**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

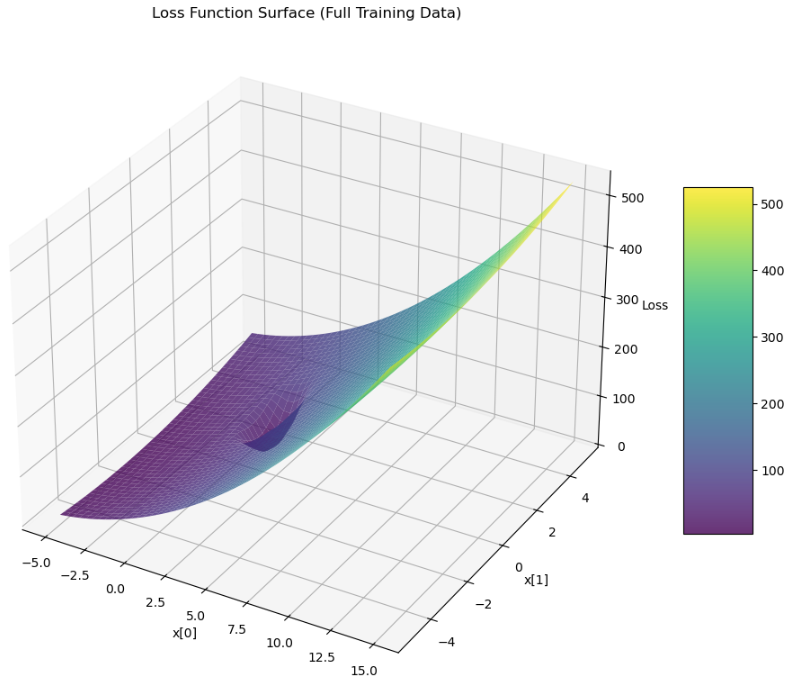
f ([10, 5]) = 340.5904

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.

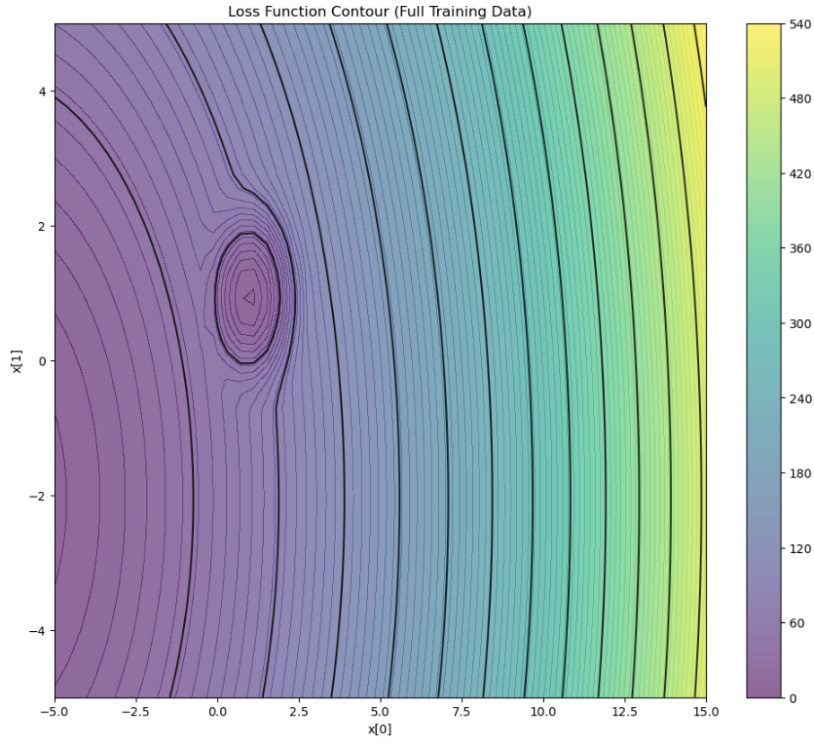
**Loss Function Surface**

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.

