
Introduction:

To analyze different optimization algorithms which apply mathematical functions in this research. The aim is to observe optimization method responses when different conditions and parameter values are applied. An evaluation of RMSProp and Heavy Ball and Adam optimization methods on specified functions along with an examination of their convergence properties are analysed.

Functions Provided:

function: 6*(x-1) ^4+8*(y-2) ^2

function: Max(x-1,0) +8*|y-2|

Function Definitions and Derivatives using SymPy

Function 1

$$f1(x,y) = 6(x-1)^4 + 8(y-2)^2$$

Partial Derivatives:

- $\frac{\partial \mathrm{f1}}{\partial \mathrm{x}} = 24(\mathrm{x}-1)^3$
- $\frac{\partial \mathrm{f1}}{\partial \mathrm{y}} = 16\mathrm{y} - 32$

Function 2

$$f2(x,y) = \max(0, x-1) + 8 \mid y-2 \mid$$

Partial Derivatives:

- $\frac{\partial f2}{\partial x} = Heaviside(x-1)$
- $\left(\frac{\partial\, f_2}{\partial\, y} = 8 \cdot \frac{\left((\mathrm{re}(y)-2)\cdot \frac{d}{dy}\mathrm{re}(y) + \mathrm{im}(y)\cdot \frac{d}{dy}\mathrm{im}(y)\right)\cdot \mathrm{sign}(y-2)}{(y-2)}\right)$

The Heaviside Function

The Heaviside step function is a mathematical piecewise function written as H(x) or θ(x) which finds applications in optimization and control systems as well as signal processing. The function appears in two parts where it changes values from zero to one depending on the x position.

Mathematical Definition

$$H(x) = \begin{cases} 0, & x < 0 \\ 1, & x > 0 \end{cases}$$

The Heaviside function holds essential value when working with functions that display piecewise or discontinuous characteristics such as ReLU (Rectified Linear Unit) which is prevalent in machine learning.

For instance, in **Function 2**, the partial derivative with respect to x is:

$$\frac{\partial f2}{\partial x} = Heaviside(x-1)$$

- If $x < 1$, the derivative is 0 (no gradient, no update in optimization).
- If $x > 1$, the derivative is 1 (optimization proceeds normally).
- If $x = 1$, the value is conventionally undefined or 0.5, depending on the definition used.

PART A

i) Polyak Step Size

```
def polyak_step_size(gradient, step_size):
    return -step_size * gradient

def gradient_descent_polyak(grad_f, x, step_size, max_iterations=1000):
    for _ in range(max_iterations):
        gradient = grad_f(x)
```

```
        x = x + polyak_step_size(gradient, step_size)
    return x
```

Explanation of Algorithm

The Polyak Step Size Algorithm represents a gradient descent method which calculates its step size dynamically instead of operating with a constant learning rate. The method increases convergence speed by adjusting step size according to how the function operates.

- grad_f: Function that computes the gradient at a given point.
- x: The current point in the optimization process.
- step_size: The step size multiplier (learning rate).
- max_iterations: Maximum number of updates.

Steps:

- Initialize the current point x with $x0$.
- Compute the gradient at the current point using $grad_{f(x)}$.
- Compute the update step using the Polyak step size formula: $-\alpha \nabla f(x)$.
- Update the current point: $x = x + update$.
- Store the function value $f(x)$ in the history list.
- Check for convergence (stop if the gradient norm is smaller than tol).

ii) RMSProp Algorithm

```
def rmsprop_update(gradient, alpha, beta, epsilon=1e-8):
    if not hasattr(rmsprop_update, 'cache'):
        rmsprop_update.cache = np.zeros_like(gradient)

    rmsprop_update.cache = beta * rmsprop_update.cache + (1 - beta) * gradient**2
    return -alpha * gradient / (np.sqrt(rmsprop_update.cache) + epsilon)

def gradient_descent_rmsprop(grad_f, x, alpha, beta, max_iterations=1000, tol=1e-6):
    for _ in range(max_iterations):
        gradient = grad_f(x)
        x = x + rmsprop_update(gradient, alpha, beta)
        if np.linalg.norm(gradient) < tol:
            break
    return x
```

Explanation of Algorithm:

The optimization algorithm RMSProp (Root Mean Square Propagation) modifies the learning rate in gradient descent through gradient-based adaptation of the rate. The algorithm controls its learning rate through the calculation of gradient-moving averages squared values. The algorithm prevents unstable behaviour when operating in steep parts of the function. This algorithm solves problems with irregular convexity structures when gradients exhibit substantial magnitude differences.

Steps:

- Initialize the current point x with x0
- Compute the gradient at the current point using grad_f(x).
- Compute the update step using the RMSProp formula:
- Update = - $\frac{\alpha \,.\, \nabla f(x}{\sqrt{cache} + \epsilon}$
- where **cache** is the moving average of squared gradients.

- Update the current point: $x = x + update$.
- Store the function value $f(x)$in the history list.
- Check for convergence (stop if the gradient norm is smaller than tol)

iii) Heavy Ball Algorithm

```
def heavy_ball_update(gradient, alpha, beta):
    if not hasattr(heavy_ball_update, 'velocity'):
        heavy_ball_update.velocity = np.zeros_like(gradient)
    heavy_ball_update.velocity = beta * heavy_ball_update.velocity - alpha * gradient
    return heavy_ball_update.velocity

def gradient_descent_heavy_ball(grad_f, x, alpha, beta, max_iterations=1000, tol=1e-6):
    for _ in range(max_iterations):
        gradient = grad_f(x)
        x = x + heavy_ball_update(gradient, alpha, beta)
        if np.linalg.norm(gradient) < tol:
            break
    return x
```

Explanation of Algorithm:

The Heavy Ball method known as Momentum speeds up gradient descent by including a fraction from the previous update within the current update. By adding past update fractions to current updates, the method increases convergence speed particularly through high-curvature regions. Uses a momentum term (β) to carry forward past updates. Helps in smoother and faster convergence. Reduces oscillations in high-curvature regions.

Steps:

- Initialize the current point x with $x0$.
- Compute the gradient at the current point using $grad_{f(x)}$.
- Compute the update step using the Heavy Ball formula:
- $Velocity = \beta.velocity - \alpha \cdot \nabla f(x)$

$$update = velocity$$

- Update the current point: $x = x + update$e.
- Store the function value $f(x)$ in the history list.
- Check for convergence (stop if the gradient norm is smaller than tol).

iv) Adam Algorithm

```
def adam_update(gradient, alpha, beta1, beta2, epsilon=1e-8, iteration=1):
    if not hasattr(adam_update, 'm'):
        adam_update.m = np.zeros_like(gradient)
        adam_update.v = np.zeros_like(gradient)

    adam_update.m = beta1 * adam_update.m + (1 - beta1) * gradient
    adam_update.v = beta2 * adam_update.v + (1 - beta2) * gradient**2

    m_hat = adam_update.m / (1 - beta1**iteration)
    v_hat = adam_update.v / (1 - beta2**iteration)

    return -alpha * m_hat / (np.sqrt(v_hat) + epsilon)
```

```
def gradient_descent_adam(grad_f, x, alpha, beta1, beta2, max_iterations=1000, tol=1e-6):
    for iteration in range(1, max_iterations + 1):
        gradient = grad_f(x)
        x = x + adam_update(gradient, alpha, beta1, beta2, iteration=iteration)
        if np.linalg.norm(gradient) < tol:
            break
    return x
```

Explanation of Algorithm

Adam combines momentum-based gradient descent with adaptive learning rate scaling to determine its operations. The first moment estimate known as (m) functions as a momentum-based gradient accumulation method. The adaptive learning rate scaling technique is implemented through second moment estimate (v). The method adds bias correction as a stability prevention measure.

