Assignment 2

Danielle Buggle 20333829

February 26, 2025

Functions downloaded from the PHP file:

```
f(x, y) = 5 * (x - 9)^4 + 6 * (y - 4)^2

f(x, y) = Max(x - 9, 0) + 6 * |y - 4|
```

Mathematical format:

$$f(x,y) = 5 \cdot (x-9)^4 + 6 \cdot (y-4)^2$$

$$f(x,y) = \max(x-9,0) + 6 \cdot |y-4|$$

Part A

(i) Polyak step size

```python
def polyak_step_size(f_x, f_star, grad_f_x):
    grad_norm_sq = np.dot(grad_f_x, grad_f_x)
    if grad_norm_sq == 0:  # Avoid division by zero
        return 0
    return (f_x - f_star) / grad_norm_sq
```

Listing 1: Function to obtain Polyak Step

The code above is based on the following equation:

$$\alpha = \frac{f(x) - f^*}{\nabla f(x)^T \nabla f(x)} \tag{1}$$

This equation is used to calculate the step size for the polyak algorithm. It is based on the difference between the current function value $f(x)$ and the optimal function value f^*. The difference is then divided by the sum of the derivatives squared, also known as the squared norm of the gradient.

This function takes three parameters

- `f_x` (current function value)
- `f_star` (optimal function value)
- `grad_f_x` (array of partial derivatives)

First the squared norm of the gradient is calculated using `np.dot()`, which is the same as `sum(grad_f_x ** 2)`. Finally, if this value is non-zero then the function returns the step size computed as (1).

(ii) RMSProp

```python
def rmsprop_update(x, grad_x, moving_avg, alpha_0, beta, epsilon=1e-8):
    grad_sq = grad_x ** 2
    moving_avg = beta * moving_avg + (1 - beta) * grad_sq

    moving_avg = np.array(moving_avg, dtype=np.float64)
    step_size = alpha_0 / (np.sqrt(moving_avg) + epsilon)
    x_new = x - step_size * grad_x
    return x_new, moving_avg
```

Listing 2: Function to perform RMSProp Step

RMSProp is an adaptive learning rate algorithm that helps improve convergence by calculating the step size for each parameter individually. The code above represents my implementation of the update step for this algorithm.

This function begins by calculating the squared gradient. After that, the moving average of the squared gradients needs to be calculated. The moving average is calculated by following the equation below:

$$sum = \beta sum + (1-\beta)\frac{df}{dx}(x_t)^2 \tag{2}$$

The new squared gradient is blended into the moving average using the decay rate β. Higher β values give more weight to past gradients whereas smaller values cause the step size to focus more on recent values.

`moving_avg = np.array(moving_avg, dtype=np.float64)` is used to ensure numerical stability and prevent unintentional data type issues. Following that the step size is calculated by dividing the base learning rate α_0 by the **square root** of the moving average plus a small constant ε to prevent division by 0. This is done by following this formula:

$$\alpha_{t+1} = \frac{\alpha_0}{\sqrt{sum} + \varepsilon} \tag{3}$$

Finally, the next step is calculated using the new step size and the function returns the updated parameter values with the new step and the new moving average.

(iii) Heavy Ball

```python
def heavy_ball_step(x, z, grad_x, alpha, beta):
    z_new = beta * z + alpha * grad_x
    x_new = x - z_new
    return x_new, z_new
```

Listing 3: Function to perform Heavy Ball Step

This function begins by updating the momentum term `z` by combining the past momentum βz with the sum of the past gradients $\alpha\nabla f(x_t)$. Similar to RMSProp high β values mean past updates have more of an influence on smaller values causing the step size to focus more on recent values. This all follows the following equation:

$$z_{t+1} = \beta z_t + \alpha\nabla f(x_t) \tag{4}$$

Finally, the function returns the updated parameter values with the new step and the new momentum term to be used in the next iteration.

(iv) Adam

```python
def adam_step(x, m, v, grad_x, alpha, beta1, beta2, epsilon, t):
    m_new = beta1 * m + (1 - beta1) * grad_x
    v_new = beta2 * v + (1 - beta2) * (grad_x ** 2)

    # Bias correction
    m_hat = m_new / (1 - beta1 ** t)
    v_hat = v_new / (1 - beta2 ** t)
    v_hat = np.array(v_hat, dtype=np.float64)

    step_size = alpha * (m_hat / (np.sqrt(v_hat) + epsilon))
    x_new = x - step_size

    return x_new, m_new, v_new
```

Listing 4: Function to perform Adam Step

The function begins by calculating the first momentum term `m_new` which is the running average of the gradient $\nabla f(x_t)$. This follows this calculation:

$$m_{t+1} = \beta_1 m_t + (1-\beta_1)\nabla f(x_t) \tag{5}$$

Then the second momentum term is calculated similarly to RMSProp using (2) where `v_new` is a moving average of squared gradients. Following that the bias correction needs to be performed since the `m` and `v` are initialized at zero, they are biased towards zero in the early iterations. The bias correction step adjusts for this by dividing by $1-\beta^t$. See below for equations for `m_hat` and `v_hat`.

$$\hat{m} = \frac{m_{t+1}}{1-\beta_1^t}, \quad \hat{v} = \frac{v_{t+1}}{1-\beta_2^t} \tag{6}$$

Once again, `v_hat = np.array(v_hat, dtype=np.float64)` is used to ensure numerical stability and prevent unintentional data type issues. Finally, the learning rate α is scaled by the ratio of $\hat{m}$ to the square root of $\hat{v}$ and then the new x parameter is calculated. The new x parameter and updated momentum terms are returned for the next iteration.

Part B

(i) α and β in RMSProp

Figures 1, 2, 6 and 7 show the convergence rate when α values are very small and for both β values RMSProp is quite a flat line, indicating a very slow convergence rate for both functions. The small learning rate causes the algorithm to take tiny steps at each iteration resulting in very slow progress.