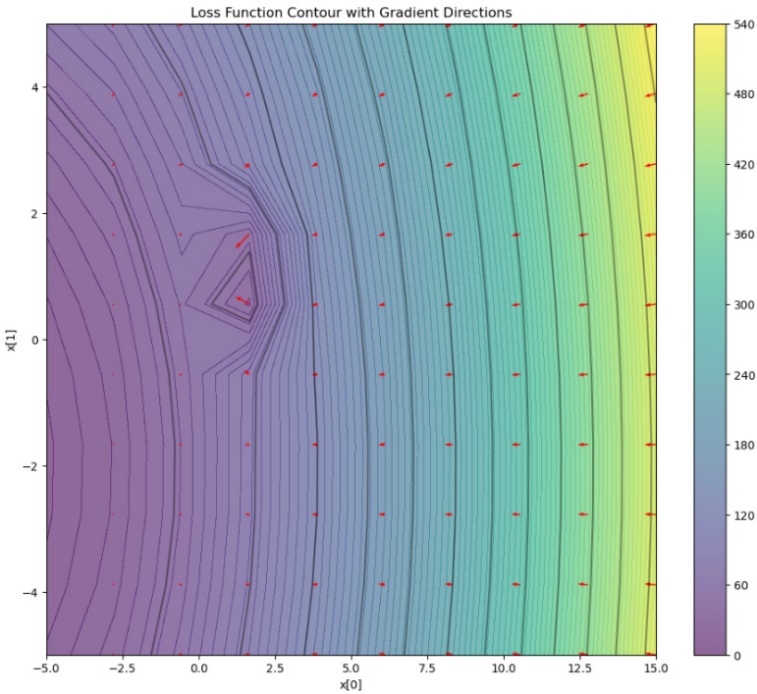
**Hessian Analysis at [9,0]**

[[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**

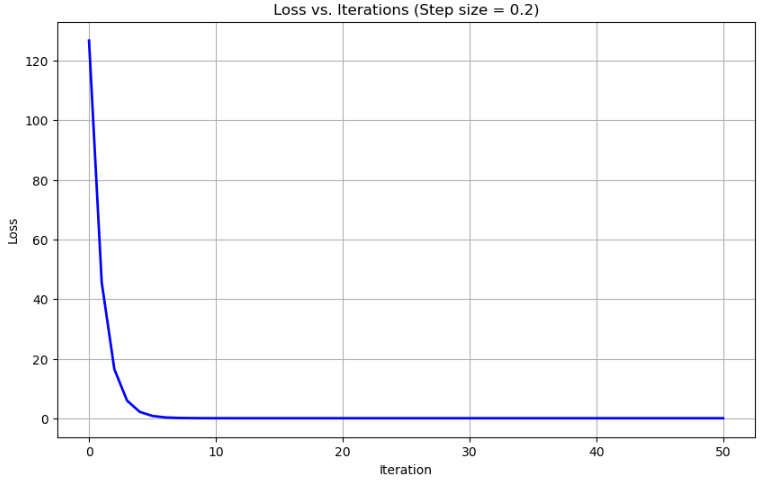
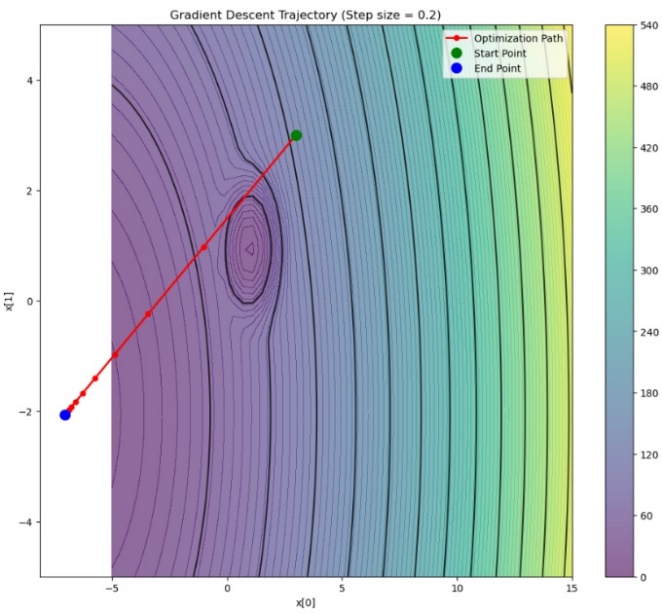
|  |  |  |  |
| --- | --- | --- | --- |
| **Step Size** | **Final Position** | **Final Loss** | **Loss Reduction** |
| 0.01 | [0.955, 0.932] | 4.048 | 96.80% |
| 0.05 | [-6.985, -2.059] | 0.110 | 99.91% |
| 0.1 | [-7.045, -2.068] | 0.107 | 99.92% |
| 0.2 | [-7.045, -2.068] | 0.107 | 99.92% |
| 0.5 |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