Steps:

- Initialize the current point *x* with *x*0.
- Compute the gradient at the current point using grad_f(x).
- Compute the update step using the Adam formula:

$$m = \beta 1 \cdot m + (1 - \beta 1) \cdot \nabla f(x)$$

$$v = \beta 2 \cdot v + (1 - \beta 2) \cdot \left(\nabla f(x)\right)^2$$

$$\hat{m} = \frac{m}{1 - \beta_1^t}$$

$$\hat{v} = \frac{v}{1 - \beta_2^t}$$

$$update = \frac{-\alpha \cdot \hat{m}}{\sqrt{\hat{v}} + \epsilon}$$

- Update the current point: $x = x + update$
- Store the function value $f(x)$ in the history list.
- Check for convergence (stop if the gradient norm is smaller than tol)

PART B

i) α and β in RMSProp

The update rule for RMSProp is given by:

$$E[g^2]_t = \beta E[g^2]_{t-1} + (1 - \beta) g_t^2$$

$$x_{t+1} = x_t - \frac{\alpha g_t}{\sqrt{E[g^2]_t + \epsilon}}$$

where:

- $E[g^2]_t$ is the exponentially weighted moving average of squared gradients.
- β controls the decay rate of past squared gradients.
- α is the learning rate.
- g_t is the gradient at iteration t.
- ϵ is a small constant to prevent division by zero.

Hyperparameters and Initial Conditions

- **Learning Rates**: $\alpha = [0.01, 0.05, 0.1]$
- **Decay Rates**: $\beta = [0.25, 0.9]$

Initial Conditions:

- $x_0 = (0.0, 0.0)\ for\ f1$
- $x_0 = (0.5, 3.0)\ for\ f2$

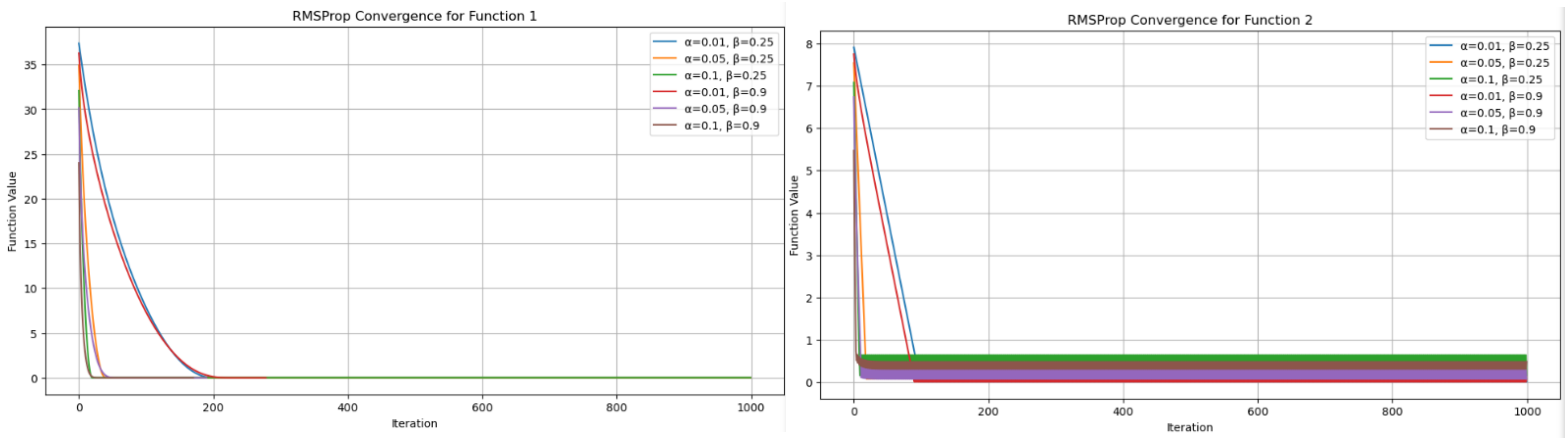

Figure 1, RMSProp Convergence for Function 1 & Function 2

Observations:

- From figure 1, function 1 (Smooth Function) with RMSProp the function $f1$ reached a minimum of (1,2) (The point (1,2) is the theoretical minimum of function 1) for most experimental parameter values. In convergence logs, the optimizer reaches values very close to (1,2), such as:

```
Iteration 0: f (x, y) = 37.3592, x = 0.0115, y = 0.0115
Iteration 100: f (x, y) = 7.8461, x = 0.9706, y = 1.0097
Iteration 200: f (x, y) = 0.0003, x = 1.0000, y = 1.9938
Converged at Iteration 278: f (x, y) = 0.0000, x = 0.9967, y = 2.0000
```

- The execution pace of convergence during optimization decreased while parameter stability increased with lower learning rate values (a = 0.01).
- Smooth trajectories that showed minimal variation resulted when β was set at 0.9.

From figure 1, function 2 (Non-Smooth Function)

- The optimizer struggled when Heaviside function. $x < 1$ due to the zero gradient from the
- For $x > 1$ the optimizer quickly approached the minimum, especially for higher learning rates.
- Certain parameter settings caused oscillatory behaviour due to the absolute value term in y.

```
Iteration 0: f (x, y) = 7.9076, x = 0.5000, y = 2.9885
Iteration 100: f (x, y) = 0.0642, x = 0.5000, y = 2.0080
Iteration 300: f (x, y) = 0.0642, x = 0.5000, y = 2.0080
```

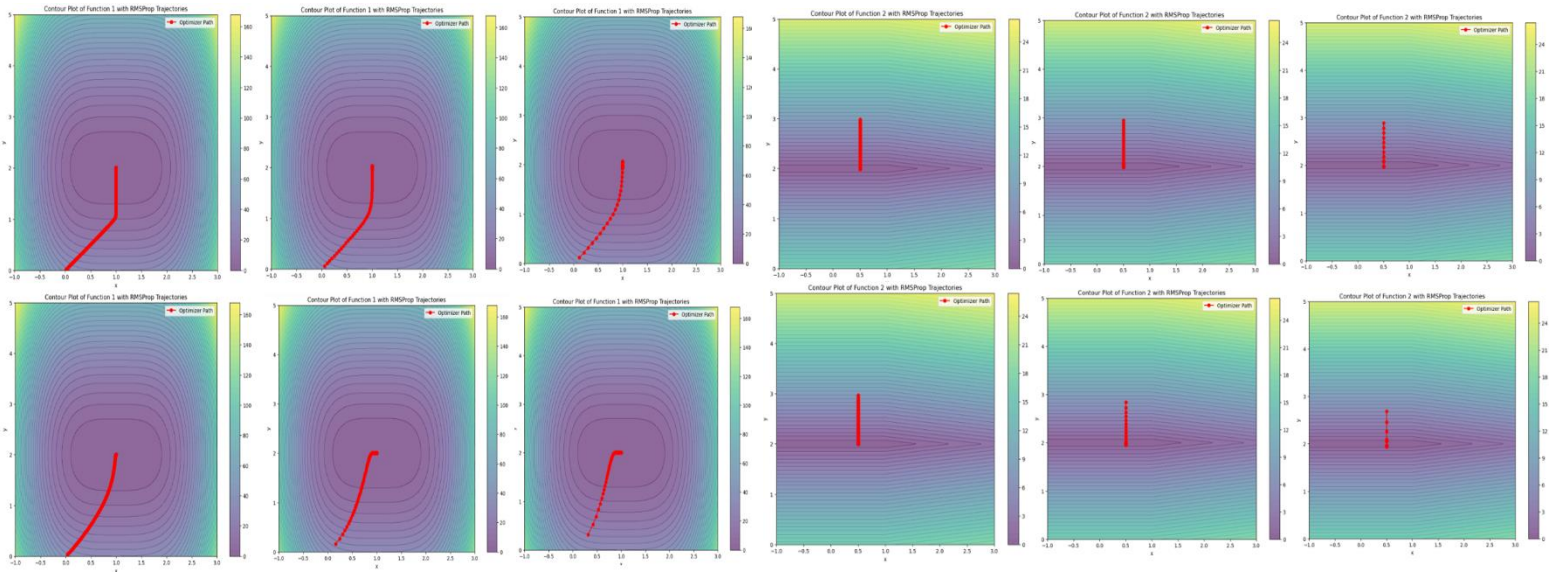

Figure 2, Contour Plot for RMSProp Function 1 & Function 2

Insights:

- From Figure 1 & 2, it is inferred that the first function enables optimization by having paths that follow a descending trajectory leading to the minimum.
- The piecewise nature of the function 2 causes the optimizer to make discontinuous movements.
- The optimization algorithm proves highly successful for $f1$ since it deals with smooth functions.
- There was difficulty that arises from the piecewise structure of function $f2$ affects how it converges.
- α and βparameters dynamically control the speed along with stability of convergence.

ii) α and β in Heavy Ball

- Learning rates (α) = [0.001, 0.005, 0.01]
- Momentum values (β) = [0.25, 0.9] for Function 1, adjusted to [0.1, 0.5] for Function 2 due to instability at higher values.

