**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 x₀.
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:
5. Update the parameters with step size
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.

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

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.

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.

**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.

**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

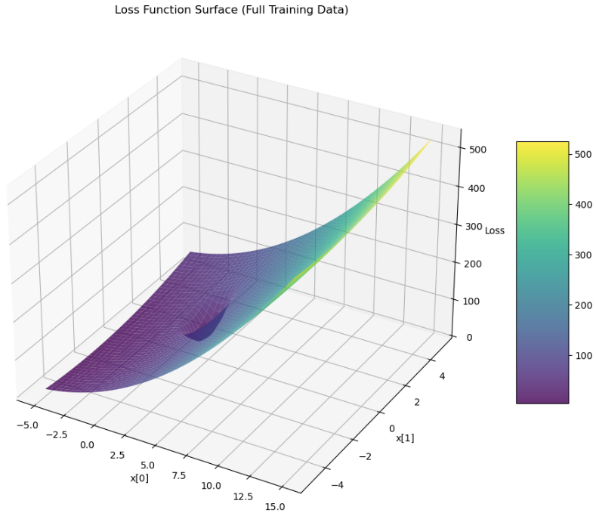
f ([10, 0]) = 294.9125

f ([10, 5]) = 340.5904

f ([0, 5]) = 99.6915

f ([5, 5]) = 195.1409

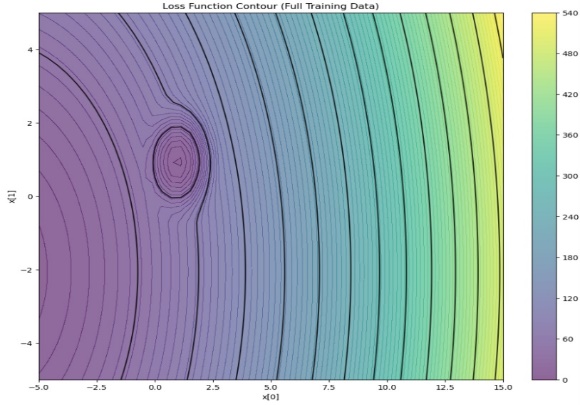
Based on these values, I select the interval **x₀ ∈ [-5, 15] and x₁ ∈ [-5, 5]** for plotting the minimum and the salient features of the loss surface.

**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 x₀ 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.

***Figure 1, 3D Wireframe Plot of Loss Function***

**Loss function contour**

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.

***Figure 2, Contour Plot for Loss Function***

**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 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.

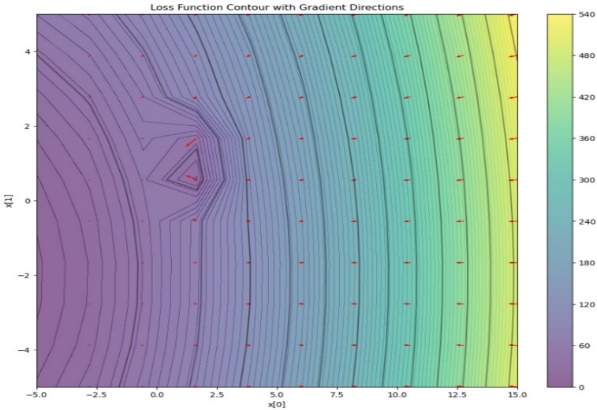
**Hessian Analysis at [9,0]**

[[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:

where is the step size (learning rate) and is the gradient of the loss function with respect to the current point

