Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Week 6 Assignment
CS7DS2 Optimisation for Machine Learning

Ujjayant Kadian (22330954)

March 13, 2025

Introduction

This report addresses the objectives of the Week 6 optimisation assignment. In summary, we:

- Implement mini-batch Stochastic Gradient Descent (SGD) with support for various step-size strategies: constant step, Polyak's adaptive step, RMSProp, Heavy Ball momentum, and Adam.
- Apply this optimiser to a given two-dimensional loss function, visualising the loss surface and computing its gradient for analysis.
- Experimentally compare full-batch gradient descent with mini-batch SGD (batch size 5) using a constant step size, examining convergence behavior, run-to-run variability, and the effects of changing the batch size and step size.
- Evaluate advanced step-size strategies (Polyak, RMSProp, Heavy Ball, Adam) in mini-batch SGD, explaining parameter choices and comparing their performance and sensitivity to batch size.

(a) (i) SGD Implementation

SGD Algorithm: SGD iteratively selects a random subset (mini-batch) from the training data, calculates the loss gradient with respect to parameters using this mini-batch, and updates parameters in the direction opposite to the gradient.
Our implementation allows for dynamic step-size adjustment through a base class method that selects the update rule based on the specified strategy (constant or adaptive).
The pseudocode for a single iteration is as follows:

1. select mini-batch N;
2. calculate gradient $g = \nabla f(x, N)$;
3. determine step = $\alpha(g)$ based on the method;
4. update x to $x - step$.

This process is repeated until convergence or the maximum number of iterations is reached.

Step-Size Variations: We implemented five schemes for α (a recap of week-4 assignment):

- **Constant:** uses a fixed learning rate α (tuned manually). The update is $x_{new} = x - \alpha \cdot g$. This simple scheme requires α small enough for stability yet large enough for speed.

- **Polyak:** uses knowledge of the optimal loss f^* (here 0, since the loss is non-negative and zero at the minimum - more about this in the next section). The step size at iteration n is $\alpha_n = \frac{f(x_n)-f^*}{|\nabla f(x_n)|^2}$. This choice aims to take the largest step that will (in theory) drop the loss to f^* in one move. In code, we implemented:

```
if np.linalg.norm(g)**2 > 1e-12:
alpha = (f_val - f_star) / (np.linalg.norm(g)**2)
step = alpha * g
```

  (with safeguards to avoid division by zero). Polyak's method can greatly accelerate convergence when f^* is known and the gradient is accurate, but if f^* is mis-specified or using a noisy mini-batch gradient, it may overshoot. We set $f^* = 0$ (global minimum of our loss - see next section) in our experiments.

- **RMSProp:** uses a decaying average of past squared gradients to adjust learning rate per coordinate. We accumulate $E[g^2] \leftarrow \rho E[g^2] + (1-\rho)g^2$ (with decay $\rho = 0.9$) and divide the step by $\sqrt{E[g^2]}$. This moderates the step size for coordinates with consistently large gradients, preventing oscillations. RMSProp thus adapts learning rates: frequently large-gradient directions get smaller effective α.

- **Heavy Ball (Momentum):** adds a momentum term β $(0 < \beta < 1)$ to smooth updates. We use $v \leftarrow \beta v - \alpha g$ and update $x \leftarrow x + v$. The velocity v accumulates past gradients, giving inertia to the motion. This helps push through shallow minima or noise, but if α or β are too high it can overshoot. Our default $\beta = 0.9$, but we found we had to reduce β in some cases to maintain stability (see Results).

- **Adam:** combines momentum and RMSProp - it maintains both first (m) and second (v) moment estimates of gradients. We update $m \leftarrow \beta_1 m + (1-\beta_1)g$, $v \leftarrow \beta_2 v + (1-\beta_2)g^2$, then use bias-corrected estimates $\hat{m}, \hat{v}$ to compute the step $-\alpha, \hat{m}/(\sqrt{\hat{v}} + \epsilon)$. Adam is known for being relatively robust to hyperparameter choices and noise. We used default $\beta_1 = 0.9$, $\beta_2 = 0.999$.

All these methods were integrated into a single SGD routine. The code uses a class `SGDOptimizer` that stores any state (like momentum velocity or Adam moments) and provides a `compute_step(grad)` method to get the update step for each iteration. This design allowed easy switching of optimisation strategy.

(a) (ii) Loss Function and Visualisation

Loss Function Definition: The loss function $f(x, N)$ is defined as the average over points $w \in N$ of a piecewise-quadratic loss. For each data point w, $z = x - w - (1,1)$ is computed and the loss contribution is $\min 20|(z)|^2, ; |(z + [9,10])|^2$ (so either a scaled quadratic centered at $w + (1,1)$ or another basin centered at $w + (-8,-9)$).
Intuitively, each data point defines a primary bowl-shaped loss around $w + (1,1)$ (steep curvature) and a secondary bowl offset by (-9,-10) (wider).
Visualisation: Using the full training set T, we plotted the surface and contours of $f(x, T)$. Figure 1 shows the wireframe 3D surface and contour plot for the full dataset. We chose the range $x_1, x_2 \in [-5, 5]$ since the training points are near $(0,0)$ and we expected the minimum around $(1,1)$ (because $w + (1,1)$ would be near $(1,1)$ for $w \approx (0,0)$). This range captures the main bowl and any surrounding rise.
The surface appears as a single bowl with a unique minimum at $x \approx (1,1)$. The contour plot confirms concentric level sets around $(1,1)$ and no other local minima in this region. The function rises more steeply in the north-east direction, indicating some anisotropy (due to the piecewise definition). Overall, $f(x, T)$ is convex in the plotted region and well-behaved for gradient methods.