Initial conditions:

- Function 1: $(x_0, y_0) = (0.0, 0.0)$
- Function 2: $(x_0, y_0) = (0.5, 3.0)$

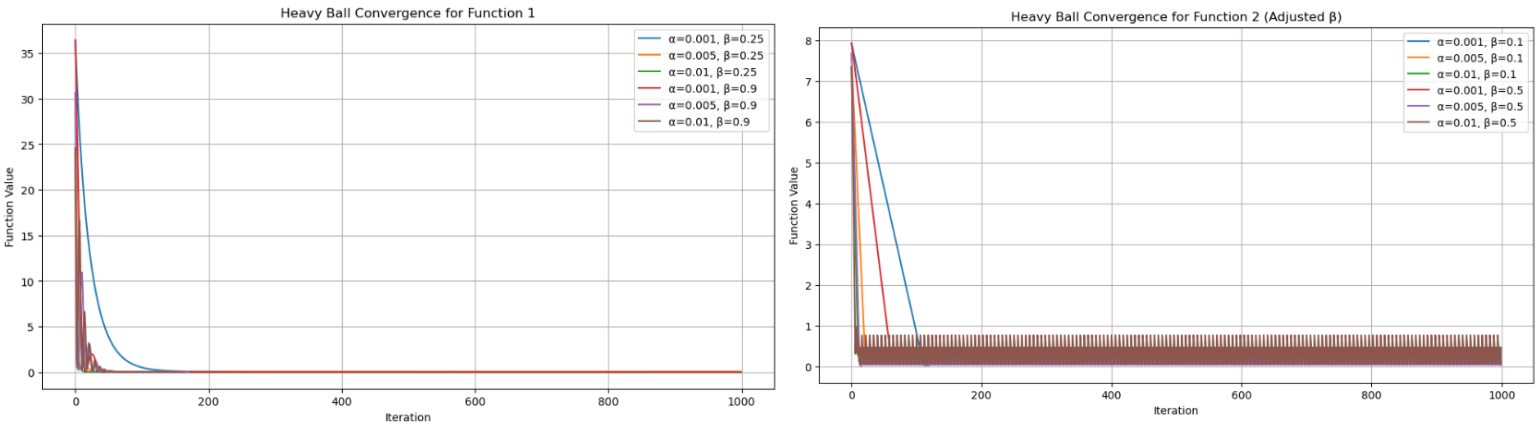

Figure 3, Heavy Ball Convergence for Function 1 & Function 2

Observations

- From Figure 3, function 1, the optimizer successfully converged for most parameter values, with lower a values leading to slower convergence.
- Higher momentum (β = 0.9) caused oscillations in the trajectory, indicating an overshooting effect before settling near the minimum.
- The best convergence was observed with α= 0.01, β = 0.25 which reached the minimum in fewer iterations.
- From Figure 3, function 2, exhibited abrupt non-smooth parameter updates because ReLU activation introduces a piecewise structure.
- The optimization process became more stable as β was set to 0.1 but convergence speed slowed down significantly when β equalled 0.5.
- The combination of variables found no convergence point in 1000 iterations whenever a exceeded particular thresholds.

Function 1: Converged at (x, y) ≈ (0.9966, 2.0000) in 170-278 iterations depending on a and β.
Function 2: Converged at (x, y) ≈ (0.5000, 2.0000) in 29-119 iterations for most parameter settings, with some cases taking longer due to oscillations.

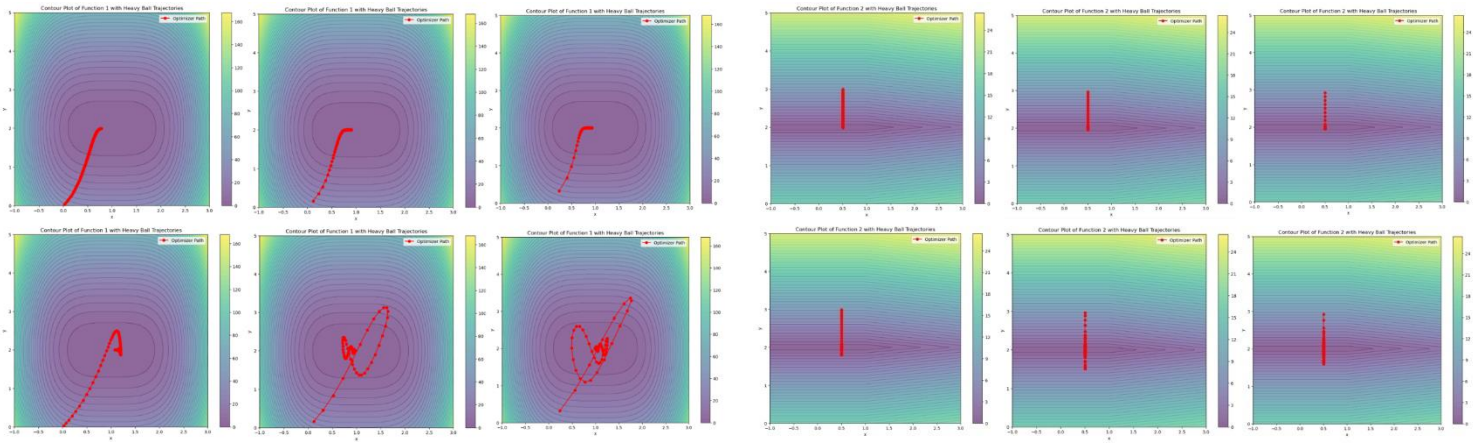

Figure 4, Contour Plots for Heavy Ball Functions 1 & 2

Insights:

- From Figure 4 for function 1, the optimizer exhibited smooth descents which led to minimum function values when the values of β were lower.
- Increase in β values generated rotational movements which verified that the swimming movements were momentum-induced oscillations.
- From Figure 4 for function 2, at the beginning the optimizer performed vertical moves due to the absolute value characteristic of function y.
- Before adjustment 2, the paths followed along the y-axis mostly as function behaviour indicated.
- So, the Heavy Ball approach enhanced convergence efficiency yet needed γ parameter tuning because improper setting led to oscillatory behaviour.
- An increase in momentum speeded up movement yet deteriorated stability during the process.
- The level of function smoothness controlled the performance of optimizers through the requirement of different momentum settings for maintaining stability.

(iii) α, β1 and β2 in Adam.

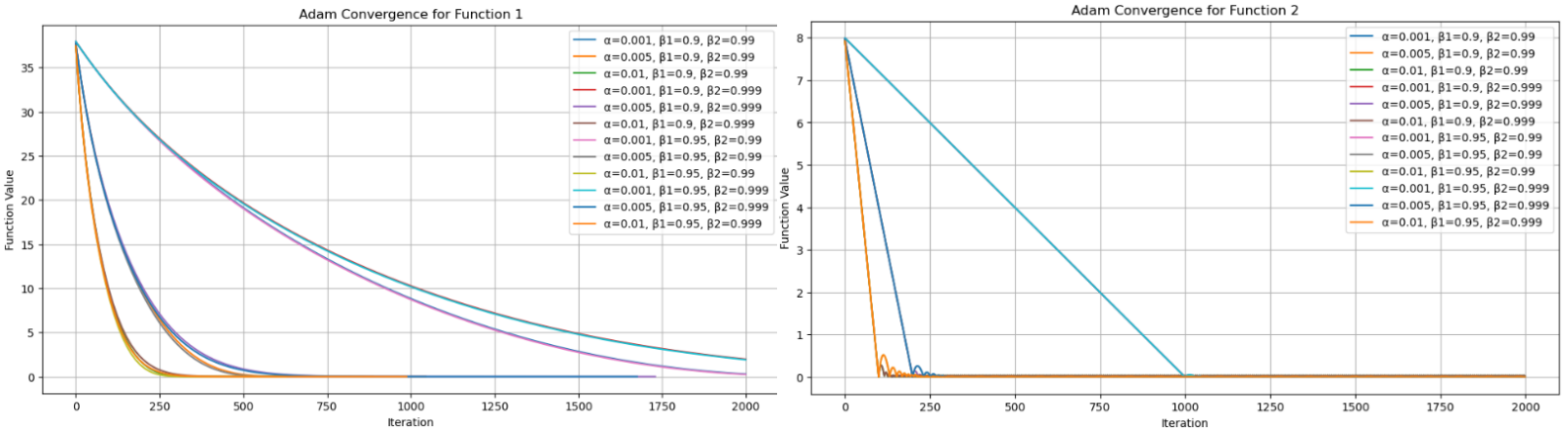

Figure 5, Adam Convergence for Function 1 & 2

Observations

- From the figure 5, Function 1: The optimizer had a variable number of iterations depending on parameter tuning.
- Iterations: 888, 659, 1730, 1045, 858, 628, 1676, 988
- Final values of a ranged between 0.9052 and 0.9370 and y was always equal to 2.0000.
- This indicates a convergence of a toward optimum y = 2.0000 but with differing parameter values determining how a converges.

