100, cmap='viridis', alpha=0.6)
    plt.colorbar(filled_contour)
    # Add trajectory if provided
    if trajectory is not None:
        traj_x = [point[0] for point in trajectory]
        traj_y = [point[1] for point in trajectory]
        plt.plot(traj_x, traj_y, 'r.-', linewidth=2, markersize=10, label='Optimization Path')
        plt.plot(traj_x[0], traj_y[0], 'go', markersize=10, label='Start Point')
        plt.plot(traj_x[-1], traj_y[-1], 'bo', markersize=10, label='End Point')
        plt.legend()
    plt.xlabel('x[0]')
    plt.ylabel('x[1]')
    plt.title(title)
    return fig
def polyak_sgd(initial_x, training_data, f_star, batch_size, max_step=0.2, max_iterations=100):
    """
    Mini-batch SGD with Polyak step size.
    Parameters:
    - initial_x: Starting point (numpy array)
    - training_data: Full training dataset
    - f_star: Optimal function value (approximate)
    - batch_size: Size of mini-batches
    - max_step: Maximum allowed step size
    - max_iterations: Maximum number of iterations
    Returns:
    - x: Final position
    - losses: List of loss values at each iteration
    - trajectory: List of positions at each iteration
    """
    x = initial_x.copy()
    losses = [f(x, training_data)]  # Evaluate on full dataset for consistency
    trajectory = [x.copy()]
```

```
    for i in range(max_iterations):
        # Randomly select a mini-batch
        indices = np.random.choice(len(training_data), batch_size, replace=False)
        mini_batch = training_data[indices]
        # Calculate gradient on mini-batch
        grad = gradient_f(x, mini_batch)
        # Calculate current loss on mini-batch
        current_f = f(x, mini_batch)
        # Calculate Polyak step size
        grad_norm_squared = np.sum(grad**2)
        if grad_norm_squared < 1e-10 or current_f <= f_star:
            step = 0  # No step if gradient is very small or already at optimum
        else:
            step = min((current_f - f_star) / grad_norm_squared, max_step)
        # Update x
        x = x - step * grad
        # Store loss and trajectory (evaluate loss on full dataset for fair comparison)
        losses.append(f(x, training_data))
        trajectory.append(x.copy())
    return x, losses, trajectory
def rmsprop_sgd(initial_x, training_data, step_size, batch_size, beta=0.9, epsilon=1e-8, max_iterations=100):
    """
    Mini-batch SGD with RMSProp.
    Parameters:
    - initial_x: Starting point (numpy array)
    - training_data: Full training dataset
    - step_size: Base learning rate
    - batch_size: Size of mini-batches
    - beta: Decay rate for moving average
    - epsilon: Small constant to avoid division by zero
    - max_iterations: Maximum number of iterations
    Returns:
    - x: Final position
    - losses: List of loss values at each iteration
    - trajectory: List of positions at each iteration
    """
    x = initial_x.copy()
    losses = [f(x, training_data)]  # Evaluate on full dataset for consistency
    trajectory = [x.copy()]
    # Initialize squared gradient accumulator
    s = np.zeros_like(x)
    for i in range(max_iterations):
        # Randomly select a mini-batch
        indices = np.random.choice(len(training_data), batch_size, replace=False)
        mini_batch = training_data[indices]
        # Calculate gradient on mini-batch
        grad = gradient_f(x, mini_batch)
        # Update squared gradient accumulator
        s = beta * s + (1 - beta) * grad**2
        # Calculate adaptive step size
        adjusted_step = step_size / (np.sqrt(s) + epsilon)
        # Update x
        x = x - adjusted_step * grad
        # Store loss and trajectory (evaluate loss on full dataset for fair comparison)
        losses.append(f(x, training_data))
        trajectory.append(x.copy())
    return x, losses, trajectory
def heavy_ball_sgd(initial_x, training_data, step_size, batch_size, beta=0.9, max_iterations=100):
    """
    Mini-batch SGD with Heavy Ball (momentum).
    Parameters:
    - initial_x: Starting point (numpy array)
    - training_data: Full training dataset
    - step_size: Learning rate
    - batch_size: Size of mini-batches
    - beta: Momentum parameter
    - max_iterations: Maximum number of iterations
    Returns:
    - x: Final position
    - losses: List of loss values at each iteration
    - trajectory: List of positions at each iteration
    """
    x = initial_x.copy()
    losses = [f(x, training_data)]  # Evaluate on full dataset for consistency
    trajectory = [x.copy()]
    # Initialize velocity
    v = np.zeros_like(x)
```