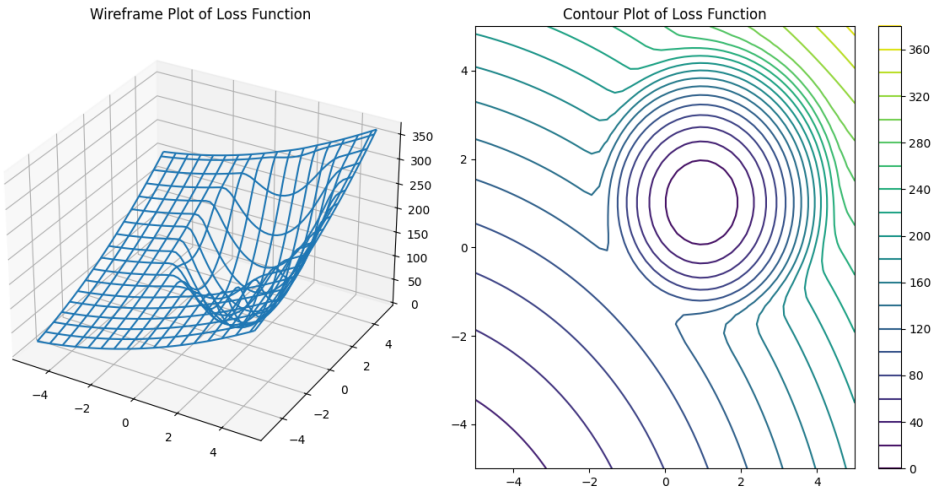

Figure 1: Loss function surface (wireframe) and contours for full data.

(a) (iii) Gradient Calculation

The loss's analytical gradient can be derived piecewise, but we opted to compute it numerically using central finite differences for generality. This approach is straightforward to implement and avoids manual differentiation of the piecewise function. We wrote a helper `finite_diff_grad(f, x, eps=1e-6)` that displaces each dimension by a small ϵ and uses $\frac{f(x+\epsilon_i)-f(x-\epsilon_i)}{2\epsilon}$ to approximate the ith partial derivative:

```
def finite_diff_grad(f, x, eps=1e-6):
grad = np.zeros_like(x)
for i in range(len(x)):
    x_plus = x.copy();   x_plus[i] += eps
    x_minus = x.copy();  x_minus[i] -= eps
    grad[i] = (f(x_plus) - f(x_minus)) / (2*eps)
return grad
```

This computes $\frac{\partial f}{\partial x_1}$ and $\frac{\partial f}{\partial x_2}$ independently by symmetric differences. We verified the implementation at a sample point (printing the gradient at $[1.0, 1.0]$) to ensure it produced sensible values. Using central differences provides $O(\epsilon^2)$ accuracy, which is more stable than forward differences for a given ϵ. The numerical gradient was used for all gradient-descent algorithms. In each SGD iteration, we passed $f(x, \text{mini\_batch})$ into this function to get an approximate $\nabla f(x, N)$ for the current mini-batch.

We acknowledge that finite differences add computational overhead (requiring 2 evaluations per dimension per step) and potential numerical error, but with only 2 dimensions and our chosen ϵ, it was very manageable and sufficiently accurate for the smooth pieces of the loss. (At the non-differentiable junction between the two quadratic pieces, the gradient is undefined; our random initializations and limited domain meant we did not exactly hit those junctions, and the numerical gradient smoothly approximates the subgradient in those regions.)

(b) (i) Constant Step Size Gradient Descent

First, we ran standard gradient descent using the full dataset $(batch = T)$ and a constant step size. Starting at $x_0 = (3, 3)$, we experimented with step sizes and chose $\alpha = 0.01$ as a good trade-off: larger values (e.g. 0.05) caused oscillations, while much smaller values (0.005) were very slow to converge. With $\alpha = 0.01$, GD rapidly decreased the loss. The loss fell from ~ 160 at $(3, 3)$ to near 0 within ~ 5 iterations, flattening out as it approached the minimum. The trajectory in the contour plot (Figure 3) shows the path of x over iterations: it moves nearly straight toward the minimum at $(1, 1)$. In early steps the algorithm covers large distance, then steps get smaller as the gradient diminishes near the optimum (points cluster near the end). The smooth, monotonic decrease of $f(x)$ without overshooting indicates our α choice was appropriate. In fact, the loss dropped most in the first few iterations and by iteration ~ 5 it reached a plateau near the minimum, consistent with gradient norms approaching zero.

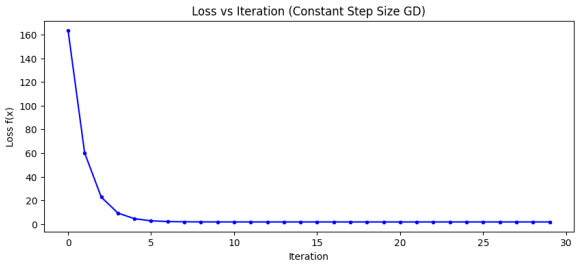

Figure 2: Loss vs iterations

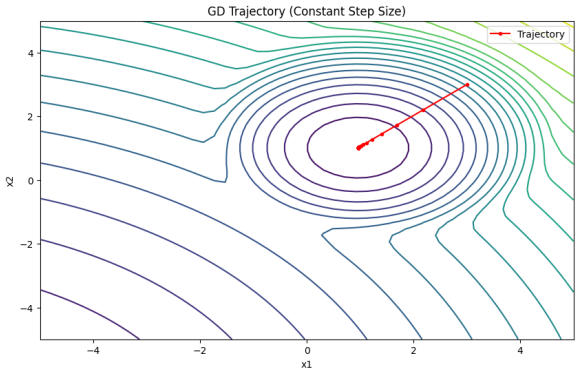

Figure 3: Gradient descent trajectory

We see a smooth and fast convergence to the center of contours at (1,1). The loss curve (left) decreases sharply initially and levels off as it nears zero, with no oscillations – evidence of a well-chosen step size. The contour trajectory (right) is almost a straight line towards the minimum, as the full-batch gradient consistently points directly toward (1,1) in this symmetric scenario.

(b) (ii) Mini-Batch SGD with Constant Step Size

Next, we switched to mini-batch SGD with batch size 5 (out of 25 total data points) to observe stochastic effects. We kept $\alpha = 0.01$ for fair comparison. Because of the random sampling of mini-batches, each run of SGD follows a different path. We ran the mini-batch SGD five times and plotted the loss trajectories and parameter paths. The loss vs. iteration curves (Figure 4, left) show a general downward trend but with noisy fluctuations due to using only 5 points per update. All runs eventually converge near $f \approx 0$, but compared to full-batch GD, the convergence is less smooth, reflecting variance in gradient estimates. There is run-to-run variability; for example, one run might reach near-minimum by 15 iterations while another still oscillates slightly around a slightly higher loss at the same time.