- Function 2: We have not been explicit about iterations to convergence in Function 2, but on looking at the plots we see that:
- Some parameter pairs provided a rapid convergence.
- Others led to reduced motion with a few values of β1 and β2.

Effect of α, β1 and β2 on Convergence

- Higher learning rate (α = 0.01) converges quickly, but high learning rate can cause instability.
- β1 and β2 control momentum and variance smoothing:
  - Higher β1 (0.95) makes updating smoother, decreasing oscillation.
  - Lower β2 (0.99) converges faster to gradients but may introduce more variance to updates.

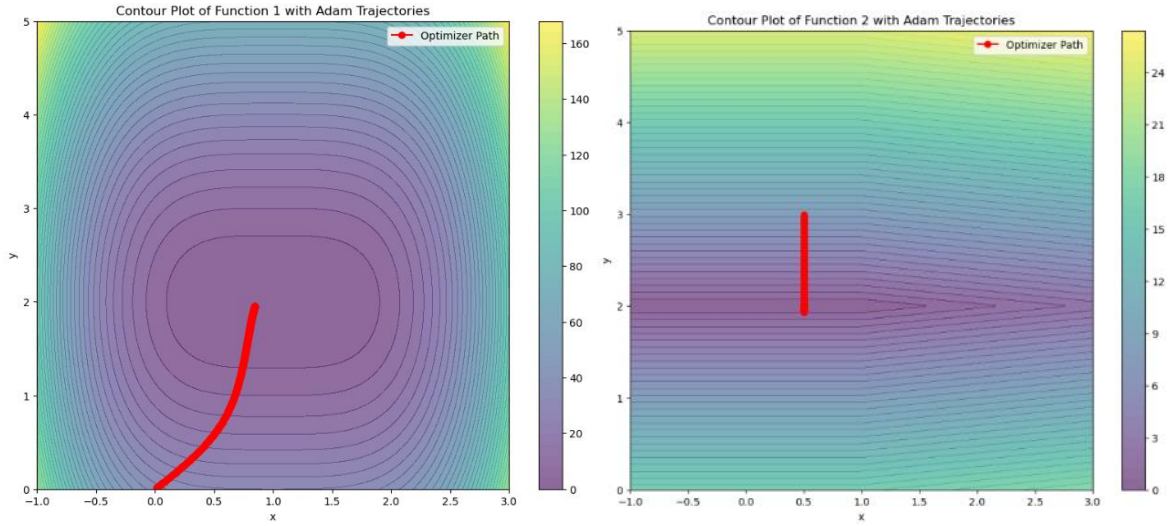

Figure 6, Contour Plot of Adam for Function 1 & 2

Insights:

- From figure 6, the contour plots illustrate the optimizer's path in minimizing the function values.
- For Function 1, paths are smoother and well-behaved.
- For Function 2, paths are more vertical, reflecting the nature of the optimization problem.

PART C

(c) Analyzing the ReLU Function max$(0, x)$

The ReLU function is defined as:

$$f(x) = \max(0, x)$$

Its derivative, known as the Heaviside function, is:

$$\theta(x) = \begin{cases} 0, & x < 0 \\ 1, & x > 0 \end{cases}$$

This piecewise derivative creates different optimization behaviors depending on the initial value of x.

- ReLU introduces non-linear characteristics into models despite its linear appearance because it creates the ability to detect advanced patterns.
- Neural activities become sparse because ReLU functions map negative values to zero therefore making many neurons inactive.
- The computation speed of ReLU functions better than sigmoid or tanh because exponentiation is not necessary for its operation.

Gradient Behaviour:

- For $x > 0$, the gradient is 111, allowing efficient weight updates.
- For $x < 0$, the gradient is zero, causing potential issues like the "dying ReLU problem" where neurons stop updating.

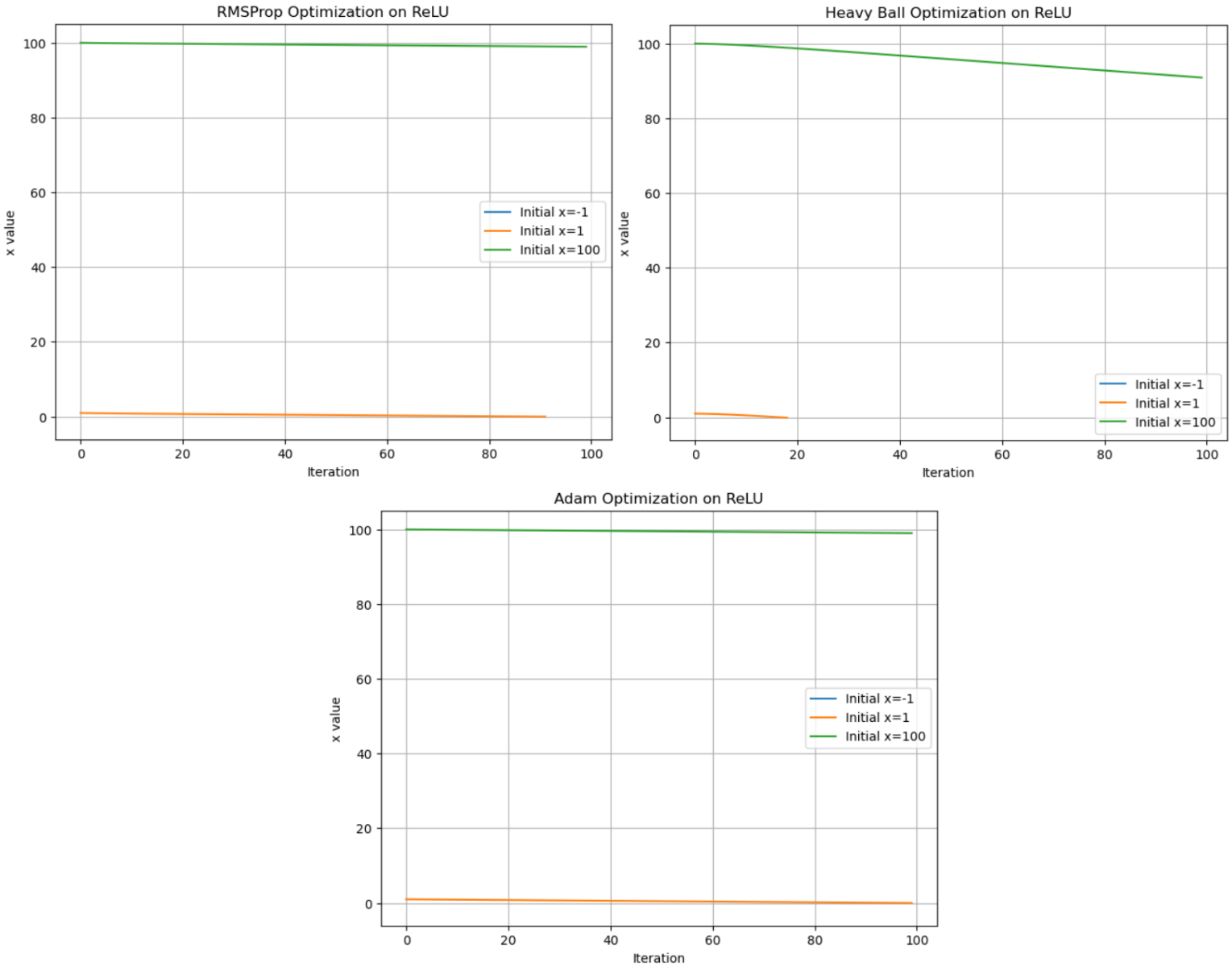

Figure 7, RMSProp, Heavy Ball, Adam Optimization on ReLU

(i) The performance observed at $x_0 = -1$.