Figures 3 and 8 have a moderate learning rate of $\alpha = 0.01$, showing that the algorithm is making better progress in comparison to figures 1 and 2, however, it still doesn't converge quickly, only reaching a function value of approximately 240 for $\beta = 0.25$ and 220 $\beta = 0.9$ for the polynomial function. When $\beta = 0.9$ i.e. when β is larger, the algorithm adapts more quickly by giving more weight to past gradients, resulting in a faster initial decrease.

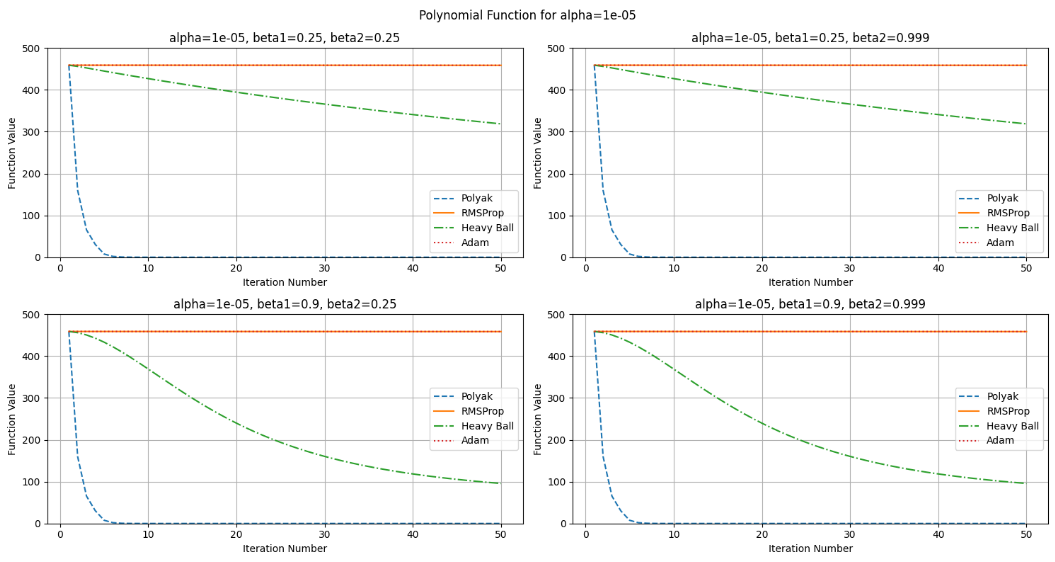

Figure 1: Subplots of function values vs iterations when $\alpha = 0.00001$

Looking at figures 4 and 5 the learning rate is much larger meaning our algorithm will take bigger steps and converge to the minimum. The larger the learning rate the quicker the convergence appears to be. For the polynomial function, both α values when β is large causes the function to initially decrease very fast but it ultimately takes longer to converge to the minimum as it takes longer to stabilise. For the ReLu function in figures 9 and 10 it performs similarly to the polynomial function and takes larger steps converging faster; however, with α too large it can cause oscillations even with varying β values.

Ultimately moderate learning rates paired with higher momentum terms often result in a faster convergence in the beginning but take a bit longer to stabilise. For higher α values a lower β term might be more beneficial. Using a very small learning rate will cause the convergence rate to be slow. This highlights the importance of parameter tuning to achieve the best performance for RMSProp.

(ii) α and β in HeavyBall

For the polynomial function, from figures 1 and 2 it appears that the smaller alpha values are converging to the minimum, with $\alpha = 0.0001$ performing slightly better. Paired with a higher beta value, in both figures, $\beta = 0.9$ causes the algorithm to converge faster. However for the ReLu function in figures 6 and 7, they all perform very similarly with flat lines converging very slowly to the minimum.

For the polynomial function in figure 3, $\alpha = 0.01$ and $\beta = 0.25$ is our optimal solution, causing the algorithm to converge extremely quickly. However, when $\beta = 0.9$ with this α value, it decreases extremely quickly for the first few iterations but then jumps back upwards causing the algorithm to diverge. This is the same for $\alpha = 0.1$ and $\alpha = 0.5$ for both β values, as can be seen in figures 4 and 5.

From figures 8, 9 and 10 the larger α values don't cause the method to diverge, it initially decreases to the minimum quite fast; however, oscillations appear as α gets larger. The reason these oscillations are occurring is because of the sharp transition at the ReLU threshold and the piecewise gradient behaviour. The piecewise nature of the ReLU function does not allow smooth movement towards the minimum causing the spikes and the algorithm to jump back and forth.

The heavy ball algorithm takes the weighted average of the terms. This means that a smaller β causes the algorithm to forget the previous momentum faster. Sometimes this can make the algorithm unstable. When β is

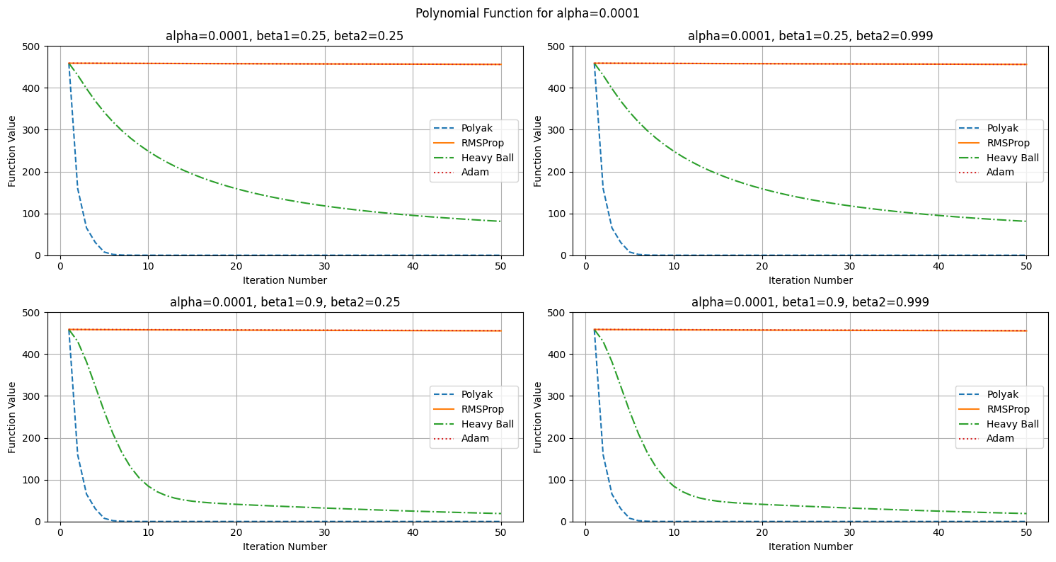

Figure 2: Subplots of function values vs iterations when $\alpha = 0.0001$

large the algorithm retains more information from previous steps, which can help it maintain its direction over time. However, if the learning rate is too high, this can cause it to overshoot the optimal solution, leading to divergence.