On the contour plot of trajectories (Figure 5, right), we see five slightly different zig-zag paths to the minimum. Unlike the smooth straight path in full-batch GD, the mini-batch paths wander as the gradient is perturbed by sampling noise. Each still converges to the vicinity of (1,1), but the exact endpoint varies a bit between runs (within a small neighborhood). This spread in final x is because the noise can push the parameters around the flat region near the minimum - an instance of SGD potentially exploring a wider minima. We did 200 iterations for mini-batch SGD (compared to $\sim$30 needed for full-batch) to allow the stochastic updates to settle; individual updates are cheaper, but more are needed to average out the noise and converge. The greater number of points in the SGD paths confirms this (they took more steps).

Importantly, across runs, SGD does not diverge - the noise causes oscillations but the overall trend is still toward the minimum. The variability from run to run is evident in the early iterations (some runs drop faster initially than others), but over time all runs hover near $f \approx 0$. Compared to full-batch GD, mini-batch SGD's progress is less stable (loss curves show jitters) and slower in terms of iterations, but eventually reaches a similar outcome.

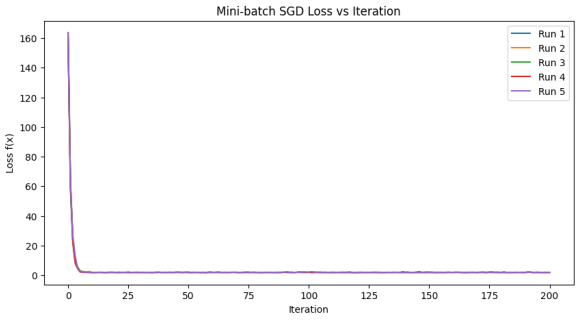

Figure 4: Mini-batch SGD loss vs iterations

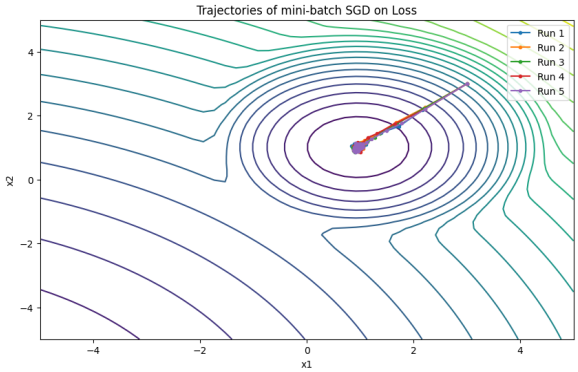

Figure 5: Mini-batch SGD trajectory

The stochastic nature is evident - loss curves descend with small random fluctuations and occasional plateaus, unlike the smooth GD curve. Trajectories (right) wander with small zig-zags instead of a direct line.

All runs converge to the same vicinity near (1,1) (indicated by clustering of trajectory endpoints) and achieve similar final losses, but the paths taken differ due to different mini-batch sequences.

(b) (iii) Effect of Batch Size

We investigated how varying the mini-batch size affects convergence (using constant $\alpha = 0.01$). We tried batch sizes 1 (pure SGD), 5, 10, and 25 (full batch). We found that smaller batches lead to noisier gradients and thus more erratic paths, whereas larger batches yield more stable descent. With batch size 1, the loss curve had the highest variance - it jumps up and down considerably, though still trending downward overall. Increasing to batch 5 or 10 smooths out some fluctuations (batch 10 was noticeably more stable than 1 or 5). Batch 25 (full batch) gave the smoothest curve (essentially identical to deterministic GD). However, interestingly, all batch sizes converged to roughly the same final point (around (1,1)), just at different rates and with different amounts of noise. Smaller batches converged in fewer iterations (they make more updates for a given number of data passes) but with high variance each step, whereas larger batches took more iterations but each step was more reliable. This aligns with theory: batch-1 SGD has highest gradient noise which can help escape shallow local minima but here there was only one global minimum. We did observe that the final x for smaller batches had a bit more spread between runs - e.g. batch 1 runs might end up slightly different distances from (1,1) depending on the random noise, whereas batch 25 runs all end at the same point each time (since it's deterministic). This suggests that SGD noise can cause a sort of "random walk" near the optimum, potentially landing in different points in a flat region ("wider" minima). In our case the effect was minor - all results were very close to the true minimum - but in more complex loss landscapes this could translate to SGD preferring wider minima that might generalize better (an often-cited regularization effect of SGD noise). In summary, noise from small batches does not prevent convergence to the optimum here; it only injects oscillations. There was no evidence that SGD was getting stuck in a higher-loss region - just that it rattled around (and slightly beyond) the optimum before settling. This mild implicit regularization might be beneficial in preventing overfitting in other contexts, but for this simple convex problem, batch size mainly impacts convergence speed and smoothness rather than the final value reached.