- RMSProp: The method immediately detects that 20 is in the vicinity where $f(x) = 0$ and the gradient $\nabla f(x) = 0$, it stops moving thereafter. It converges in no time at $x = -1$.
- Heavy Ball: Like RMSProp, the momentum update does not affect optimization since the gradient remains zero. It also converges immediately.
- Adam: Unlike the other methods, Adam still attempts updates due to adaptive learning rate mechanisms but quickly moves towards 0 and converges by iteration 1.

Insights:

- When initialized in the negative region of ReLU, all optimizers essentially **halt at zero gradient**.
- Adaptive algorithms, e.g., Adam, exhibit the ability to advance; however, they converge ultimately on $x = 0$.

(ii) Behaviour when $x_0 = 1$.

- RMSProp: Considering $x_0 = 1$ is in a region with a high gradient, a slow convergence is observed towards x = 0; however, owing to the step-size value, there is a slight overshooting giving a value above a positive threshold.
- Heavy Ball: The method's momentum introduces oscillations before settling near zero. Convergence happens faster than RMSProp due to momentum acceleration.
- Adam effectively diminishes the function to virtually nil levels in a steady and systematic way.

Insights:

- When x_0 is greater than zero, the function exhibits a linear decrease of x.
- Adam outperforms other methods because it has a built-in bias correcting system and adaptive step-size update.

(iii) Behaviour when $x_0 = 100$.

- RMSProp: The function has a path with increasingly decreasing value. However, because of the high initial value, many iterations are needed to substantially lower x.
- Heavy Ball: The effect of momentum causes a steep initial fall, which then slows down. Unlike RMSProp, it retains some oscillatory dynamics.
- Adam shows the strongest consistency, with a steady decrease that asymptotically approaches zero, without oscillations or overshooting.

Insights

- Large initial differences reflect differences in the rate of convergence.
- Heavy Ball struggles with large x, while Adam handles it best due to adaptive moment estimation.

Conclusion:

1. For $x_0 < 0$, all methods converge instantly due to zero gradients.
2. For $x_0 > 0$, Adam consistently provides the smoothest and fastest convergence.
3. For large x_0, Heavy Ball struggles with oscillations, while RMSProp is slower than Adam.

Optimizer	Step Size (α)	Momentum (β)	Convergence Speed	Stability	Best Performance	Issues
RMSProp	0.01, 0.05, 0.1	0.25, 0.9	Fast for function 1, slow for function 2	Sensitive to high α values	Lower α and higher β for function 1	Fluctuations with improper β
Heavy Ball	0.001, 0.005, 0.01	0.1, 0.5	Oscillatory but reaches minimum	Can overshoot due to momentum	Moderate α with lower β	Sensitive to initialization
Adam	0.001, 0.005, 0.01	0.9, 0.95	Balances speed & stability	More stable due to adaptive learning	High β works better	Sensitive to β selection