(iii) α, β_1 and β_2 in Adam

Figures 1, 2, 6 and 7 show the convergence rate when α values are very small and for all β values Adam is quite a flat line, indicating a very slow convergence rate. β_1 and β_2 have no impact, α is too small to make any significant progress. Figures 3 and 8 show increasing α causes the Adam algorithm to perform slightly better, however, the varying β_1 and β_2 values all seem to perform pretty similarly.

Looking at figure 4 with a larger α value the differences in varying β_1 and β_2 start to emerge. For $\beta_1 = 0.25$ and $\beta_2 = 0.25$, it performs quite well converging in approximately 25 iterations however we can see increasing β_2 causes it to take slightly longer. Looking at our bottom row of the subplots having $\beta_1 = 0.9$ and $\beta_2 = 0.25$ causes it to oscillate slightly but $\beta_1 = 0.9$ and $\beta_2 = 0.999$ makes it smooth out and converge. A high β_2 smooths updates but slows convergence, while a high β_1 may cause oscillations as it focuses more reliance on past gradients. In terms of the ReLu function in figure 9, it performs quite similarly to the polynomial function but it harder to identify any clear differences between the β values.

For our final figure 5, $\alpha = 0.5$ performs extremely well for $\beta_1 = 0.25$ and $\beta_2 = 0.25$ and has a very similar performance for $\beta_1 = 0.25$ and $\beta_2 = 0.999$. However, for $\beta_1 = 0.9$ and $\beta_2 = 0.25$, large oscillations are introduced, but increasing $\beta_2 = 0.999$ helps smooth out the algorithm and cause it to converge. Once again for our ReLu function in figure 10, it performs similarly to RMSProp with small oscillations at the bottom but ultimately smoothing out to converge after some time. However it is difficult to identify any difference in changing the β values.

Ultimately, for polynomial functions larger α values make the algorithm more sensitive to β_1 and β_2. A lower β_1 leads to faster but noisier updates, similar to low momentum in the Heavy Ball method. A higher β_1 adds momentum, which can help cnverge quicker but if paired with a small β_2 it may cause oscillations. Since β_2 controls adaptive learning rates like in RMSProp, a lower β_2 allows quicker adaptation but can increase

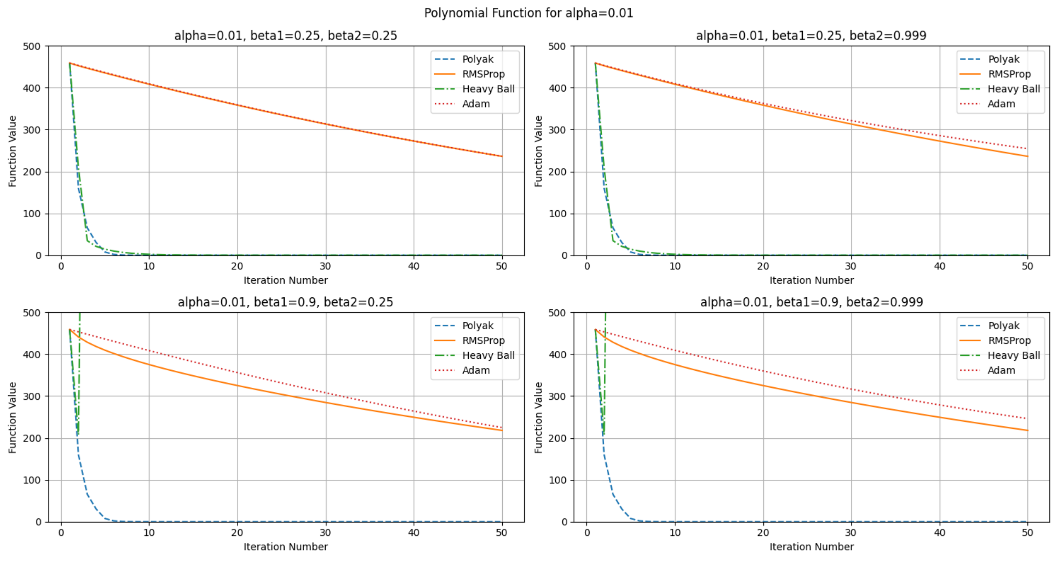

Figure 3: Subplots of function values vs iterations when $\alpha = 0.01$

noise, while a higher β_2 smooths updates but slows convergence. Fine-tuning these parameters is necessary to optimise this algorithm

Part C

$$f(x,y) = \max(0, x-9) + 6|y-4| \tag{7}$$

(i)

From figure 11, at $x = -1$, the function simplifies to $f(-1,y) = 6|y-4|$. The gradient with respect to x is zero because $\max(0,x-9)$ is flat for $x < 9$, while the gradient with respect to y is 6 if $y > 4$ and -6 if $y < 4$.

The optimisation algorithms still converge because of their update steps using momentum and adaptive learning rates for different gradient magnitudes. y moves toward 4 to minimize $6|y-4|$. Once y stabilises at 4, x stays at -1 because there is no gradient pushing it forward. The algorithms are converging as much as they can minimising as much as possible given the flat gradient at x.

(ii)

Figure 12, shows that the behaviour does not change for $x = -1$. The function behavior remains the same as when $x = -1$ because the gradient of $\max(0, x-9)$ is still zero for $x < 9$. When $x < 9$ the derivative of the function is 0 so when $x = 1$ the derivative remains 0 just like for $x = -1$.