***Step Size Selection and Consequences***

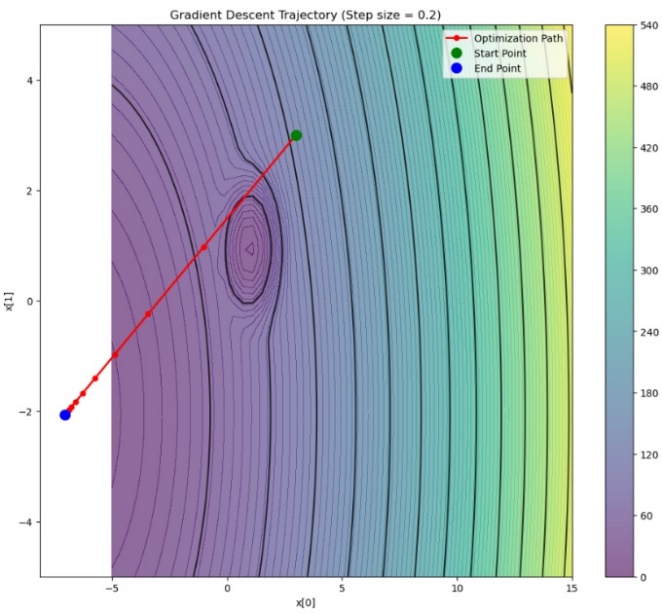
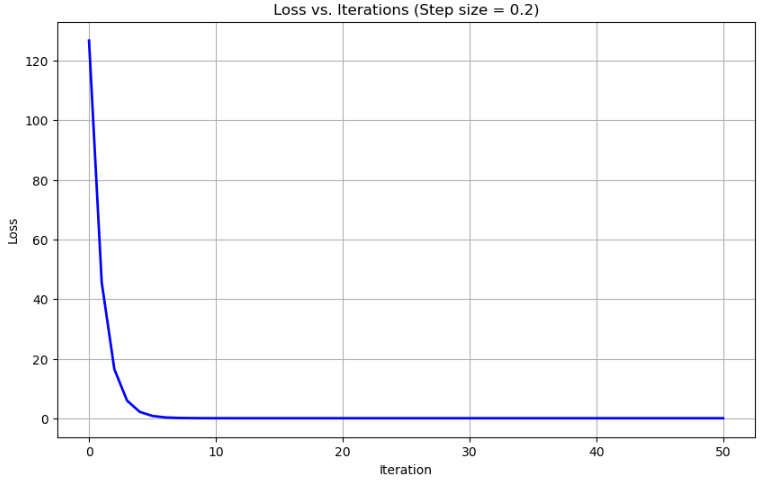
On experimenting with different step sizes (0.01, 0.05, 0.1, 0.2, 0.5) and used the one with the most appropriate trade-off between convergence speed and stability. The results after 50 iterations are shown in Table 2 below

|  |  |  |  |
| --- | --- | --- | --- |
| **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.

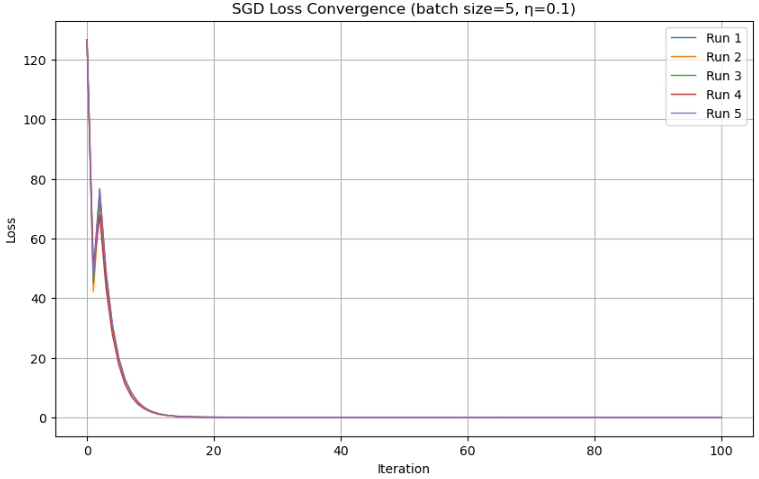
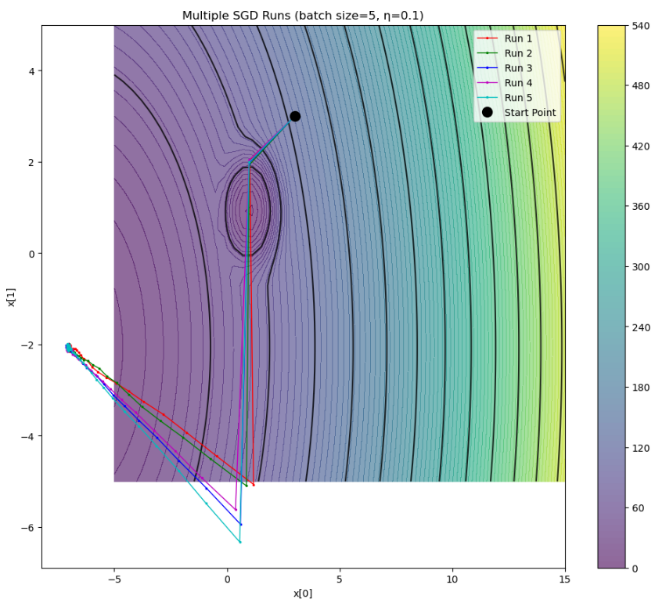
**ii) Mini Batch SGD with Fixed Size**