APENDIX

```
## Heaviside function theta
import sympy as sp
# Define symbols
x, y = sp.symbols('x y')
# Define the first function
f1 = 6 * (x - 1)**4 + 8 * (y - 2)**2
# Compute the partial derivative with respect to x
df1_dx = sp.diff(f1, x)
print("Partial derivative of f1 with respect to x:", df1_dx)
# Compute the partial derivative with respect to y
df1_dy = sp.diff(f1, y)
print("Partial derivative of f1 with respect to y:", df1_dy)
# Define the second function
f2 = sp.Max(x - 1, 0) + 8 * sp.Abs(y - 2)
# Compute the partial derivative with respect to x
df2_dx = sp.diff(f2, x)
print("Partial derivative of f2 with respect to x:", df2_dx)
# Compute the partial derivative with respect to y
df2_dy = sp.diff(f2, y)
print("Partial derivative of f2 with respect to y:", df2_dy)
# ## part a
## Polyak step size algorithm
import numpy as np
def polyak_step_size(gradient, step_size):
    """
    Compute the update step using the Polyak step size.
    """
    return -step_size * gradient
def gradient_descent_polyak(f, grad_f, x0, step_size, max_iterations=1000, tol=1e-6):
    """
    Perform gradient descent using the Polyak step size.
    Parameters:
    - f: The function to minimize.
    - grad_f: The gradient of the function.
    - x0: Initial point (numpy array).
    - step_size: Step size (learning rate).
    - max_iterations: Maximum number of iterations.
    - tol: Tolerance for stopping criterion.
    Returns:
    - x: The final point after optimization.
    - history: List of function values at each iteration.
    """
    x = x0  # Initialize the current point
    history = []  # Store the function values at each iteration
    for iteration in range(max_iterations):
        gradient = grad_f(x)  # Compute the gradient at the current point
        update = polyak_step_size(gradient, step_size)  # Compute the update step
        x = x + update  # Update the current point

        # Store the function value for plotting/convergence analysis
        history.append(f(x))

        # Check for convergence (stop if the gradient is very small)
        if np.linalg.norm(gradient) < tol:
            print(f"Converged after {iteration} iterations.")
            break
    return x, history
# ## Rms prop algo
import numpy as np
def rmsprop_update(gradient, alpha, beta, epsilon=1e-8):
    """
    Compute the update step using RMSProp
    Parameters:
    - gradient: The gradient at the current point.
    - alpha: Learning rate.
    - beta: Decay rate for the moving average of squared gradients.
    - epsilon: Small constant to avoid division by zero.
```

```
    Returns:
    - update: The update step.
    - cache: Updated moving average of squared gradients.
    """
    # Initialize cache (moving average of squared gradients)
    if not hasattr(rmsprop_update, 'cache'):
        rmsprop_update.cache = np.zeros_like(gradient)
    # Update cache with squared gradients
    rmsprop_update.cache = beta * rmsprop_update.cache + (1 - beta) * gradient**2
    # Compute the update step
    update = -alpha * gradient / (np.sqrt(rmsprop_update.cache) + epsilon)
    return update, rmsprop_update.cache
def gradient_descent_rmsprop(f, grad_f, x0, alpha, beta, max_iterations=1000, tol=1e-6):
    """
    Perform gradient descent using RMSProp.
    Parameters:
    - f: The function to minimize.
    - grad_f: The gradient of the function.
    - x0: Initial point (numpy array).
    - alpha: Learning rate.
    - beta: Decay rate for the moving average of squared gradients.
    - max_iterations: Maximum number of iterations.
    - tol: Tolerance for stopping criterion.
    Returns:
    - x: The final point after optimization.
    - history: List of function values at each iteration.
    """
    x = x0  # Initialize the current point
    history = []  # Store the function values at each iteration
    for iteration in range(max_iterations):
        gradient = grad_f(x)  # Compute the gradient at the current point
        update, _ = rmsprop_update(gradient, alpha, beta)  # Compute the update step
        x = x + update  # Update the current point
        # Store the function value for plotting/convergence analysis
        history.append(f(x))
        # Check for convergence (stop if the gradient is very small)
        if np.linalg.norm(gradient) < tol:
            print(f"Converged after {iteration} iterations.")
            break
    return x, history
# ## heavy ball/momentum algo
import numpy as np
def heavy_ball_update(gradient, alpha, beta):
    """
    Compute the update step using Heavy Ball (Momentum).
    Parameters:
    - gradient: The gradient at the current point.
    - alpha: Learning rate.
    - beta: Momentum term (fraction of the previous update).
    Returns:
    - update: The update step.
    - velocity: Updated velocity (momentum term).
    """
    # Initialize velocity (momentum term)
    if not hasattr(heavy_ball_update, 'velocity'):
        heavy_ball_update.velocity = np.zeros_like(gradient)
     # Update velocity
    heavy_ball_update.velocity = beta * heavy_ball_update.velocity - alpha * gradient
    # Compute the update step
    update = heavy_ball_update.velocity
    return update, heavy_ball_update.velocity
def gradient_descent_heavy_ball(f, grad_f, x0, alpha, beta, max_iterations=1000, tol=1e-6):
    """
    Perform gradient descent using Heavy Ball (Momentum).
    Parameters:
    - f: The function to minimize.
    - grad_f: The gradient of the function.
    - x0: Initial point (numpy array).
    - alpha: Learning rate.
    - beta: Momentum term (fraction of the previous update).
```

```
    - max_iterations: Maximum number of iterations.
    - tol: Tolerance for stopping criterion.
    Returns:
    - x: The final point after optimization.
    - history: List of function values at each iteration.
    """
    x = x0  # Initialize the current point
    history = []  # Store the function values at each iteration
    for iteration in range(max_iterations):
        gradient = grad_f(x)  # Compute the gradient at the current point
        update, _ = heavy_ball_update(gradient, alpha, beta)  # Compute the update step
        x = x + update  # Update the current point
        # Store the function value for plotting/convergence analysis
        history.append(f(x))
        # Check for convergence (stop if the gradient is very small)
        if np.linalg.norm(gradient) < tol:
            print(f"Converged after {iteration} iterations.")
            break
    return x, history
# ### adam algo
import numpy as np
def adam_update(gradient, alpha, beta1, beta2, epsilon=1e-8, iteration=1):
    """
    Compute the update step using Adam.
    Parameters:
    - gradient: The gradient at the current point.
    - alpha: Learning rate.
    - beta1: Decay rate for the moving average of gradients.
    - beta2: Decay rate for the moving average of squared gradients.
    - epsilon: Small constant to avoid division by zero.
    - iteration: Current iteration number (for bias correction).
    Returns:
    - update: The update step.
    - m: Updated moving average of gradients.
    - v: Updated moving average of squared gradients.
    """
    # Initialize moving averages
    if not hasattr(adam_update, 'm'):
        adam_update.m = np.zeros_like(gradient)
        adam_update.v = np.zeros_like(gradient)
    # Update moving averages
    adam_update.m = beta1 * adam_update.m + (1 - beta1) * gradient
    adam_update.v = beta2 * adam_update.v + (1 - beta2) * gradient**2
    # Bias correction
    m_hat = adam_update.m / (1 - beta1**iteration)
    v_hat = adam_update.v / (1 - beta2**iteration)
    # Compute the update step
    update = -alpha * m_hat / (np.sqrt(v_hat) + epsilon)
    return update, adam_update.m, adam_update.v
def gradient_descent_adam(f, grad_f, x0, alpha, beta1, beta2, max_iterations=1000, tol=1e-6):
    """
    Perform gradient descent using Adam.
    Parameters:
    - f: The function to minimize.
    - grad_f: The gradient of the function.
    - x0: Initial point (numpy array).
    - alpha: Learning rate.
    - beta1: Decay rate for the moving average of gradients.
    - beta2: Decay rate for the moving average of squared gradients.
    - max_iterations: Maximum number of iterations.
    - tol: Tolerance for stopping criterion.
    Returns:
    - x: The final point after optimization.
    - history: List of function values at each iteration.
    """
    x = x0  # Initialize the current point
    history = []  # Store the function values at each iteration

    for iteration in range(1, max_iterations + 1):
        gradient = grad_f(x)  # Compute the gradient at the current point
```

```
        update, _, _ = adam_update(gradient, alpha, beta1, beta2, iteration=iteration)  # Compute the update step
        x = x + update  # Update the current point
        # Store the function value for plotting/convergence analysis
        history.append(f(x))