(iii)

Figure 13, it appears that polyak initially overshoots and then slowly decreases which could be due to large step sizes in the beginning. The other algorithms all appear to decrease slowly but then start to plateau around function value of 85. Since the function has a piecewise structure, the algorithms seem to face difficulty escaping flat regions where the gradient is close to zero.

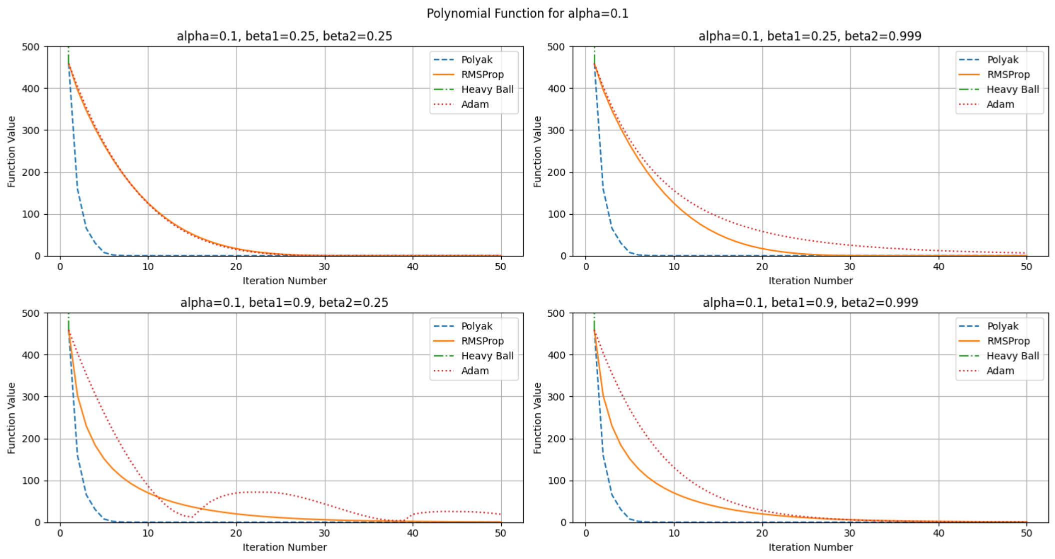

Figure 4: Subplots of function values vs iterations when $\alpha = 0.1$

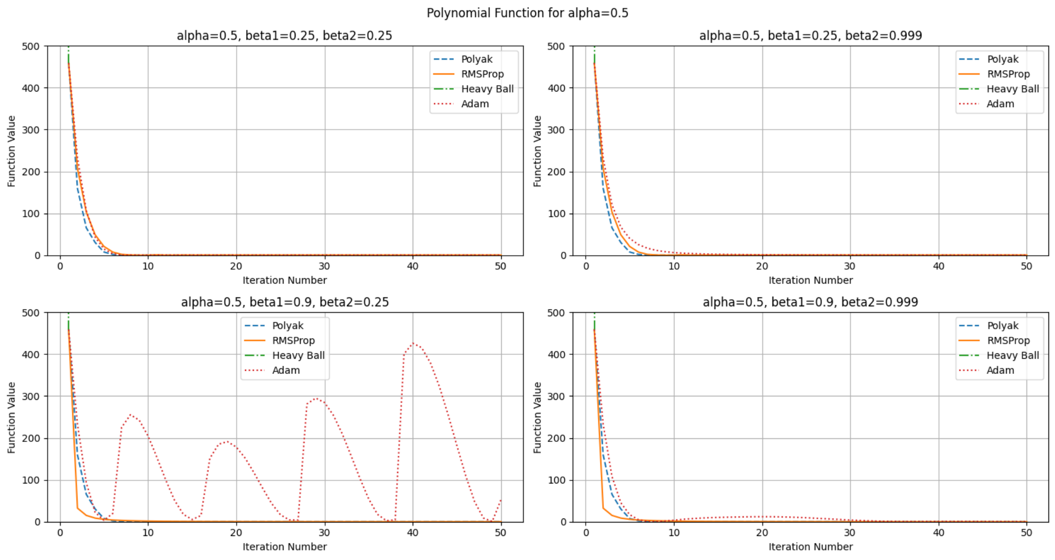

Figure 5: Subplots of function values vs iterations when $\alpha = 0.5$

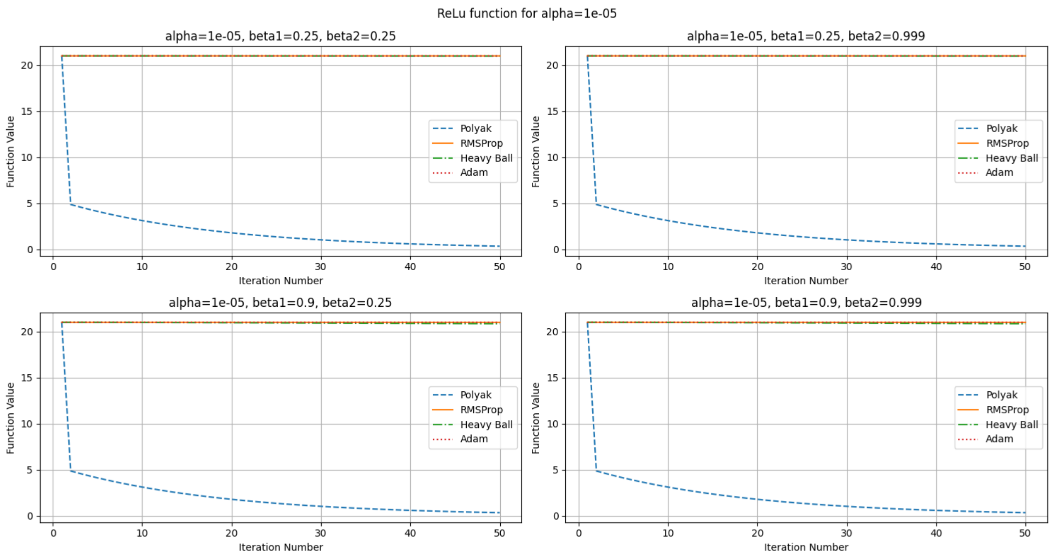

Figure 6: ReLu: Subplots of function values vs iterations when $\alpha = 0.00001$