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

***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.

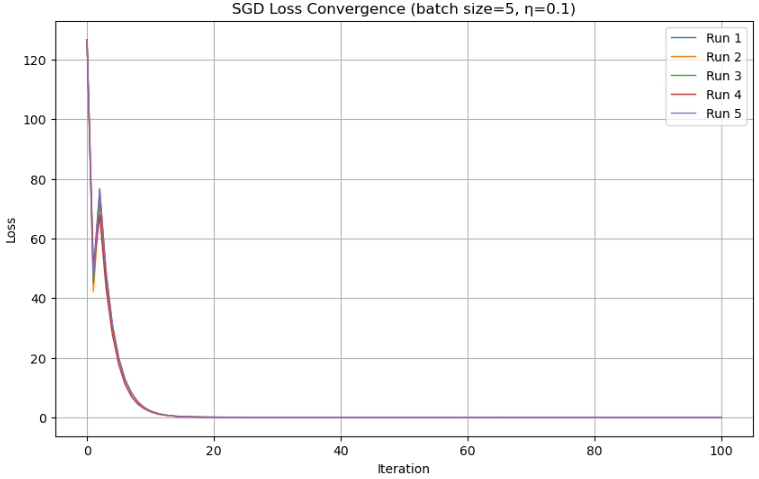
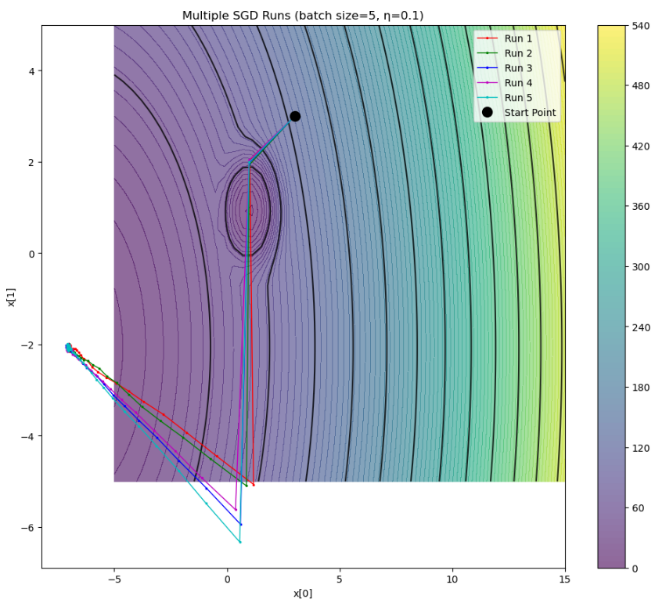
A step size too small will lead the convergence towards worse local optima, but the appropriate step size will enable the algorithm to converge towards the global optimum within time.

**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.

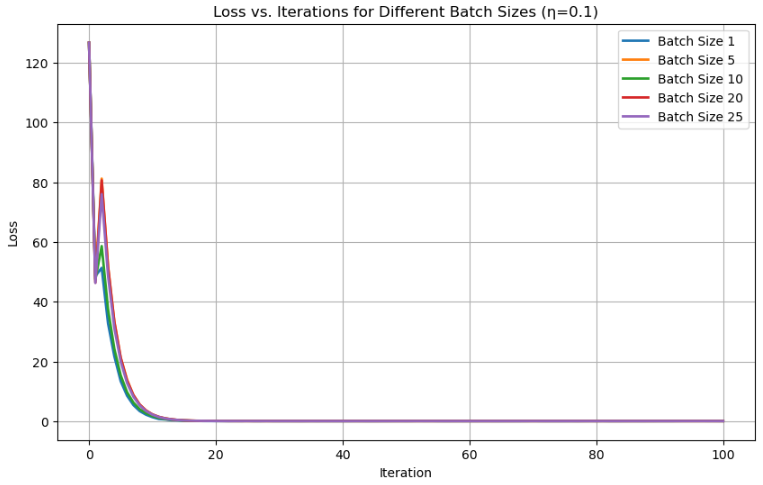
Mini-batch SGD does not lose the ability to find the global minimum with the introduction of useful stochasticity capable of escaping local minimum for more difficult topographies.