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

|  |  |  |
| --- | --- | --- |
| **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 |

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).

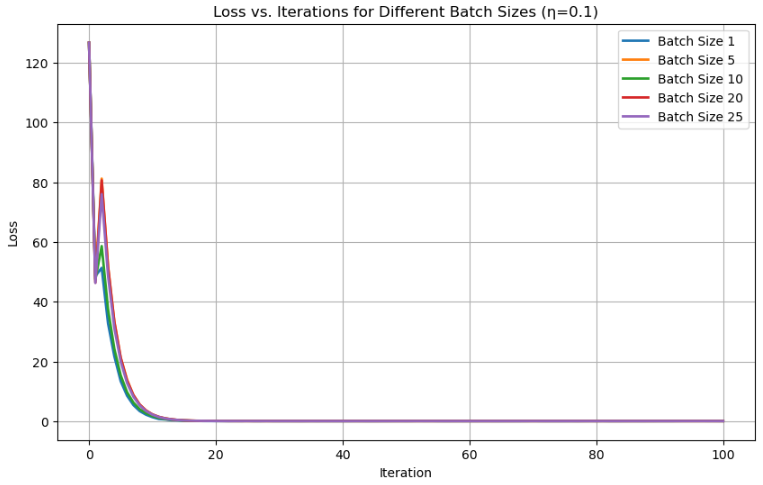
***Table 4, Summary of Results of Different Batch Sizes***

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.

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

**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 x₁ 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)

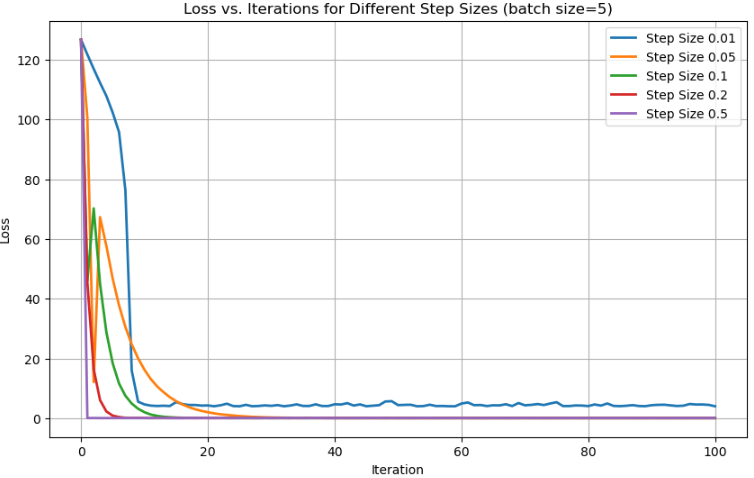
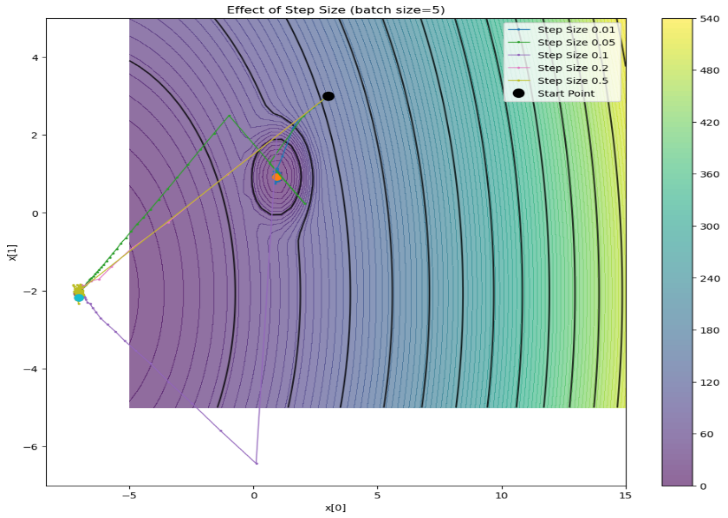
* Have regularization-type properties and could help promote machine learning task generalization

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**

|  |  |  |
| --- | --- | --- |
| **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 |

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.