```
    for i in range(max_iterations):
        # Randomly select a mini-batch
        indices = np.random.choice(len(training_data), batch_size, replace=False)
        mini_batch = training_data[indices]
        # Calculate gradient on mini-batch
        grad = gradient_f(x, mini_batch)
        # Update with momentum
        v = beta * v - step_size * grad
        x = x + v
        # Store loss and trajectory (evaluate loss on full dataset for fair comparison)
        losses.append(f(x, training_data))
        trajectory.append(x.copy())
    return x, losses, trajectory
def adam_sgd(initial_x, training_data, step_size, batch_size, beta1=0.9, beta2=0.999, epsilon=1e-8,
max_iterations=100):
    """
    Mini-batch SGD with Adam optimizer.
    Parameters:
    - initial_x: Starting point (numpy array)
    - training_data: Full training dataset
    - step_size: Base learning rate
    - batch_size: Size of mini-batches
    - beta1: Exponential decay rate for first moment
    - beta2: Exponential decay rate for second moment
    - epsilon: Small constant to avoid division by zero
    - max_iterations: Maximum number of iterations
    Returns:
    - x: Final position
    - losses: List of loss values at each iteration
    - trajectory: List of positions at each iteration
    """
    x = initial_x.copy()
    losses = [f(x, training_data)]  # Evaluate on full dataset for consistency
    trajectory = [x.copy()]
    # Initialize moment estimates
    m = np.zeros_like(x)  # First moment
    v = np.zeros_like(x)  # Second moment
    t = 0  # Timestep
    for i in range(max_iterations):
        t += 1
        # Randomly select a mini-batch
        indices = np.random.choice(len(training_data), batch_size, replace=False)
        mini_batch = training_data[indices]
        # Calculate gradient on mini-batch
        grad = gradient_f(x, mini_batch)
        # Update biased first moment estimate
        m = beta1 * m + (1 - beta1) * grad
        # Update biased second raw moment estimate
        v = beta2 * v + (1 - beta2) * (grad**2)
        # Compute bias-corrected first moment estimate
        m_hat = m / (1 - beta1**t)
        # Compute bias-corrected second raw moment estimate
        v_hat = v / (1 - beta2**t)
        # Update parameters
        x = x - step_size * m_hat / (np.sqrt(v_hat) + epsilon)
        # Store loss and trajectory (evaluate loss on full dataset for fair comparison)
        losses.append(f(x, training_data))
        trajectory.append(x.copy())

    return x, losses, trajectory
def run_advanced_optimizer_experiment():
    """
    Run and compare the advanced optimization methods: Polyak, RMSProp, Heavy Ball, and Adam.
    """
    # Set random seed for reproducibility
    np.random.seed(42)
    # Generate training data
    training_data = generate_trainingdata(25)
    # Initial point
    initial_x = np.array([3.0, 3.0])
    # Compute loss surface for visualization
    X, Y, Z = plot_loss_surface(training_data)
    # Find approximate minimum value for Polyak step size
    min_idx = np.unravel_index(np.argmin(Z), Z.shape)
    f_star = Z[min_idx] * 0.95  # Slightly lower than observed minimum to ensure convergence
    # Fixed parameters
    batch_size = 5
```

```
max_iterations = 100
# Part (c)(i): Polyak step size
print("\nRunning SGD with Polyak step size")
final_x_polyak, losses_polyak, trajectory_polyak = polyak_sgd(
    initial_x, training_data, f_star, batch_size, max_step=0.2, max_iterations=max_iterations
)
print(f"Polyak step size:")
print(f"  Final x: {final_x_polyak}")
print(f"  Final loss: {losses_polyak[-1]:.6f}")
# Plot trajectory
fig = plot_contour(X, Y, Z, trajectory_polyak,
                   title=f'SGD with Polyak Step Size (batch size={batch_size})')
# plt.savefig('polyak_trajectory.png')
plt.show()
plt.close(fig)
# Part (c)(ii): RMSProp
print("\nRunning SGD with RMSProp")
# Parameters for RMSProp
step_size_rmsprop = 0.1
beta_rmsprop = 0.9