**(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, Loss Vs Iteration for Different Batch Sizes = 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 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)

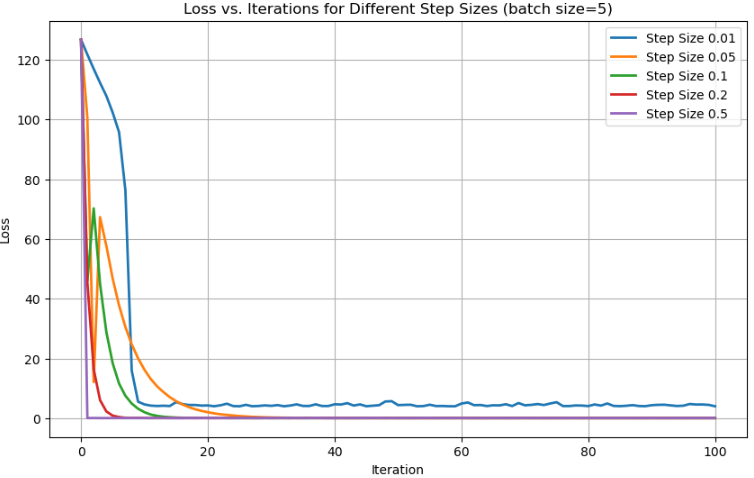
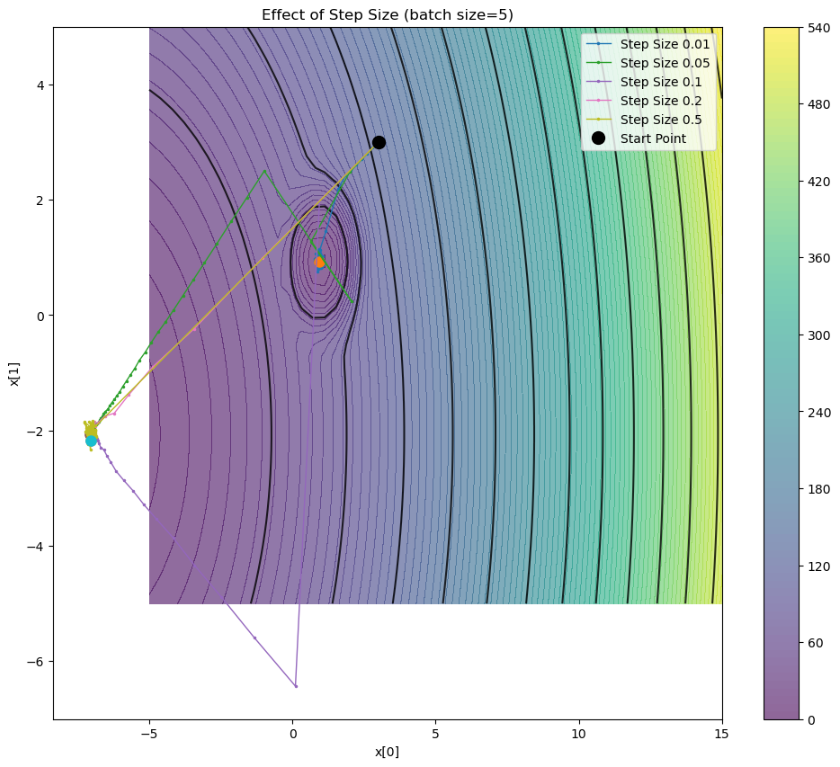
* Have regularization-type properties and could help promote machine learning task generalization