*****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

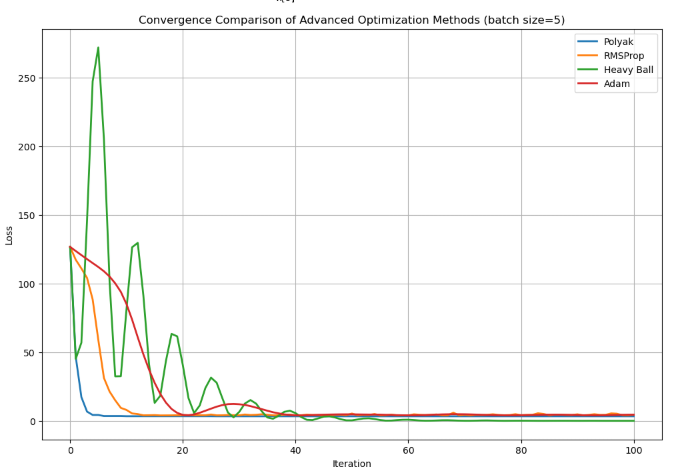
**Comparison with Batch Size Implications**

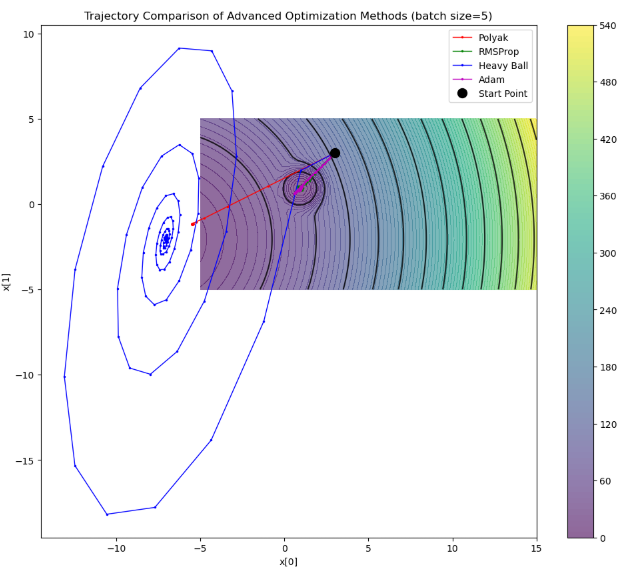
Comparing the effect of step size (Figure 9 and Figure 10) with batch size effects (Figure 8):

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.

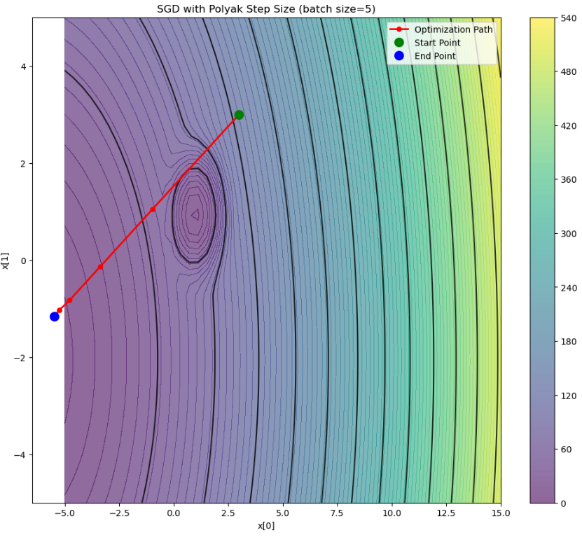
**Insights:** Step-size selection is superior to batch-size selection for converging towards the global optimum within this optimization context. The step size must be wide enough not to get caught into poor local optima but not too wide relative to stable convergence.