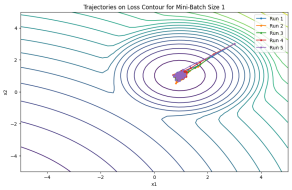

Figure 6: Batch size 1

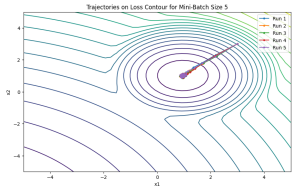

Figure 7: Batch size 5

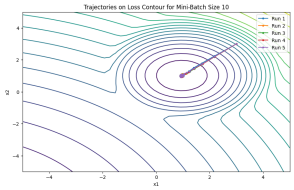

Figure 8: Batch size 10

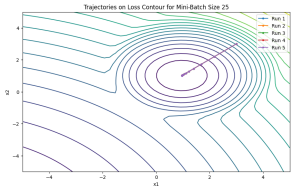

Figure 9: Batch size 25

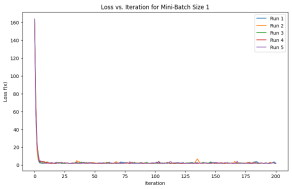

Figure 10: Loss: Batch 1

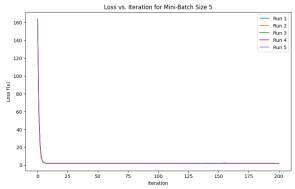

Figure 11: Loss: Batch 5

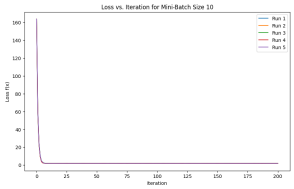

Figure 12: Loss: Batch 10

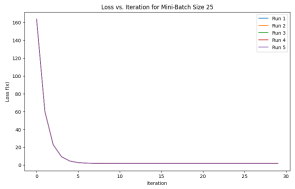

Figure 13: Loss: Batch 25

(b) (iv) Effect of Step Size

Finally, we kept batch size = 5 and varied the constant step size α. We tried $\alpha = 0.001$ (very small), 0.005, 0.01 (baseline), 0.05, and 0.1 (quite large). Consistent with expectations, a very small step (0.001) yielded an extremely slow but steady descent - the loss barely decreased over the same number of iterations, undershooting significantly. Medium steps (0.005 and 0.01) performed well, with 0.01 being fastest without instability. At $\alpha = 0.05$, initial descent was faster, but we noticed larger oscillations; it "appeared" to converge in terms of moving average of loss, but there was noticeable jitter and run-to-run variability increased. At $\alpha = 0.1$, the effect was pronounced: the loss would often sharply decrease then increase, bouncing around near the minimum. Interestingly, even with $\alpha = 0.1$, the loss values on the mini-batch still often reached near-zero (because with such a large step, the algorithm can overshoot and occasionally land on a very low loss for a particular mini-batch), but the trajectory of x did not settle. In fact, for $\alpha = 0.1$ the parameter values kept wandering - the contour trajectories showed the algorithm jumping around the optimum rather than staying near (1,1). This is a crucial observation: an overly large step size can make the loss (evaluated on each mini-batch) look low, while the parameters are actually diverging from the true optimum. In our experiments,

$\alpha = 0.1$ caused x to bounce in a radius around (1,1) without converging - effectively the algorithm didn't "settle" even though each batch occasionally gave a low loss. Reducing α fixed this. This highlights that one must monitor more than just instantaneous loss - a large α with noisy gradients can yield deceptively low batch losses but isn't minimizing the true objective. Overall, α needs to be below a certain threshold (for us, ~ 0.05) for convergence; otherwise SGD enters a regime of persistent oscillation. Smaller α always eventually converged (just slowly), whereas too large α failed to converge in the iteration limit. This comparison reinforced the earlier choice of 0.01 as balanced - it was near the largest that still converged stably.

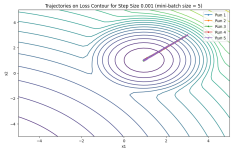

Figure 14: Step size 0.001

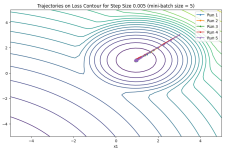

Figure 15: Step size 0.005

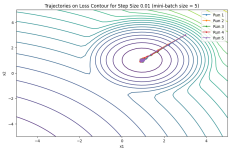

Figure 16: Step size 0.01

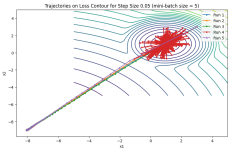

Figure 17: Step size 0.05

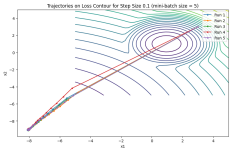

Figure 18: Step size 0.1

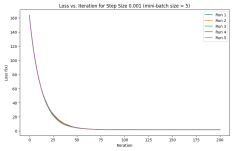

Figure 19: Loss: Step 0.001

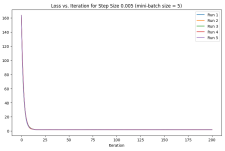

Figure 20: Loss: Step 0.005

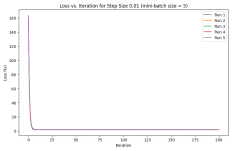

Figure 21: Loss: Step 0.01

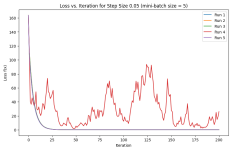

Figure 22: Loss: Step 0.05

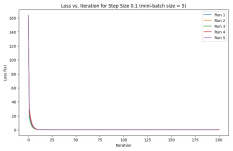

Figure 23: Loss: Step 0.1

(c) Comparision of Optimisation Methods

In this section, we present a consolidated analysis of four step-size adaptation methods: Polyak Step Size, RMSProp, Heavy Ball Momentum, and Adam, integrating parameter choice explanations, effects on loss over time, trajectory behavior, and sensitivity to mini-batch size into four broad sections.

Polyak Step-Size Method

Polyak step size dynamically adjusts the learning rate based on the current function value relative to an estimated optimal loss ($f_{opt} = 0$ - the minimum loss possible here), ensuring theoretically optimal convergence in deterministic settings. However, direct application without stabilizers ($\epsilon = 10^{-6}$, $\gamma = 0.001$) led to instability due to excessive step sizes. This adaptation caused slower convergence than well-tuned constant step-size GD, as the method cautiously adjusted step sizes instead of maintaining consistent updates. Loss decreased steadily over time, showing stable yet slow convergence, making it the least aggressive optimizer in this comparison. The trajectory of parameter updates was significantly shorter, indicating gradual movement towards the minimum without overshooting, which is advantageous for stability but inefficient in well-conditioned problems. Unlike other methods, Polyak was unaffected by mini-batch size, as its adjustments depended on function values rather than gradient variations, meaning convergence characteristics remained the same regardless of batch size. While theoretically promising, Polyak's slower progress, requirement of an accurate f_{opt}, and susceptibility to noisy gradient estimates made it less effective in this mini-batch setting compared to RMSProp and Adam.