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**

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:

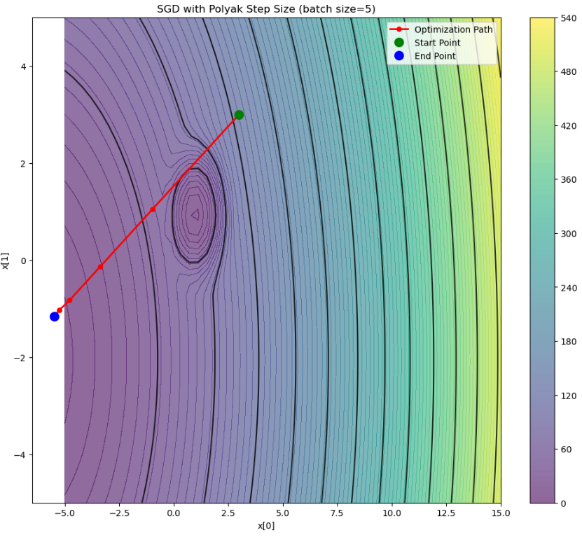
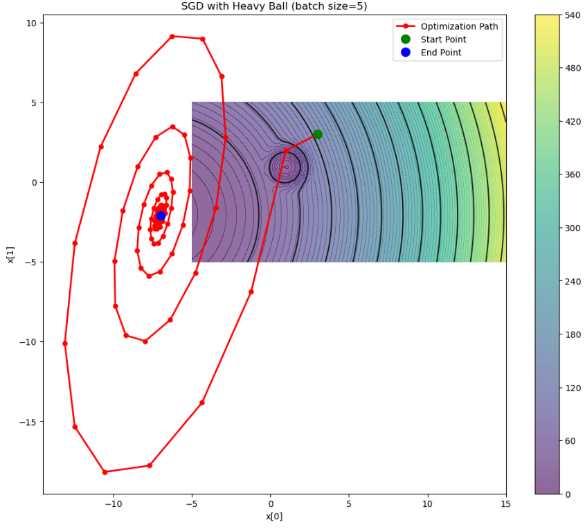
* With the smallest step size (0.01), it converges to a different path towards a local optimum around [0.94, 0.93]
* Medium step sizes follow direct routes towards the global minimum region
* Larger step sizes are associated with larger oscillatory motion around the minimum

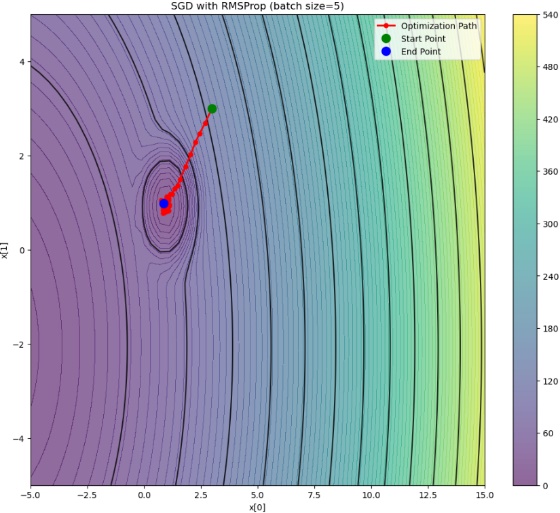
**Comparison with Batch Size Implications**