final_x_rmsprop, losses_rmsprop, trajectory_rmsprop = rmsprop_sgd(
    initial_x, training_data, step_size_rmsprop, batch_size,
    beta=beta_rmsprop, epsilon=1e-8, max_iterations=max_iterations
)
print(f"RMSProp (step_size={step_size_rmsprop}, beta={beta_rmsprop}):")
print(f"  Final x: {final_x_rmsprop}")
print(f"  Final loss: {losses_rmsprop[-1]:.6f}")
# Plot trajectory
fig = plot_contour(X, Y, Z, trajectory_rmsprop,
                   title=f'SGD with RMSProp (batch size={batch_size})')
# plt.savefig('rmsprop_trajectory.png')
plt.show()
plt.close(fig)
# Part (c)(iii): Heavy Ball
print("\nRunning SGD with Heavy Ball")
# Parameters for Heavy Ball
step_size_hb = 0.1
beta_hb = 0.9
final_x_hb, losses_hb, trajectory_hb = heavy_ball_sgd(
    initial_x, training_data, step_size_hb, batch_size,
    beta=beta_hb, max_iterations=max_iterations
)
print(f"Heavy Ball (step_size={step_size_hb}, beta={beta_hb}):")
print(f"  Final x: {final_x_hb}")
print(f"  Final loss: {losses_hb[-1]:.6f}")
# Plot trajectory
fig = plot_contour(X, Y, Z, trajectory_hb,
                   title=f'SGD with Heavy Ball (batch size={batch_size})')
# plt.savefig('heavy_ball_trajectory.png')
plt.show()
plt.close(fig)
# Part (c)(iv): Adam
print("\nRunning SGD with Adam")
# Parameters for Adam
step_size_adam = 0.1
beta1_adam = 0.9
beta2_adam = 0.999
final_x_adam, losses_adam, trajectory_adam = adam_sgd(
    initial_x, training_data, step_size_adam, batch_size,
    beta1=beta1_adam, beta2=beta2_adam, epsilon=1e-8, max_iterations=max_iterations
)
print(f"Adam (step_size={step_size_adam}, beta1={beta1_adam}, beta2={beta2_adam}):")
print(f"  Final x: {final_x_adam}")
print(f"  Final loss: {losses_adam[-1]:.6f}")
# Plot trajectory
fig = plot_contour(X, Y, Z, trajectory_adam,
                   title=f'SGD with Adam (batch size={batch_size})')
# plt.savefig('adam_trajectory.png')
plt.show()
plt.close(fig)
# Compare convergence of all methods
plt.figure(figsize=(12, 8))
plt.plot(losses_polyak, linewidth=2, label='Polyak')
plt.plot(losses_rmsprop, linewidth=2, label='RMSProp')
plt.plot(losses_hb, linewidth=2, label='Heavy Ball')
plt.plot(losses_adam, linewidth=2, label='Adam')
```

```
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.title(f'Convergence Comparison of Advanced Optimization Methods (batch size={batch_size})')
    plt.grid(True)
    plt.legend()
    plt.show()
    # plt.savefig('advanced_methods_comparison.png')
    plt.close()
    # Compare trajectories of all methods on the same contour
    fig = plt.figure(figsize=(12, 10))
    # Create contour plot
    contour = plt.contour(X, Y, Z, 15, colors='black', alpha=0.8)
    filled_contour = plt.contourf(X, Y, Z, 100, cmap='viridis', alpha=0.6)
    plt.colorbar(filled_contour)
    # Plot trajectories
    plt.plot([t[0] for t in trajectory_polyak], [t[1] for t in trajectory_polyak], 'r.-', linewidth=1,
markersize=3, label='Polyak')
    plt.plot([t[0] for t in trajectory_rmsprop], [t[1] for t in trajectory_rmsprop], 'g.-', linewidth=1,
markersize=3, label='RMSProp')
    plt.plot([t[0] for t in trajectory_hb], [t[1] for t in trajectory_hb], 'b.-', linewidth=1, markersize=3,
label='Heavy Ball')
    plt.plot([t[0] for t in trajectory_adam], [t[1] for t in trajectory_adam], 'm.-', linewidth=1, markersize=3,
label='Adam')
    plt.plot(initial_x[0], initial_x[1], 'ko', markersize=10, label='Start Point')
    plt.legend()
    plt.xlabel('x[0]')
    plt.ylabel('x[1]')
    plt.title(f'Trajectory Comparison of Advanced Optimization Methods (batch size={batch_size})')
    plt.show()
    # plt.savefig('advanced_methods_trajectories.png')
    plt.close()
    # Check effect of batch size on advanced methods
    # We'll focus on Adam as an example
    batch_sizes = [1, 5, 10, 20, 25]
    adam_batch_results = {}
    for bs in batch_sizes:
        print(f"\nRunning Adam with batch size {bs}")
        final_x, losses, trajectory = adam_sgd(
            initial_x, training_data, step_size_adam, bs,
            beta1=beta1_adam, beta2=beta2_adam, epsilon=1e-8, max_iterations=max_iterations
        )
        adam_batch_results[bs] = (final_x, losses, trajectory)
        print(f"Batch size {bs}:")
        print(f"  Final x: {final_x}")
        print(f"  Final loss: {losses[-1]:.6f}")
    # Plot loss convergence for different batch sizes with Adam
    plt.figure(figsize=(10, 6))
    for bs, (_, losses, _) in adam_batch_results.items():
        plt.plot(losses, linewidth=2, label=f'Batch Size {bs}')

    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.title(f'Adam: Loss vs. Iterations for Different Batch Sizes')
    plt.grid(True)
    plt.legend()
    plt.show()
    # plt.savefig('adam_batch_size_comparison.png')
    plt.close()
    return {
        'polyak': (final_x_polyak, losses_polyak, trajectory_polyak),
        'rmsprop': (final_x_rmsprop, losses_rmsprop, trajectory_rmsprop),
        'heavy_ball': (final_x_hb, losses_hb, trajectory_hb),
        'adam': (final_x_adam, losses_adam, trajectory_adam),
        'adam_batch_results': adam_batch_results
    }
# Run advanced optimizer experiment
if __name__ == "__main__":
    results = run_advanced_optimizer_experiment()
# %%
```