        # Check for convergence (stop if the gradient is very small)
        if np.linalg.norm(gradient) < tol:
            print(f"Converged after {iteration} iterations.")
            break
    return x, history
# ## Part b 1 rms prop
# ### RMSProp Implementation
import numpy as np
import matplotlib.pyplot as plt
# Function 1: f1(x, y) = 6(x-1)^4 + 8(y-2)^2
def f1(x):
    return 6 * (x[0] - 1) ** 4 + 8 * (x[1] - 2) ** 2
def grad_f1(x):
    return np.array([24 * (x[0] - 1) ** 3, 16 * (x[1] - 2)])
# Function 2: f2(x, y) = max(x-1,0) + 8|y-2|
def f2(x):
    return np.maximum(x[0] - 1, 0) + 8 * np.abs(x[1] - 2)
def grad_f2(x):
    return np.array([
        1.0 if x[0] > 1 else 0.0,  # Derivative of max(x-1, 0)
        8 * np.sign(x[1] - 2)  # Derivative of 8|y-2|
    ])
# RMSProp Update Rule
def rmsprop_update(gradient, cache, alpha, beta, epsilon=1e-8):
    cache = beta * cache + (1 - beta) * gradient**2
    update = -alpha * gradient / (np.sqrt(cache) + epsilon)
    return update, cache
# RMSProp Gradient Descent Function
def gradient_descent_rmsprop(f, grad_f, x0, alpha, beta, max_iterations=1000, tol=1e-6):
    x = np.array(x0, dtype=float)
    history = []
    cache = np.zeros_like(x)  # Initialize cache
    for iteration in range(max_iterations):
        gradient = grad_f(x)
        update, cache = rmsprop_update(gradient, cache, alpha, beta)
        x += update
        history.append(f(x))
        # Print values every 100 iterations
        if iteration % 100 == 0:
            print(f"Iteration {iteration}: f(x, y) = {f(x):.4f}, x = {x[0]:.4f}, y = {x[1]:.4f}")
        # Stop if gradient norm is small
        if np.linalg.norm(gradient) < tol:
            print(f"Converged at Iteration {iteration}: f(x, y) = {f(x):.4f}, x = {x[0]:.4f}, y = {x[1]:.4f}")
            break

    return x, history
# Parameters for RMSProp
alphas = [0.01, 0.05, 0.1]  # Learning rates to test
betas = [0.25, 0.9]  # Decay rates
x0_f1 = [0.0, 0.0]  # Initial point for f1
x0_f2 = [0.5, 3.0]  # Initial point for f2
# Run RMSProp and plot results for Function 1 and Function 2
functions = [f1, f2]
grad_functions = [grad_f1, grad_f2]
initial_points = [x0_f1, x0_f2]
function_names = ["Function 1", "Function 2"]
for i, (f, grad_f, x0, name) in enumerate(zip(functions, grad_functions, initial_points, function_names)):
    plt.figure(figsize=(12, 6))
    for beta in betas:
        for alpha in alphas:
            _, history = gradient_descent_rmsprop(f, grad_f, x0, alpha, beta)
            plt.plot(history, label=f'α={alpha}, β={beta}')
    plt.xlabel("Iteration")
    plt.ylabel("Function Value")
    plt.title(f"RMSProp Convergence for {name}")
    plt.legend()
```

```
    plt.grid(True)
    plt.show()
# Generate Contour Plots with Trajectory
def plot_contour(f, x_history, y_history, name):
    x_range = np.linspace(-1, 3, 400)
    y_range = np.linspace(0, 5, 400)
    x_mesh, y_mesh = np.meshgrid(x_range, y_range)
    # Compute function values over the grid
    if name == "Function 1":
        z_mesh = 6 * (x_mesh - 1) ** 4 + 8 * (y_mesh - 2) ** 2
    else:
        z_mesh = np.maximum(x_mesh - 1, 0) + 8 * np.abs(y_mesh - 2)
    plt.figure(figsize=(10, 8))
    cp = plt.contourf(x_mesh, y_mesh, z_mesh, levels=50, cmap='viridis', alpha=0.6)
    plt.colorbar(cp)
    plt.plot(x_history, y_history, 'ro-', label='Optimizer Path')  # Plot trajectory
    plt.title(f'Contour Plot of {name} with RMSProp Trajectories')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()
    plt.show()
for i, (f, grad_f, x0, name) in enumerate(zip(functions, grad_functions, initial_points, function_names)):
    for beta in betas:
        for alpha in alphas:
            x_history, y_history = [], []
            x = np.array(x0, dtype=float)
            cache = np.zeros_like(x)  # Reset cache
            for iteration in range(500):  # Limit iterations for better visualization
                gradient = grad_f(x)
                update, cache = rmsprop_update(gradient, cache, alpha, beta)
                x += update
                x_history.append(x[0])
                y_history.append(x[1])

            # Plot Contour
            plot_contour(f, x_history, y_history, name)
# ### part b 2 heavy ball
import numpy as np
import matplotlib.pyplot as plt
# Function 1: f1(x, y) = 6(x-1)^4 + 8(y-2)^2
def f1(x):
    return 6 * (x[0] - 1) ** 4 + 8 * (x[1] - 2) ** 2
def grad_f1(x):
    return np.array([24 * (x[0] - 1) ** 3, 16 * (x[1] - 2)])
# Function 2: f2(x, y) = max(x-1,0) + 8|y-2|
def f2(x):
    return np.maximum(x[0] - 1, 0) + 8 * np.abs(x[1] - 2)
def grad_f2(x):
    return np.array([
        1.0 if x[0] > 1 else 0.0,
        8 * np.sign(x[1] - 2)
    ])
# Heavy Ball Update Rule
def heavy_ball_update(grad, velocity, alpha, beta):
    velocity = beta * velocity - alpha * grad
    return velocity, velocity
# Heavy Ball Gradient Descent with Improved Convergence Checking
def gradient_descent_heavy_ball(f, grad_f, x0, alpha, beta, max_iterations=1000, tol=1e-6):
    x = np.array(x0, dtype=float)
    history = []
    velocity = np.zeros_like(x)  # Initialize velocity
    f_prev = f(x)  # Store previous function value
    for iteration in range(max_iterations):
        grad = grad_f(x)
        update, velocity = heavy_ball_update(grad, velocity, alpha, beta)
        x += update
        f_current = f(x)
        history.append(f_current)
        # Print values every 100 iterations
        if iteration % 100 == 0:
```

```
            print(f"Iteration {iteration}: f(x, y) = {f_current:.4f}, x = {x[0]:.4f}, y = {x[1]:.4f}")
        # Stop if gradient norm is small OR function value stops decreasing
        if np.linalg.norm(grad) < tol or abs(f_current - f_prev) < 1e-8:
            print(f"Converged at Iteration {iteration}: f(x, y) = {f_current:.4f}, x = {x[0]:.4f}, y = {x[1]:.4f}")
            break
        f_prev = f_current  # Update previous function value
    return x, history
# Parameters for Heavy Ball with Reduced Momentum for Function 2
alphas = [0.001, 0.005, 0.01]  # Learning rates
betas_f1 = [0.25, 0.9]  # Momentum factors for Function 1
betas_f2 = [0.1, 0.5]  # Reduced momentum for Function 2 to prevent oscillations
x0_f1 = [0.0, 0.0]  # Initial point for f1
x0_f2 = [0.5, 3.0]  # Initial point for f2
# Run Heavy Ball and plot results for Function 1
plt.figure(figsize=(12, 6))
for beta in betas_f1:
    for alpha in alphas:
        _, history = gradient_descent_heavy_ball(f1, grad_f1, x0_f1, alpha, beta)
        plt.plot(history, label=f'α={alpha}, β={beta}')
plt.xlabel("Iteration")
plt.ylabel("Function Value")
plt.title("Heavy Ball Convergence for Function 1")
plt.legend()
plt.grid(True)
plt.show()
# Run Heavy Ball and plot results for Function 2 (Reduced Momentum)
plt.figure(figsize=(12, 6))
for beta in betas_f2:  # Use smaller momentum for stability
    for alpha in alphas:
        _, history = gradient_descent_heavy_ball(f2, grad_f2, x0_f2, alpha, beta)
        plt.plot(history, label=f'α={alpha}, β={beta}')
plt.xlabel("Iteration")
plt.ylabel("Function Value")
plt.title("Heavy Ball Convergence for Function 2 (Adjusted β)")
plt.legend()
plt.grid(True)
plt.show()
# # contoutr plot
# Generate Contour Plots with Trajectory
def plot_contour(f, x_history, y_history, name):
    x_range = np.linspace(-1, 3, 400)
    y_range = np.linspace(0, 5, 400)
    x_mesh, y_mesh = np.meshgrid(x_range, y_range)
    # Compute function values over the grid
    if name == "Function 1":
        z_mesh = 6 * (x_mesh - 1) ** 4 + 8 * (y_mesh - 2) ** 2
    else:
        z_mesh = np.maximum(x_mesh - 1, 0) + 8 * np.abs(y_mesh - 2)
    plt.figure(figsize=(10, 8))
    cp = plt.contourf(x_mesh, y_mesh, z_mesh, levels=50, cmap='viridis', alpha=0.6)
    plt.colorbar(cp)
    plt.plot(x_history, y_history, 'ro-', label='Optimizer Path')  # Plot trajectory
    plt.title(f'Contour Plot of {name} with Heavy Ball Trajectories')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()
    plt.show()
# Run Heavy Ball for contour plotting and collect history
for i, (f, grad_f, x0, name) in enumerate(zip([f1, f2], [grad_f1, grad_f2], [[0, 0], [0.5, 3.0]], ["Function 1", "Function 2"])):
    for beta in [0.25, 0.9]:  # Using both momentum values
        for alpha in [0.001, 0.005, 0.01]:
            x_history, y_history = [], []
            x = np.array(x0, dtype=float)
            velocity = np.zeros_like(x)  # Reset velocity
                for iteration in range(300):  # Limit iterations for better visualization
                grad = grad_f(x)
                update, velocity = heavy_ball_update(grad, velocity, alpha, beta)
                x += update
                x_history.append(x[0])
                y_history.append(x[1])
```

```
        # Plot Contour
        plot_contour(f, x_history, y_history, name)
# # Step Size vs Iteration
def plot_step_size(f, grad_f, x0, alpha, beta, name):
    x = np.array(x0, dtype=float)