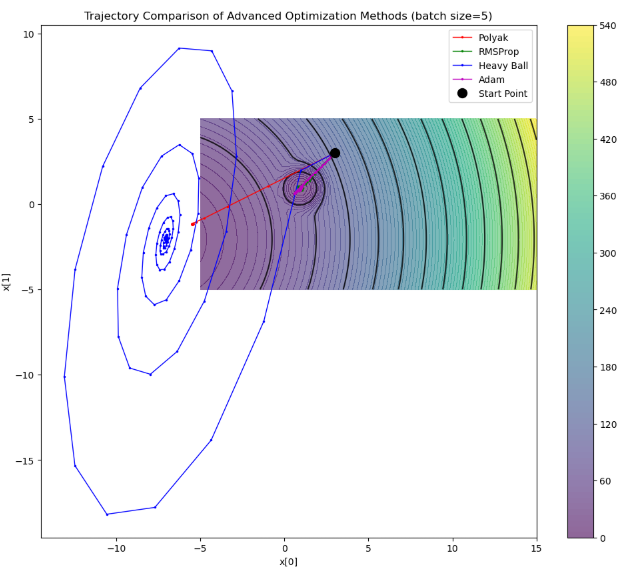
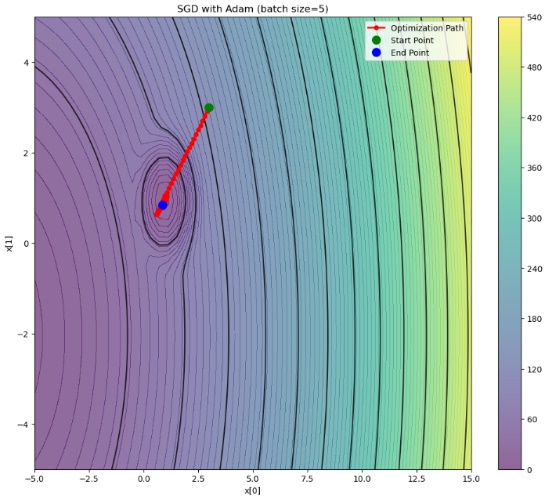
Comparing the effect of step size (Figure 9 and Figure 10) with batch size effects (Figure 8):

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.

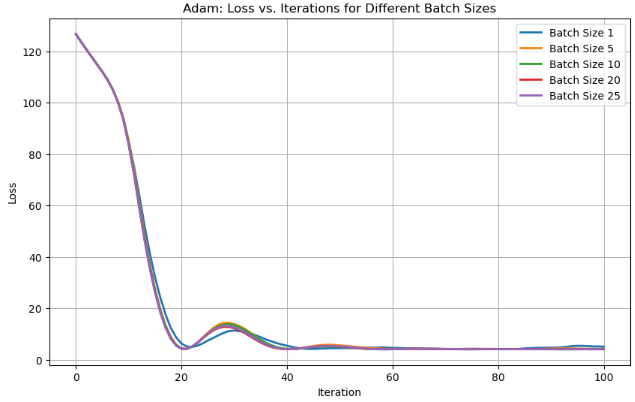
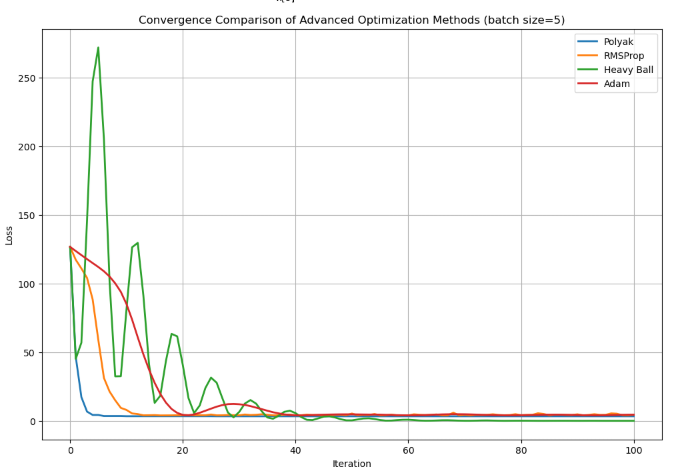
**Insights:** Step-size selection is superior to batch-size selection for converging towards the global optimum within this optimization context. The step size must be wide enough not to get caught into poor local optima but not too wide relative to stable convergence.

**PART C**

***Fig 11, SGD with Polyak Step Size Fig 12, SGD with RMSProp Fig 13, SGD with Heavy Ball***



***Fig 14, SGD with Adam Fig 15, Trajectory Comparison of Advanced Optimization Methods***

***Fig 16, Convergence Comparison of Optimization Methods Fig 17, Loss Vs Iteration (Adam)***

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.

**(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.

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.

(iii) Heavy Ball (Momentum)

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 spiralling 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**

Adam combines momentum and adaptive learning rates:

Selected Standard Parameters:

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

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 it to drop 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.