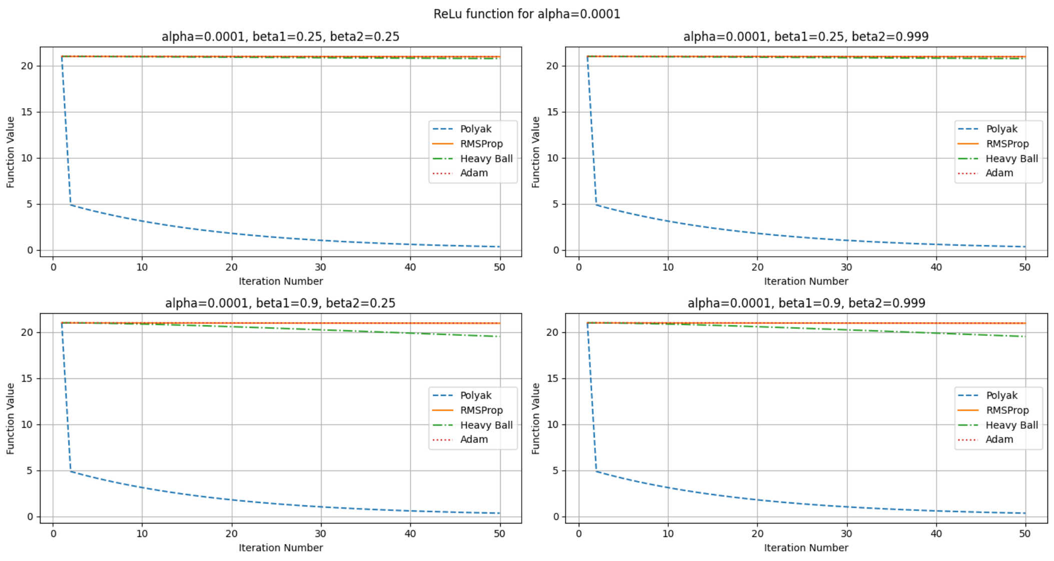

Figure 7: ReLu: Subplots of function values vs iterations when $\alpha = 0.0001$

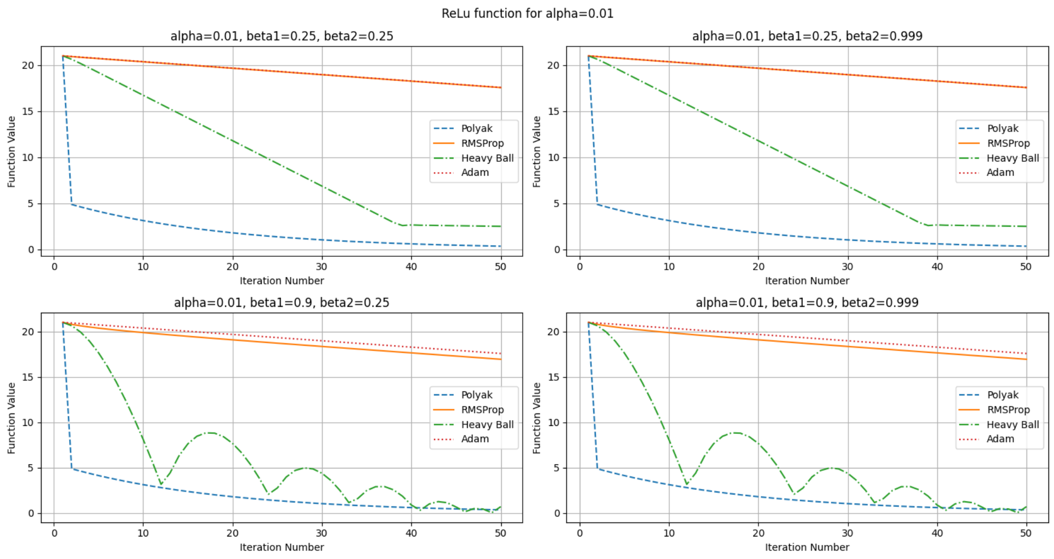

Figure 8: ReLu: Subplots of function values vs iterations when $\alpha = 0.01$

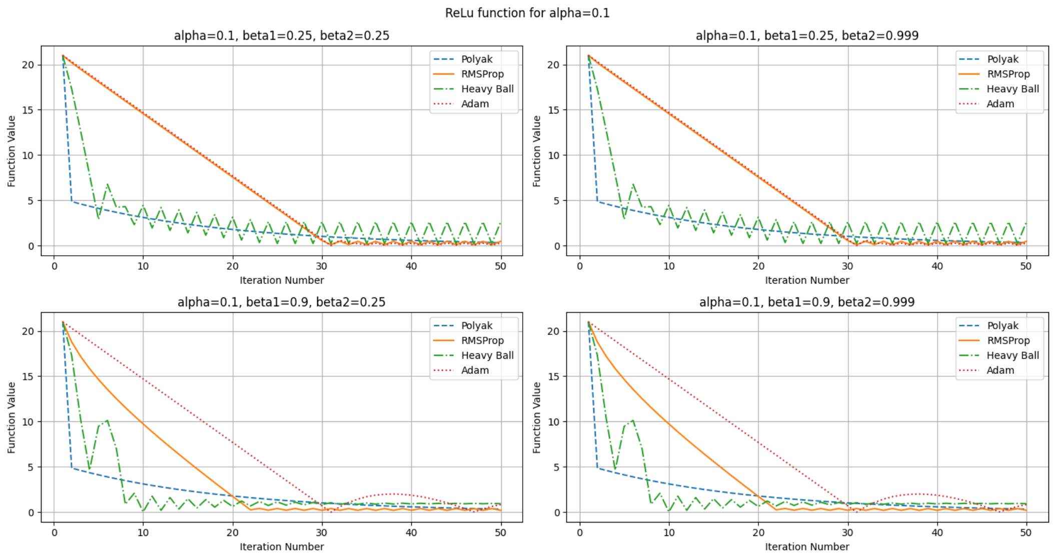

Figure 9: ReLu: Subplots of function values vs iterations when $\alpha = 0.1$

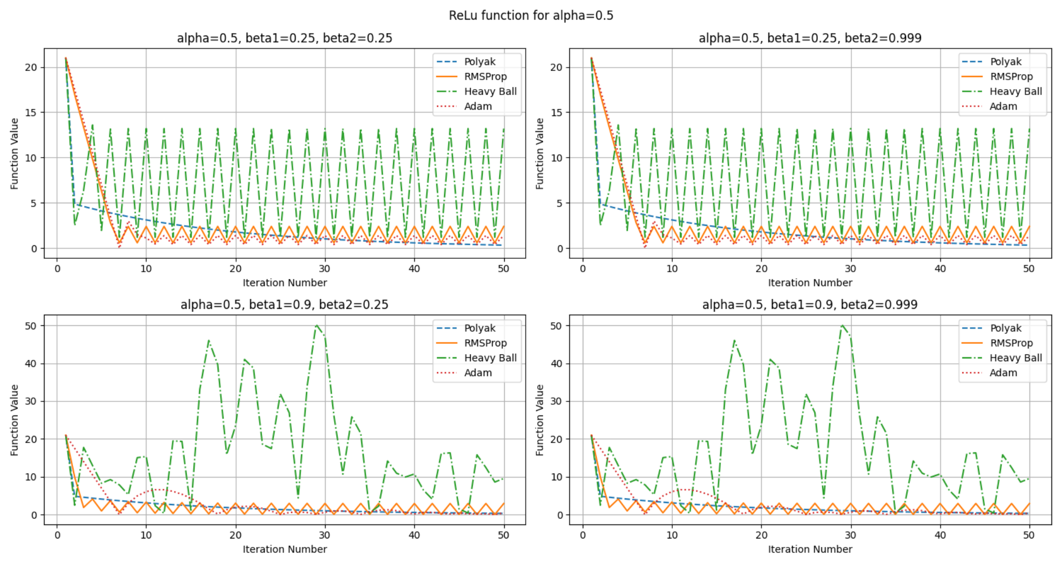

Figure 10: ReLu: Subplots of function values vs iterations when $\alpha = 0.5$

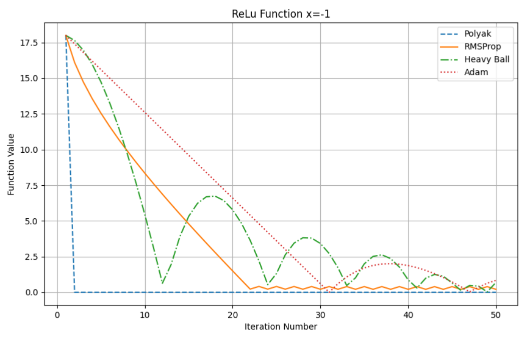

Figure 11: ReLu Function x=-1

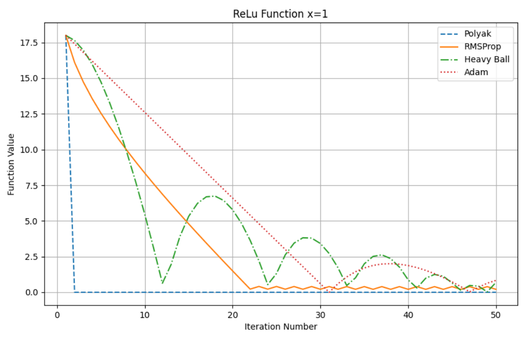

Figure 12: ReLu Function x=1

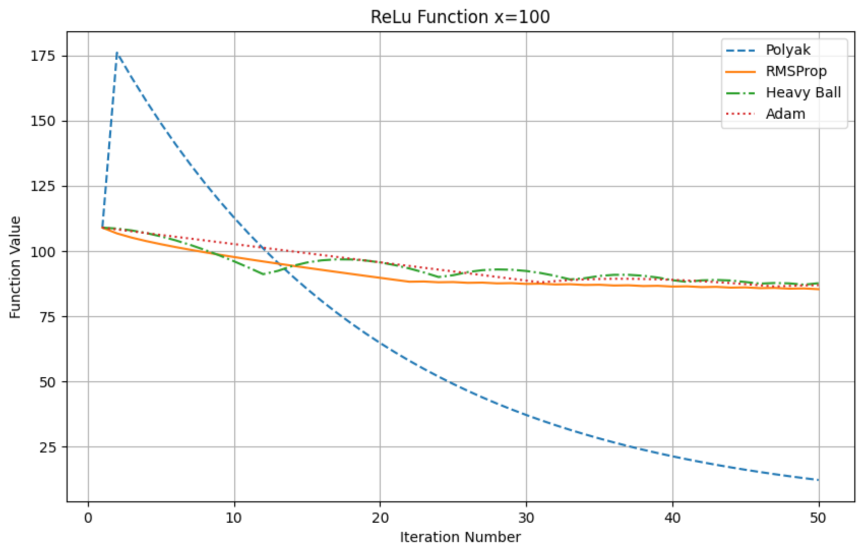

Figure 13: ReLu Function x=100

Code

derivative.py

```python
import numpy as np
import sympy


def derivative_expr(var, func):
    return sympy.diff(func, var)


def evaluate_derivative(vars, func, values):
    derivatives = [derivative_expr(var, func).simplify() for var in vars]  # Compute
         partial derivatives
    evaluated_gradient = []
    for d in derivatives:
        # Check if the derivative contains a Piecewise expression
        if isinstance(d, sympy.Piecewise):
            evaluated_value = d.subs(values).evalf()
        else:
            # Convert to numerical function and evaluate
            func_numeric = sympy.lambdify(vars, d, 'numpy')
            evaluated_value = func_numeric(*values.values())

        evaluated_gradient.append(evaluated_value)

    return np.array(evaluated_gradient)
```