RMSProp

RMSProp maintains a running average of squared gradients, allowing it to normalize step sizes and prevent excessive updates in directions with consistently high gradients. With a decay rate ($\rho = 0.9$) and an adaptive learning rate ($\alpha = 0.1$), this method exhibited fast initial convergence, reaching near-optimal loss within 50 iterations while maintaining stability. Unlike Polyak, RMSProp effectively mitigated noisy gradient variations by scaling updates based on recent gradient magnitudes, preventing erratic parameter movements. The trajectory of updates showed an efficient and direct path to the minimum, albeit with minor fluctuations due to mini-batch noise. Increasing batch size led to smoother and slightly faster convergence, as noise in gradient estimates decreased, further stabilizing updates. Compared to constant step-size GD, RMSProp performed similarly in terms of total iterations but provided greater robustness to parameter tuning, making it particularly useful when training involves inconsistent gradient magnitudes or poorly scaled features. However, given that our loss function was relatively well-behaved and had a clear minimum, the normalization did not significantly outperform well-tuned GD, meaning its main advantage was in its consistency across different conditions rather than raw speed.

Heavy Ball Momentum

Heavy Ball momentum incorporates past gradients into updates using a velocity term to accelerate convergence in consistent gradient directions. Despite extensive tuning, finding stable parameters proved difficult, as initial rapid progress was frequently followed by severe divergence after ~ 100 iterations. With an initial learning rate ($\alpha = 0.1$) and an extremely low momentum coefficient (momentum=0.001), we attempted to minimize overshooting, but even at reduced momentum, instability persisted. Loss initially decreased comparably to RMSProp and Adam but began increasing uncontrollably due to accumulated momentum, demonstrating the method's sensitivity to even minor parameter misconfigurations. The trajectory revealed erratic, large jumps instead of a smooth descent, indicating excessive step sizes due to momentum accumulation. Unlike RMSProp and Adam, which adjusted step sizes dynamically, Heavy Ball momentum continued applying updates based on previous velocities, making it more vulnerable to stochastic gradient noise, particularly in mini-batch settings. Adjusting batch size did not resolve instability issues, as divergence persisted regardless of gradient noise reduction. Compared to other methods, Heavy Ball was the least reliable and was ultimately unsuitable for this problem, as its tendency to overshoot outweighed any potential acceleration benefits.

Adam

Adam combines the benefits of momentum (like Heavy Ball) and adaptive step-size scaling (like RMSProp), offering a robust and efficient optimizer with minimal parameter tuning. Using standard parameters ($\alpha = 0.1$, $\beta_1 = 0.9$, $\beta_2 = 0.999$), Adam demonstrated fast and stable convergence, similar to RMSProp but with reduced sensitivity to batch-to-batch gradient noise. Loss dropped rapidly within 50 iterations, with minor overshooting between 25-50 iterations, stabilizing thereafter, making it highly efficient. The trajectory showed a direct and smooth path to the minimum, outperforming RMSProp slightly in terms of consistency. Mini-batch noise had less impact on Adam than RMSProp, as its momentum terms smoothed out stochastic fluctuations, ensuring steady descent even with smaller batch sizes. Unlike Heavy Ball, Adam's adaptive mechanisms prevented velocity from accelerating excessively, making it significantly more stable. Compared to constant step-size GD, Adam required less hyperparameter tuning and maintained convergence across varying conditions, making it the most robust method among those tested. Smaller mini-batches introduced only minor trajectory fluctuations, and increasing the batch size consistently improved convergence smoothness and speed, yet it remained remarkably robust across all batch sizes. Although it did not drastically outperform RMSProp in terms of speed, its ability to balance step size adjustments and maintain trajectory stability made it the preferred choice when training requires adaptability across different conditions.

Plots

The loss curves, trajectories, and parameter updates for each optimization method are illustrated in Figure 32, which supports the observations discussed above.

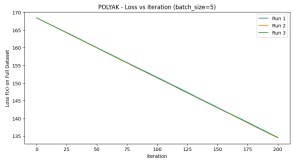

Figure 24: Polyak Loss (batch=5)

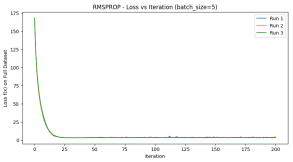

Figure 25: RMSProp Loss (batch=5)

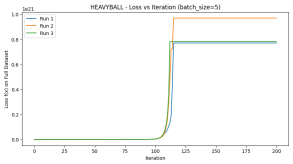

Figure 26: Heavy Ball Loss (batch=5)

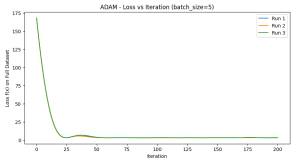

Figure 27: Adam Loss (batch=5)

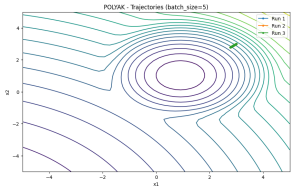

Figure 28: Polyak Trajectory (batch=5)

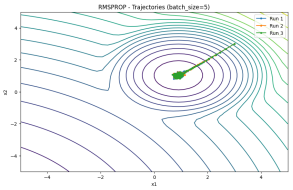

Figure 29: RMSProp Trajectory (batch=5)

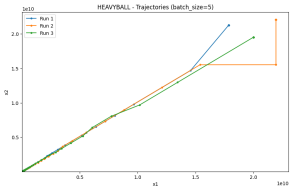

Figure 30: Heavy Ball Trajectory (batch=5)

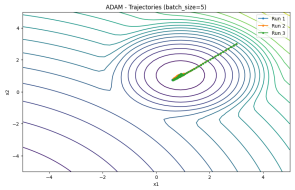

Figure 31: Adam Trajectory (batch=5)

Figure 32: Comparison of Loss and Trajectory for Different Optimization Methods (batch=5)

To illustrate the response of the optimization methods to varying batch sizes, we compare the loss curves of RMSProp and Adam for batch sizes of 5 and 25. Notably, for these methods, larger batch sizes tend to facilitate faster convergence, as previously mentioned.