    velocity = np.zeros_like(x)
    step_sizes = []
    for iteration in range(1000):
        grad = grad_f(x)
        update, velocity = heavy_ball_update(grad, velocity, alpha, beta)
        x += update
        step_sizes.append(np.linalg.norm(update))  # Compute step size
    plt.plot(step_sizes, label=f'α={alpha}, β={beta}')
for i, (f, grad_f, x0, name) in enumerate(zip([f1, f2], [grad_f1, grad_f2], [[0, 0], [0.5, 3.0]], ["Function 1", "Function 2"])):
    plt.figure(figsize=(10, 6))
    for beta in [0.25, 0.9]:  # Using both momentum values
        for alpha in [0.001, 0.005, 0.01]:
            plot_step_size(f, grad_f, x0, alpha, beta, name)
    plt.xlabel("Iteration")
    plt.ylabel("Step Size (Update Magnitude)")
    plt.title(f"Step Size vs Iteration for {name}")
    plt.legend()
    plt.grid(True)
    plt.show()
# ## adam
import numpy as np
import matplotlib.pyplot as plt
# Adam Update Rule
def adam_update(grad, m, v, t, alpha, beta1, beta2, epsilon=1e-8):
    """Performs an Adam optimization update step."""
    m = beta1 * m + (1 - beta1) * grad  # First moment estimate
    v = beta2 * v + (1 - beta2) * (grad ** 2)  # Second moment estimate
    # Bias correction
    m_hat = m / (1 - beta1 ** t)
    v_hat = v / (1 - beta2 ** t)
    update = -alpha * m_hat / (np.sqrt(v_hat) + epsilon)  # Compute update step
    return update, m, v  # Return update, new moment estimates
# Adam Gradient Descent Implementation
def gradient_descent_adam(f, grad_f, x0, alpha, beta1, beta2, max_iterations=2000, tol=1e-6):
    x = np.array(x0, dtype=float)
    history = []
    m, v = np.zeros_like(x), np.zeros_like(x)  # Initialize moment estimates
    f_prev = f(x)  # Store previous function value
    for t in range(1, max_iterations + 1):
        grad = grad_f(x)
        update, m, v = adam_update(grad, m, v, t, alpha, beta1, beta2)
        x += update
        f_current = f(x)
        history.append(f_current)
        # Print values every 50 iterations
        if t % 50 == 0:
            print(f"Iteration {t}: f(x, y) = {f_current:.4f}, x = {x[0]:.4f}, y = {x[1]:.4f}")
        # Stop if gradient norm is small OR function value stops decreasing
        if np.linalg.norm(grad) < 1e-4 or abs(f_current - f_prev) < 1e-6:
            print(f"Converged at Iteration {t}: f(x, y) = {f_current:.4f}, x = {x[0]:.4f}, y = {x[1]:.4f}")
            break
        f_prev = f_current  # Update previous function value
    return x, history
# Parameters for Adam
alphas = [0.001, 0.005, 0.01]  # Learning rates
beta1_values = [0.9, 0.95]  # First moment decay rates
beta2_values = [0.99, 0.999]  # Second moment decay rates
x0_f1 = [0.0, 0.0]  # Initial point for f1
x0_f2 = [0.5, 3.0]  # Initial point for f2
# Run Adam and plot results for Function 1
plt.figure(figsize=(12, 6))
for beta1 in beta1_values:
    for beta2 in beta2_values:
        for alpha in alphas:
            _, history = gradient_descent_adam(f1, grad_f1, x0_f1, alpha, beta1, beta2)
```

```
            plt.plot(history, label=f'α={alpha}, β1={beta1}, β2={beta2}')
plt.xlabel("Iteration")
plt.ylabel("Function Value")
plt.title("Adam Convergence for Function 1")
plt.legend()
plt.grid(True)
plt.show()
# Run Adam and plot results for Function 2
plt.figure(figsize=(12, 6))
for beta1 in beta1_values:
    for beta2 in beta2_values:
        for alpha in alphas:
            _, history = gradient_descent_adam(f2, grad_f2, x0_f2, alpha, beta1, beta2)
            plt.plot(history, label=f'α={alpha}, β1={beta1}, β2={beta2}')
plt.xlabel("Iteration")
plt.ylabel("Function Value")
plt.title("Adam Convergence for Function 2")
plt.legend()
plt.grid(True)
plt.show()
# ## contour plot
# Generate Contour Plots with Trajectory
def plot_contour_adam(f, x_history, y_history, name):
    x_range = np.linspace(-1, 3, 400)
    y_range = np.linspace(0, 5, 400)
    x_mesh, y_mesh = np.meshgrid(x_range, y_range)
    # Compute function values over the grid
    if name == "Function 1":
        z_mesh = 6 * (x_mesh - 1) ** 4 + 8 * (y_mesh - 2) ** 2
    else:
        z_mesh = np.maximum(x_mesh - 1, 0) + 8 * np.abs(y_mesh - 2)
    plt.figure(figsize=(10, 8))
    cp = plt.contourf(x_mesh, y_mesh, z_mesh, levels=50, cmap='viridis', alpha=0.6)
    plt.colorbar(cp)
    plt.plot(x_history, y_history, 'ro-', label='Optimizer Path')  # Plot trajectory
    plt.title(f'Contour Plot of {name} with Adam Trajectories')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()
    plt.show()
# Run Adam for contour plotting and collect history
for i, (f, grad_f, x0, name) in enumerate(zip([f1, f2], [grad_f1, grad_f2], [[0, 0], [0.5, 3.0]], ["Function 1", "Function 2"])):
    for beta1 in beta1_values:
        for beta2 in beta2_values:
            for alpha in alphas:
                x_history, y_history = [], []
                x = np.array(x0, dtype=float)
                m, v = np.zeros_like(x), np.zeros_like(x)
                for t in range(300):  # Limit iterations for better visualization
                    grad = grad_f(x)
                    update, m, v = adam_update(grad, m, v, t + 1, alpha, beta1, beta2)
                    x += update
                    x_history.append(x[0])
                    y_history.append(x[1])
                # Plot Contour
                plot_contour_adam(f, x_history, y_history, name)
# ### Part c (i)
### Implementing the ReLU Function and its Derivative
import numpy as np
import matplotlib.pyplot as plt
# Define ReLU Function
def relu(x):
    return np.maximum(0, x)
# Define ReLU Gradient using Heaviside function
def grad_relu(x):
    return np.heaviside(x, 0)  # 0 for x < 0, 1 for x > 0
### Apply RMSProp to ReLU
# RMSProp Update Function
def rmsprop_update(grad, s, alpha, beta, epsilon=1e-8):
    s = beta * s + (1 - beta) * grad**2
```

```
    update = -alpha * grad / (np.sqrt(s) + epsilon)
    return update, s
# Apply RMSProp to ReLU
def rmsprop_relu(x0, alpha=0.01, beta=0.9, max_iterations=100):
    x = x0
    s = 0  # Initialize moving average
    history = []
    for iteration in range(max_iterations):
        grad = grad_relu(x)
        update, s = rmsprop_update(grad, s, alpha, beta)
        x += update
        history.append(x)
        if iteration % 10 == 0:
            print(f"Iteration {iteration}: x = {x:.4f}")
        # If gradient is 0 (ReLU is flat), stop updating
        if grad == 0:
            print(f"Converged at Iteration {iteration}: x = {x:.4f}")
            break
    return history
# Run RMSProp for different initial conditions
x_vals = [-1, 1, 100]
plt.figure(figsize=(8, 6))
for x0 in x_vals:
    history = rmsprop_relu(x0)
    plt.plot(history, label=f"Initial x={x0}")
plt.xlabel("Iteration")
plt.ylabel("x value")
plt.title("RMSProp Optimization on ReLU")
plt.legend()
plt.grid()
plt.show()
# ### Apply Heavy Ball to ReLU
# Heavy Ball Update Function
def heavy_ball_update(grad, velocity, alpha, beta):
    velocity = beta * velocity - alpha * grad
    update = velocity
    return update, velocity
# Apply Heavy Ball to ReLU
def heavy_ball_relu(x0, alpha=0.01, beta=0.9, max_iterations=100):
    x = x0
    velocity = 0  # Initialize velocity
    history = []
    for iteration in range(max_iterations):
        grad = grad_relu(x)
        update, velocity = heavy_ball_update(grad, velocity, alpha, beta)
        x += update
        history.append(x)
        if iteration % 10 == 0:
            print(f"Iteration {iteration}: x = {x:.4f}")
        if grad == 0:
            print(f"Converged at Iteration {iteration}: x = {x:.4f}")
            break
    return history
plt.figure(figsize=(8, 6))
for x0 in x_vals:
    history = heavy_ball_relu(x0)
    plt.plot(history, label=f"Initial x={x0}")
plt.xlabel("Iteration")
plt.ylabel("x value")
plt.title("Heavy Ball Optimization on ReLU")
plt.legend()
plt.grid()
plt.show()
### Apply Adam to ReLU
# Adam Update Function
def adam_update(grad, m, v, t, alpha, beta1, beta2, epsilon=1e-8):
    m = beta1 * m + (1 - beta1) * grad
    v = beta2 * v + (1 - beta2) * grad**2
    m_hat = m / (1 - beta1 ** t)
    v_hat = v / (1 - beta2 ** t)
```

```
    update = -alpha * m_hat / (np.sqrt(v_hat) + epsilon)
    return update, m, v
# Apply Adam to ReLU
def adam_relu(x0, alpha=0.01, beta1=0.9, beta2=0.99, max_iterations=100):
    x = x0
    m, v = 0, 0
    history = []
    for t in range(1, max_iterations + 1):
        grad = grad_relu(x)
        update, m, v = adam_update(grad, m, v, t, alpha, beta1, beta2)
        x += update
        history.append(x)
        if t % 10 == 0:
            print(f"Iteration {t}: x = {x:.4f}")
        if grad == 0:
            print(f"Converged at Iteration {t}: x = {x:.4f}")
            break
    return history
plt.figure(figsize=(8, 6))
for x0 in x_vals:
    history = adam_relu(x0)
    plt.plot(history, label=f"Initial x={x0}")
plt.xlabel("Iteration")
plt.ylabel("x value")
plt.title("Adam Optimization on ReLU")
plt.legend()
plt.grid()
plt.show()
```