grad_desc_updates.py

```python
from matplotlib import pyplot as plt
from derivative import *

"""
Part A
"""
def polyak_step_size(f_x, f_star, grad_f_x):
    """
    Calculate Polyak Step Size

    :param f_x: Current function value f(x).
    :param f_star: Optimal function value.
    :param grad_f_x: Array of partial derivatives.
    :return: Step size - alpha.
    """
    grad_norm_sq = np.dot(grad_f_x, grad_f_x)
    if grad_norm_sq == 0:  # Avoid division by zero
        return 0
    return (f_x - f_star) / grad_norm_sq


def rmsprop_update(x, grad_x, moving_avg, alpha_0, beta, epsilon=1e-8):
    """
    Perform update step using RMSProp

    :param x: Current parameter values
    :param grad_x: Gradient at x
    :param moving_avg: Moving average of squared gradients
    :param alpha_0: Base learning rate
    :param beta: Decay rate
    :param epsilon: Small constant to prevent division by zero
    :return: Parameter values with new step and the new moving average.
    """
    grad_sq = grad_x ** 2
    moving_avg = beta * moving_avg + (1 - beta) * grad_sq

    moving_avg = np.array(moving_avg, dtype=np.float64)
    step_size = alpha_0 / (np.sqrt(moving_avg) + epsilon)
    x_new = x - step_size * grad_x
    return x_new, moving_avg, step_size


def heavy_ball_step(x, z, grad_x, alpha, beta):
    """
    Computes next step using Heavy Ball method.

    :param x: Current parameter value.
    :param z: Momentum term.
    :param grad_x: Gradient at x.
    :param alpha: Learning rate.
    :param beta: Momentum coefficient (memory)
    :return: Updated x and momentum z.
    """
    z_new = beta * z + alpha * grad_x
    x_new = x - z_new
    return x_new, z_new


def adam_step(x, m, v, grad_x, alpha, beta1, beta2, epsilon, t):
    """
    Compute next step using Adam algorithm.

    :param x: Current parameter values
    :param m: First momentum term
    :param v: Second momentum term
    :param grad_x: Gradient at x
    :param alpha: Learning rate
    :param beta1: Decay rate for first momentum
    :param beta2: Decay rate for second momentum
    :param epsilon: Small constant to prevent division by zero
    :param t: Current time step
    :return: Updated x with new step, and updated momentum terms - m and v.
    """
    m_new = beta1 * m + (1 - beta1) * grad_x
    v_new = beta2 * v + (1 - beta2) * (grad_x ** 2)
    # Bias correction
    m_hat = m_new / (1 - beta1 ** t)
    v_hat = v_new / (1 - beta2 ** t)
    v_hat = np.array(v_hat, dtype=np.float64)

    step_size = alpha * (m_hat / (np.sqrt(v_hat) + epsilon))
    x_new = x - step_size

    return x_new, m_new, v_new, step_size


def polyak_optimization(func, start, f_star, num_iters):
    """
    Run Polyak optimization and track function values over iterations.

    :param func: Function to perform Polyak on.
    :param start: Initial (x, y) values.
    :param f_star: Optimal function value.
    :param num_iters: Number of iterations
    :return: Function values at each iteration.
    """
    vars = [x, y]  # Variables used in differentiation
    x_t, y_t = start
    f_values = []
    step_sizes = []

    for t in range(1, num_iters + 1):
        values = {x: x_t, y: y_t}
        f_x = func.subs(values).evalf()
        grad_f_x = evaluate_derivative(vars, func, values)
        alpha_t = polyak_step_size(f_x, f_star, grad_f_x)
        step_sizes.append(alpha_t)
        x_t -= alpha_t * grad_f_x[0]
        y_t -= alpha_t * grad_f_x[1]
        f_values.append(f_x)

    return f_values, step_sizes


def rmsprop_optimization(func, start, alpha_0, beta, num_iters, epsilon=1e-8):
    """
    Run RMSProp optimization and track function values over iterations.

    :param func: Function to perform RMSProp on.
    :param start: Initial (x, y) values.
    :param alpha_0: Learning rate.
    :param beta: Decay rate.
    :param num_iters: Number of iterations.
    :param epsilon: Small constant to prevent division by zero.
    :return: Function values at each iteration.
    """
    vars = [x, y]
```

```python
    x_t = np.array(start, dtype=np.float64)
    f_values = []
    E_g2 = np.zeros(2)
    step_sizes = []

    for _ in range(num_iters):
        values = {x: x_t[0], y: x_t[1]}
        f_x = func.subs(values).evalf()
        grad_f_x = evaluate_derivative(vars, func, values)

        x_t, E_g2, step_size = rmsprop_update(x_t, grad_f_x, E_g2, alpha_0, beta,
            epsilon)

        f_values.append(f_x)
        step_sizes.append(step_size)

    return f_values, step_sizes


def heavy_ball_optimization(func, start, alpha, beta, num_iters):
    """
    Run Heavy Ball optimization and track function values over iterations.

    :param func: Function to perform Heavy Ball on.
    :param start: Initial (x, y) values.
    :param alpha: Learning rate.
    :param beta: Momentum coefficient.
    :param num_iters: Number of iterations.
    :return:
    """
    vars = [x, y]
    x_t = np.array(start, dtype=np.float64)
    z_t = np.zeros(2)  # Initialize momentum term
    f_values = []
    step_sizes = []

    for _ in range(num_iters):
        values = {x: x_t[0], y: x_t[1]}
        f_x = func.subs(values).evalf()
        grad_f_x = evaluate_derivative(vars, func, values)

        x_t, z_t = heavy_ball_step(x_t, z_t, grad_f_x, alpha, beta)

        f_values.append(f_x)
        step_sizes.append(z_t)

    return f_values, step_sizes


def adam_optimization(func, start, alpha, beta1, beta2, num_iters, epsilon=1e-8):
    """
    Run Adam optimization and track function values over iterations.
    :param func: Function to perform Adam algorithm on.
    :param start: Initial (x, y) values.
    :param alpha: Learning rate.