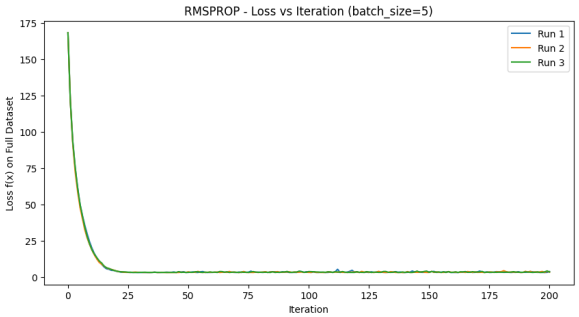

Figure 33: RMSProp Loss (batch=5)

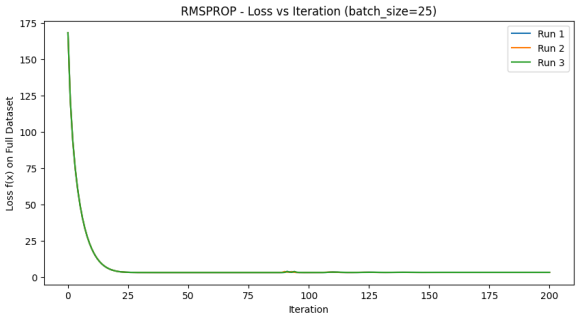

Figure 34: RMSProp Loss (batch=25)

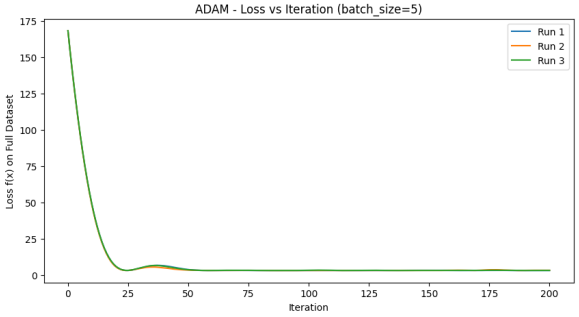

Figure 35: Adam Loss (batch=5)

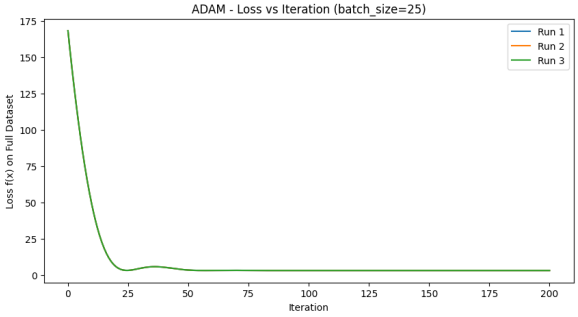

Figure 36: Adam Loss (batch=25)

Figure 37: Comparison of Loss Curves with Different Batch Sizes

Conclusion

This report evaluated various stochastic optimization strategies through systematic experimentation on a two-dimensional, piecewise-quadratic loss function. Initially, full-batch gradient descent exhibited smooth and rapid convergence, providing a stable baseline for comparison. Transitioning to mini-batch stochastic gradient descent (SGD), we observed that smaller batch sizes introduced greater variability and noise into parameter updates, leading to stochastic yet reliable convergence around the global minimum. These experiments highlighted a trade-off: smaller batch sizes accelerated early progress but resulted in noisy trajectories, whereas larger batches provided stable, predictable convergence at the cost of computational efficiency per iteration.

Exploring adaptive optimization methods: Polyak's adaptive step, RMSProp, Heavy Ball momentum, and Adam provided deeper insights into their practical behaviors. Polyak's method, despite its theoretical appeal, demonstrated cautious yet consistent convergence, limited primarily by sensitivity to gradient noise and reliance on an accurate optimal loss estimate. RMSProp effectively moderated the influence of noisy gradients by adaptively scaling step sizes, resulting in robust and stable convergence across varying batch sizes. Conversely, the Heavy Ball method proved highly sensitive to hyperparameter selection and noise, frequently overshooting the optimum and diverging under stochastic conditions, underscoring its limitations for mini-batch scenarios without extensive tuning.

Adam emerged as the most robust and practically effective optimizer, combining the beneficial aspects of RMSProp and momentum-based approaches. It consistently showed rapid convergence, reduced sensitivity to hyperparameters, and resilience to gradient noise across batch sizes. Adam's performance suggests it is particularly advantageous when optimizing complex, noisy functions or when precise hyperparameter tuning is impractical.

Overall, this assignment reinforced the importance of carefully selecting both the optimization algorithm and associated hyperparameters to balance convergence speed, stability, and robustness to noise. While simpler methods like constant-step SGD can suffice in straightforward problems, adaptive approaches, especially Adam, offer significant practical advantages in realistic settings involving noisy or inconsistent gradients.