**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**

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

To run, I utilized the estimate (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:

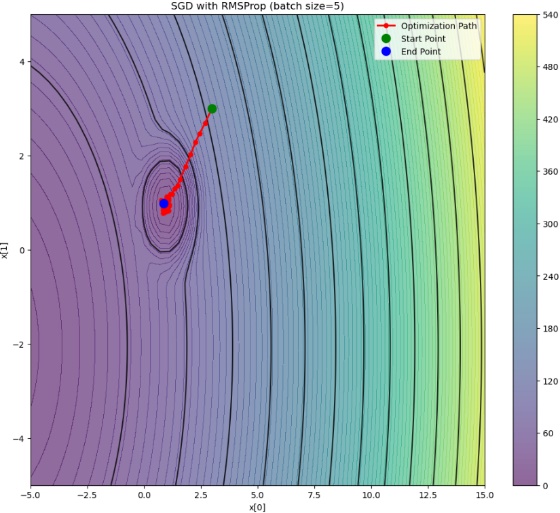
* Follows a relatively direct path towards a low level.
* It converges quickly with minimal oscillations
* It does converge towards the middle minimum and not the global minimum as with the fixed step sizes.

***Fig 11, SGD with Polyak Step Size***

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:

******Selected parameters:

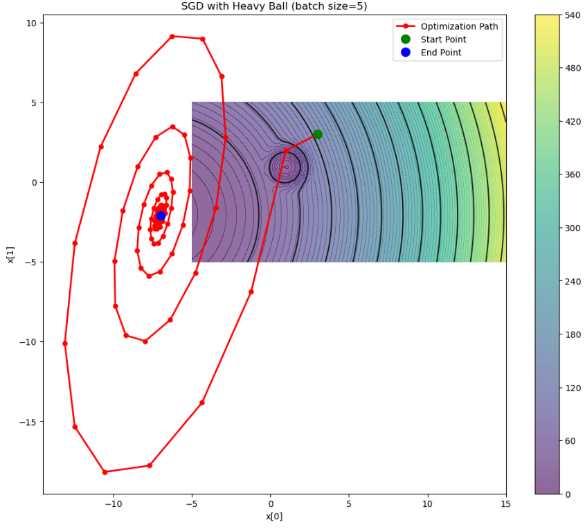
* Step size (η): 0.1 (as the performance was satisfactory in part b)
* Decay rate (β): 0.9 (default value utilized for balancing recent and previous gradients)
* ε: 10⁻⁸ (a very small number to prevent division by

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.

***Fig 12, SGD with RMSProp***

**(iii) Heavy Ball (Momentum)** The Heavy Ball method uses momentum to accelerate the convergence:

Selected parameters:

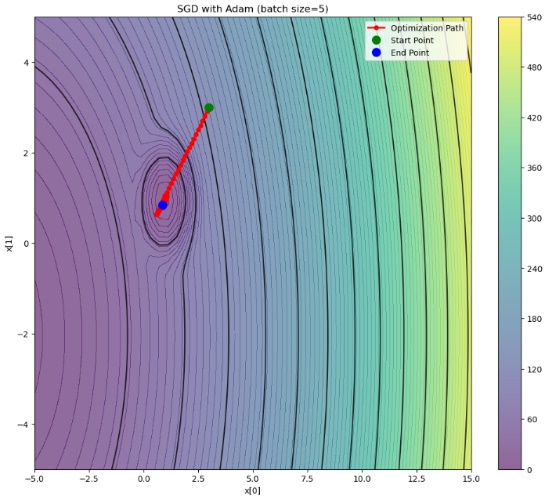
* Step size (η): 0.1 (same as with other methods)
* Beta (momentum coefficient): 0.9 (Default value with good acceleration with no oscillation)

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.

***Fig 13, SGD with Heavy Ball***

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**

Adam combines momentum and adaptive learning rates:

Selected Standard Parameters:

* Step size (η): 0.1
* (for momentum)
* (for squared gradients)
* :

***Fig 14, SGD with Adam***

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**

|  |  |  |
| --- | --- | --- |
| **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 |

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

***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()

# %%