    :param beta1: Decay rate for first momentum.
    :param beta2: Decay rate for second momentum.
    :param num_iters: Number of iterations.
    :param epsilon: Small constant for numerical stability.
    :return: Function values at each iteration.
    """
    vars = [x, y]
    x_t = np.array(start, dtype=np.float64)
    m_t = np.zeros(2)  # Initialize first moment
    v_t = np.zeros(2)  # Initialize second moment
    f_values = []
    step_sizes = []

    for t in range(1, num_iters + 1):
        values = {x: x_t[0], y: x_t[1]}
        f_x = func.subs(values).evalf()
        grad_f_x = evaluate_derivative(vars, func, values)

        x_t, m_t, v_t, step_size = adam_step(x_t, m_t, v_t, grad_f_x, alpha, beta1,
            beta2, epsilon, t)

        f_values.append(f_x)
        step_sizes.append(step_size)

    return f_values, step_sizes

"""
Part B
"""
x, y = sympy.symbols('x y', real=True)
f_expr = 5 * (x - 9) ** 4 + 6 * (y - 4) ** 2
f_star = 0

# Define parameter values to test
alpha_values = [0.00001, 0.0001, 0.01, 0.1, 0.5]
beta_values = [0.25, 0.9]
beta2_values = [0.25, 0.999]
start = (12, 1)
num_iters = 50

for alpha in alpha_values:
    fig, axes = plt.subplots(2, 2, figsize=(15, 8))
    for i, beta in enumerate(beta_values):
        for j, beta2 in enumerate(beta2_values):
            ax = axes[i, j]

            polyak_values, polyak_step_sizes = polyak_optimization(f_expr, start,
                f_star, num_iters)
            rmsprop_values, rmsprop_step_sizes = rmsprop_optimization(f_expr, start,
                 alpha, beta, num_iters)
            heavy_ball_values, heavyball_step_sizes = heavy_ball_optimization(f_expr
                , start, alpha, beta, num_iters)
            adam_values, adam_step_sizes = adam_optimization(f_expr, start, alpha,
                beta, beta2, num_iters)

            # Plot the optimisation results on the current subplot
            ax.plot(range(1, num_iters + 1), polyak_values, label="Polyak",
                linestyle="--")
            ax.plot(range(1, num_iters + 1), rmsprop_values, label="RMSProp",
                linestyle="-")
            ax.plot(range(1, num_iters + 1), heavy_ball_values, label="Heavy Ball",
                linestyle="-.")
            ax.plot(range(1, num_iters + 1), adam_values, label="Adam", linestyle=":
                ")

            # Set labels and title for each subplot
            ax.set_xlabel("Iteration Number")
            ax.set_ylabel("Function Value")
            ax.set_title(f"alpha={alpha}, beta1={beta}, beta2={beta2}")
            ax.legend()
            ax.set_ylim(0, 500)
            ax.grid()

    fig.suptitle(f"Polynomial Function for alpha={alpha}")
    plt.tight_layout()
    plt.savefig(f"images/partB/func/func_values/Comparison_of_{alpha}_beta.png")

f_expr = sympy.Max(x - 9, 0) + 6 * sympy.Abs(y - 4)

for alpha in alpha_values:
    fig, axes = plt.subplots(2, 2, figsize=(15, 8))
    for i, beta in enumerate(beta_values):
        for j, beta2 in enumerate(beta2_values):
            ax = axes[i, j]

            polyak_values, polyak_step_sizes = polyak_optimization(f_expr, start,
                f_star, num_iters)
            rmsprop_values, rmsprop_step_sizes = rmsprop_optimization(f_expr, start,
                 alpha, beta, num_iters)
            heavy_ball_values, heavyball_step_sizes = heavy_ball_optimization(f_expr
                , start, alpha, beta, num_iters)
            adam_values, adam_step_sizes = adam_optimization(f_expr, start, alpha,
                beta, beta2, num_iters)

            # Plot the optimisation results on the current subplot
            ax.plot(range(1, num_iters + 1), polyak_values, label="Polyak",
                linestyle="--")
            ax.plot(range(1, num_iters + 1), rmsprop_values, label="RMSProp",
                linestyle="-")
            ax.plot(range(1, num_iters + 1), heavy_ball_values, label="Heavy Ball",
                linestyle="-.")
            ax.plot(range(1, num_iters + 1), adam_values, label="Adam", linestyle=":
                ")
            # Set labels and title for each subplot
            ax.set_xlabel("Iteration Number")
            ax.set_ylabel("Function Value")
            ax.set_title(f"alpha={alpha}, beta1={beta}, beta2={beta2}")
            ax.legend()
            ax.grid()

    fig.suptitle(f"ReLu function for alpha={alpha}")
    plt.tight_layout()
    plt.savefig(f"images/partB/ReLu/ReLu_Comparison_of_{alpha}_beta.png")

"""
Part C
"""
f_expr = sympy.Max(x - 9, 0) + 6 * sympy.Abs(y - 4)
f_star = 0
num_iters = 50
start_values = [(-1, 1), (1, 1), (100, 1)]

for start in start_values:
    polyak_values, polyak_step_sizes = polyak_optimization(f_expr, start, f_star,
        num_iters)
    rmsprop_values, rmsprop_step_sizes = rmsprop_optimization(f_expr, start, 0.1,
        0.9, num_iters)
    heavy_ball_values, heavyball_step_sizes = heavy_ball_optimization(f_expr, start,
         0.01, 0.9, num_iters)
    adam_values, adam_step_sizes = adam_optimization(f_expr, start, 0.1, 0.9, 0.999,
         num_iters)

    plt.figure(figsize=(10, 6))
    plt.plot(range(1, num_iters + 1), polyak_values, label="Polyak", linestyle="--")
    plt.plot(range(1, num_iters + 1), rmsprop_values, label="RMSProp", linestyle="-"
        )
    plt.plot(range(1, num_iters + 1), heavy_ball_values, label="Heavy Ball",
        linestyle="-.")
    plt.plot(range(1, num_iters + 1), adam_values, label="Adam", linestyle=":")

    plt.xlabel("Iteration Number")
    plt.ylabel("Function Value")
    plt.title(f"ReLu Function x={start[0]}")
    plt.legend()
    plt.grid()
    plt.savefig(f"images/partC/Comparison of ReLu Function x={start[0]}.png")
```