A Code

```
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(20*(z[0]**2+z[1]**2), (z[0]+9)**2+(z[1]+10)**2)
        count=count+1
    return y/count

# Generate training data
train_data = generate_trainingdata(m=25)

class SGDOptimizer:
    def __init__(self, method='constant', **kwargs):
        self.method = method
        self.params = kwargs
        self.state = {}

        # Initialize method-specific parameters
        if method == 'rmsprop':
            self.state['cache'] = 0
            self.rho = kwargs.get('rho', 0.9)
        elif method == 'heavyball':
            self.state['velocity'] = 0
            self.momentum = kwargs.get('momentum', 0.9)
        elif method == 'adam':
            self.state['m'] = 0
            self.state['v'] = 0
            self.beta1 = kwargs.get('beta1', 0.9)
            self.beta2 = kwargs.get('beta2', 0.999)
            self.epsilon = kwargs.get('epsilon', 1e-8)
            self.t = 0

    def compute_step(self, grad: np.ndarray) -> float:
        method_functions = {
            'constant': self._constant,
            'polyak': self._polyak,
            'rmsprop': self._rmsprop,
            'heavyball': self._heavyball,
            'adam': self._adam
        }
        return method_functions[self.method](grad)

    def _constant(self, grad):
        return self.params['alpha'] * grad

    def _polyak(self, grad):
        return self.params['gamma'] * ((self.params['f'] - self.params['f_opt']) /
            (np.linalg.norm(grad)**2 + self.params['epsilon'])) * grad

    def _rmsprop(self, grad):
        self.state['cache'] = self.rho * self.state['cache'] + (1 - self.rho) *
            grad**2
        return self.params['alpha'] * grad / (np.sqrt(self.state['cache']) + 1e-6)

    def _heavyball(self, grad):
        self.state['velocity'] = self.momentum * self.state['velocity'] -
            self.params['alpha'] * grad
        return self.state['velocity']

    def _adam(self, grad):
        self.t += 1
        self.state['m'] = self.beta1 * self.state['m'] + (1 - self.beta1) * grad
        self.state['v'] = self.beta2 * self.state['v'] + (1 - self.beta2) * grad**2

        # Bias correction
        m_hat = self.state['m'] / (1 - self.beta1**self.t)
        v_hat = self.state['v'] / (1 - self.beta2**self.t)

        return self.params['alpha'] * m_hat / (np.sqrt(v_hat) + self.epsilon)

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Compute loss over grid
```

```
def compute_loss(x1, x2, data):
    X = np.array([x1, x2])
    return f(X, data)

# Create grid
x = np.linspace(-5, 5, 50)
y = np.linspace(-5, 5, 50)
X, Y = np.meshgrid(x, y)
Z = np.zeros_like(X)

for i in range(X.shape[0]):
    for j in range(X.shape[1]):
        Z[i,j] = compute_loss(X[i,j], Y[i,j], train_data)

# Plotting
fig = plt.figure(figsize=(12, 6))
ax = fig.add_subplot(121, projection='3d')
ax.plot_wireframe(X, Y, Z, rstride=3, cstride=3)
ax.set_title('Wireframe Plot of Loss Function')

ax2 = fig.add_subplot(122)
contour = ax2.contour(X, Y, Z, 20)
plt.colorbar(contour)
ax2.set_title('Contour Plot of Loss Function')
plt.tight_layout()
plt.show()

def finite_diff_grad(f, x, eps=1e-6):
    grad = np.zeros_like(x)
    for i in range(len(x)):
        x_plus = x.copy()
        x_plus[i] += eps
        x_minus = x.copy()
        x_minus[i] -= eps
        grad[i] = (f(x_plus) - f(x_minus)) / (2*eps)
    return grad

# Sample usage of finite difference gradient calculation
initial_point = np.array([1.0, 1.0])
gradient = finite_diff_grad(lambda x: f(x, train_data), initial_point)
print("Gradient at initial point {}: {}".format(initial_point, gradient))

def gradient_descent(f, grad_f, x0, alpha=0.01, max_iter=1000, tol=1e-6):
    x = x0.copy()
    trajectory = [x0]
    losses = [f(x0)]

    for _ in range(max_iter):
        grad = grad_f(x)
        x_new = x - alpha * grad
        trajectory.append(x_new)
        losses.append(f(x_new))

        if np.linalg.norm(x_new - x) < tol:
            break
        x = x_new

    return np.array(trajectory), np.array(losses)

x0 = np.array([3.0, 3.0])
traj_gd, losses_gd = gradient_descent(
    lambda x: f(x, train_data),
    lambda x: finite_diff_grad(lambda x: f(x, train_data), x),
    x0,
    alpha=0.01
)
plt.figure(figsize=(10, 4))
plt.plot(losses_gd, 'b.-')
plt.title('Loss vs Iteration (Constant Step Size GD)')
plt.xlabel('Iteration')
plt.ylabel('Loss f(x)')
plt.show()

plt.figure(figsize=(10, 6))
plt.contour(X, Y, Z, 20)
plt.plot(traj_gd[:,0], traj_gd[:,1], 'r.-', label='Trajectory')
```

```
plt.title('GD Trajectory (Constant Step Size)')
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.show()

def mini_batch_sgd(f, x0, train_data, batch_size=5, alpha=0.01, max_iter=200,
    tol=1e-6, optimizer_method='constant'):
    optimizer = SGDOptimizer(method=optimizer_method, alpha=alpha)

    x = x0.copy()
    trajectory = [x.copy()]
    # Record the loss on the full training data (for comparison)
    losses = [f(x, train_data)]

    for it in range(max_iter):
        # Randomly select a mini-batch of indices (without replacement)
        indices = np.random.choice(len(train_data), size=batch_size, replace=False)
        mini_batch = train_data[indices]

        # Compute approximate gradient on the mini-batch via finite differences
        grad = finite_diff_grad(lambda x: f(x, mini_batch), x)

        # Compute update step using the optimizer
        step = optimizer.compute_step(grad)
        x_new = x - step

        trajectory.append(x_new.copy())
        losses.append(f(x_new, train_data))

        if np.linalg.norm(x_new - x) < tol:
            break
        x = x_new

    return np.array(trajectory), np.array(losses)

# Run several experiments starting from x0 = [3, 3]
num_runs = 5
x0 = np.array([3.0, 3.0])
all_trajectories = []
all_losses = []

for run in range(num_runs):
    traj, losses = mini_batch_sgd(f, x0, train_data, batch_size=5, alpha=0.01,
        max_iter=200)
    all_trajectories.append(traj)
    all_losses.append(losses)

# Plotting loss vs iteration for each run
plt.figure(figsize=(10, 5))
for i, losses in enumerate(all_losses):
    plt.plot(losses, label=f'Run {i+1}')
plt.xlabel('Iteration')
plt.ylabel('Loss f(x)')
plt.title('Mini-batch SGD Loss vs Iteration')
plt.legend()
plt.show()

# Plotting the trajectories on the contour of the loss function
plt.figure(figsize=(10, 6))
plt.contour(X, Y, Z, levels=20)
for i, traj in enumerate(all_trajectories):
    plt.plot(traj[:, 0], traj[:, 1], marker='.', label=f'Run {i+1}')
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Trajectories of mini-batch SGD on Loss')
plt.legend()
plt.show()

# Varying mini-batch sizes experiment:
batch_sizes = [1, 5, 10, len(train_data)]  # 1: stochastic; len(train_data): full
    batch
results = {}

for bs in batch_sizes:
    trajectories_bs = []
```

```
        losses_bs = []
        for run in range(num_runs):
            traj, loss_run = mini_batch_sgd(f, x0, train_data, batch_size=bs,
                alpha=0.01, max_iter=200)
            trajectories_bs.append(traj)
            losses_bs.append(loss_run)
        results[bs] = {'trajectories': trajectories_bs, 'losses': losses_bs}

    # Plot loss vs iteration for varying mini-batch sizes:
    for bs in batch_sizes:
        plt.figure(figsize=(10, 6))
        i = 0
        for loss_run in results[bs]['losses']:
            plt.plot(loss_run, label=f'Run {i + 1}')
            i += 1
        plt.xlabel('Iteration')
        plt.ylabel('Loss f(x)')
        plt.title(f'Loss vs. Iteration for Mini-Batch Size {bs}')
        plt.legend()
        plt.show()

    # Plot trajectories on the loss contour for varying mini-batch sizes:
    for bs in batch_sizes:
        plt.figure(figsize=(10, 6))
        plt.contour(X, Y, Z, levels=20)
        i = 0
        for traj in results[bs]['trajectories']:
            plt.plot(traj[:, 0], traj[:, 1], marker='.', label=f'Run {i + 1}')
            i += 1
        plt.xlabel('x1')
        plt.ylabel('x2')
        plt.title(f'Trajectories on Loss Contour for Mini-Batch Size {bs}')
        plt.legend()
        plt.show()


    step_sizes = [0.001, 0.005, 0.01, 0.05, 0.1]
    results_steps = {}

    for alpha in step_sizes:
        trajectories_alpha = []
        losses_alpha = []
        for run in range(num_runs):
            traj, loss_run = mini_batch_sgd(f, x0, train_data, batch_size=5,
                alpha=alpha, max_iter=200)
            trajectories_alpha.append(traj)
            losses_alpha.append(loss_run)
        results_steps[alpha] = {'trajectories': trajectories_alpha, 'losses':
            losses_alpha}

    # Plot loss vs. iteration for different step sizes:
    for alpha in step_sizes:
        plt.figure(figsize=(10, 6))
        i = 0
        for loss_run in results_steps[alpha]['losses']:
            plt.plot(loss_run, label=f'Run {i + 1}')
            i += 1
        plt.xlabel('Iteration')
        plt.ylabel('Loss f(x)')
        plt.title(f'Loss vs. Iteration for Step Size {alpha} (mini-batch size = 5)')
        plt.legend()
        plt.show()

    # Plot trajectories on the loss contour for different step sizes:
    for alpha in step_sizes:
        plt.figure(figsize=(10, 6))
        plt.contour(X, Y, Z, levels=20)
        i = 0
        for traj in results_steps[alpha]['trajectories']:
            plt.plot(traj[:, 0], traj[:, 1], marker='.', label=f'Run {i + 1}')
            i += 1
        plt.xlabel('x1')
        plt.ylabel('x2')
        plt.title(f'Trajectories on Loss Contour for Step Size {alpha} (mini-batch size
            = 5)')
        plt.legend()
```

```
    plt.show()


def mini_batch_sgd_custom(method, method_params, x0, train_data,
                          batch_size=5, max_iter=100, num_runs=3):

    all_trajectories = []
    all_losses = []

    for run in range(num_runs):
        optimizer = SGDOptimizer(method=method, **method_params)

        x = x0.copy()
        trajectory = [x.copy()]
        losses = [f(x, train_data)]

        for it in range(max_iter):
            indices = np.random.choice(len(train_data), size=batch_size,
                replace=False)
            mini_batch = train_data[indices]

            grad = finite_diff_grad(lambda xx: f(xx, mini_batch), x)

            step = optimizer.compute_step(grad)

            x_new = x - step
            trajectory.append(x_new.copy())
            losses.append(f(x_new, train_data))

            x = x_new

        all_trajectories.append(np.array(trajectory))
        all_losses.append(np.array(losses))

    return all_trajectories, all_losses

# (i) Polyak step size
#   Since we know through our expensive run of GD the global minima is around (1.0,
    1.0)
#   thus we can deduce that the optimal loss should be around 0.0 -> f_opt = 0.0
polyak_params = {
    'f': f(x0, train_data),  # we can just use the current f for demonstration
    'f_opt': 0.0,
    'epsilon': 1e-6,
    'gamma': 0.001
}

# (ii) RMSProp
rmsprop_params = {
    'alpha': 0.1,  # learning rate
    'rho': 0.9      # decay rate for RMSProp
}

# (iii) Heavy Ball
heavyball_params = {
    'alpha': 0.1,   # step size
    'momentum': 0.001  # momentum coefficient
}

# (iv) Adam
adam_params = {
    'alpha': 0.1,    # step size
    'beta1': 0.9,
    'beta2': 0.999,
    'epsilon': 1e-8
}

batch_sizes = [1, 5, 10, len(train_data)]
max_iter = 200
num_runs = 3  # number of runs for each method

methods = {
    'polyak': polyak_params,
    'rmsprop': rmsprop_params,
    'heavyball': heavyball_params,
    'adam': adam_params
```

```
    }

    results = {}

    for batch_size in batch_sizes:
        for method_name, params in methods.items():
            trajectories, losses = mini_batch_sgd_custom(
                method_name, params, x0, train_data,
                batch_size=batch_size, max_iter=max_iter, num_runs=num_runs
            )
            results[(method_name, batch_size)] = (trajectories, losses)

    # Plot losses vs iteration for each method and batch size
    for (method_name, batch_size), (trajectories, losses) in results.items():
        plt.figure(figsize=(10, 5))
        for run_id, loss_run in enumerate(losses):
            plt.plot(loss_run, label=f'Run {run_id+1}')
        plt.title(f'{method_name.upper()} - Loss vs Iteration (batch_size={batch_size})')
        plt.xlabel('Iteration')
        plt.ylabel('Loss f(x) on Full Dataset')
        plt.legend()
        plt.show()

    # Plot trajectories on the loss contour for each method and batch size
    for (method_name, batch_size), (trajectories, losses) in results.items():
        plt.figure(figsize=(10, 6))
        plt.contour(X, Y, Z, levels=20)
        for run_id, traj in enumerate(trajectories):
            plt.plot(traj[:, 0], traj[:, 1], marker='.', label=f'Run {run_id+1}')
        plt.title(f'{method_name.upper()} - Trajectories (batch_size={batch_size})')
        plt.xlabel('x1')
        plt.ylabel('x2')
        plt.legend()
        plt.show()
```