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

Week 4 Assignment
CS7DS2 Optimisation for Machine Learning

Ujjayant Kadian (22330954)

February 27, 2025

Introduction

This report addresses the objectives of the Week 4 Optimisation assignment. In summary, we:

1. Implement four gradient descent variants: *Polyak step size*, *RMSProp*, *Heavy Ball* (momentum), and *Adam*.
2. Apply these algorithms to given test functions to study the impact of changing key parameters (learning rate α, momentum factors β, β_1, β_2).
3. Analyze the behavior of each method on the ReLU function $\max(0, x)$ under different initial conditions.

The primary goal is to demonstrate understanding of each optimization algorithm and to discuss how tuning hyperparameters affects convergence.

(a)(i) Polyak Step Size

The **Polyak step size** adaptively chooses the learning rate using knowledge of the difference between the current function value and the optimal value f^*. The update step size α_n at iteration n is given by:

$$\alpha_n = \frac{f(\theta_n) - f^*}{\|\nabla f(\theta_n)\|^2},$$

where f^* is the known (or estimated) minimum function value. This formula drives the iteration directly toward the minimum. In our implementation, we assume f^* is known for the test functions (both have a global minimum value of 0). The following code snippet computes the Polyak step size and updates the parameter:

```
# Polyak update: uses f_star (optimal f value) if known
grad_norm_sq = np.sum(grad**2)
if grad_norm_sq > 1e-12:                    # avoid division by zero
    alpha_n = (f_val - f_star) / grad_norm_sq
    theta = theta - alpha_n * grad
```

In this code, 'grad' is the gradient $\nabla f(\theta_n)$ at the current point, and 'f_val' is $f(\theta_n)$. We calculate α_n using the difference $(f\_val - f\_star)$ and the squared norm of the gradient. The parameter vector 'theta' is then updated by moving in the negative gradient direction with this step size. The check on 'grad_norm_sq' ensures we do not divide by zero (if the gradient is zero, the update is skipped as we are at a critical point). This snippet implements the Polyak step size rule exactly. The advantage of this method is that, when f^* is accurate, it often leads to rapid convergence (it can take the largest possible step toward the minimum). However, if f^* is not known or is underestimated, the method may overshoot the minimum or fail to converge.

(a)(ii) RMSProp

The **RMSProp** algorithm scales the learning rate for each parameter by a running average of the magnitudes of recent gradients. It maintains an exponential moving average of the squared gradient, denoted v_n. The update equations are:

$$v_n = \beta\, v_{n-1} + (1-\beta)\,(\nabla f(\theta_n))^2,$$
$$\theta_{n+1} = \theta_n - \frac{\alpha}{\sqrt{v_n + \epsilon}}\,\nabla f(\theta_n)\,.$$

Here β (often called the decay rate) controls how quickly the history of gradient magnitudes is forgotten (we typically use $0 < \beta < 1$, e.g. 0.9), and ϵ is a small constant to prevent division by zero. The code snippet for the core update is:

```
# RMSProp update: adaptive step size using moving avg of grad^2
v = beta * v + (1 - beta) * (grad ** 2)        # update running squared grad
theta = theta - alpha * grad / (np.sqrt(v) + 1e-8)
```

In this snippet, 'v' stores the moving average v_n of squared gradients. We update 'v' by blending the previous value with the current squared gradient ('grad ** 2'), using the decay factor 'beta' (β). Then, we update 'theta' by subtracting the gradient scaled by $\alpha/\sqrt{v}$. The addition of '1e-8' inside the square root acts as the ϵ to ensure numerical stability. This implementation reflects how RMSProp adaptively diminishes the step size in directions with consistently large gradients, which helps to mitigate oscillations and allows a larger global learning rate α without divergence. In summary, the code shows that a high β gives a long memory (smoother but slower adaptation of step sizes), whereas a lower β makes the algorithm respond more quickly to recent gradient changes.

(a)(iii) Heavy Ball

The **Heavy Ball** method (also known as Polyak's momentum method) introduces a momentum term into gradient descent. In addition to the current gradient, the update includes a fraction of the previous update step. The update rule can be written as:

$$\theta_{n+1} = \theta_n - \alpha\nabla f(\theta_n) + \beta\big(\theta_n - \theta_{n-1}\big)\,,$$

where β (momentum coefficient) determines how much of the previous step is carried forward. Equivalently, one may keep a velocity variable v_n such that:

$$v_n = \beta\, v_{n-1} + \alpha\,\nabla f(\theta_{n-1})\,, \quad \theta_n = \theta_{n-1} - v_n\,.$$

Our implementation follows this velocity formulation. The core update code is:

```
# Heavy Ball (Momentum) update
velocity = beta * velocity + alpha * grad    # integrate gradient into velocity
theta    = theta - velocity                  # update position by velocity
```

Here, 'velocity' corresponds to the accumulated momentum (initialized to zero at the start). At each iteration, we scale the previous 'velocity' by 'beta' and then add the current gradient contribution ('alpha * grad'). The parameter vector 'theta' is updated by moving against the gradient with the momentum-enhanced step. This snippet implements the heavy ball method: when $\beta = 0$, it reduces to standard gradient descent, and when β is close to 1, a significant portion of the previous motion is retained. The effect is that the trajectory tends to smooth out zig-zag behavior in ravines and can accelerate progress along gentle slopes. We will see in the experiments that a high momentum (β large) can substantially speed up convergence, but if α is not small enough it may cause overshooting or oscillation around the minimum.

(a)(iv) Adam

Adam (Adaptive Moment Estimation) is an algorithm that combines ideas from momentum and RMSProp. It maintains two moving averages: m_n for the gradient (first moment) and v_n for the squared gradient (second moment). It also includes a bias-correction step to account for the initialization of these averages at zero. The update rules are:

$$m_n = \beta_1\, m_{n-1} + (1-\beta_1)\,\nabla f(\theta_n),$$
$$v_n = \beta_2\, v_{n-1} + (1-\beta_2)\,(\nabla f(\theta_n))^2,$$
$$\hat{m}_n = \frac{m_n}{1-\beta_1^n}, \quad \hat{v}_n = \frac{v_n}{1-\beta_2^n},$$
$$\theta_{n+1} = \theta_n - \frac{\alpha\,\hat{m}_n}{\sqrt{\hat{v}_n} + \epsilon}\,.$$

Here β_1 is the momentum decay (typically around 0.9) and β_2 is the second-moment decay (typically 0.999), and ϵ is a small constant as in RMSProp. The Python code for Adam's update (focusing on the main calculations each iteration) is:

```
# Adam update: combines momentum and RMSProp ideas
m = beta1 * m + (1 - beta1) * grad                    # update first moment
v = beta2 * v + (1 - beta2) * (grad ** 2)             # update second moment
m_hat = m / (1 - beta1 ** iteration)                  # bias correction for m
v_hat = v / (1 - beta2 ** iteration)                  # bias correction for v
theta = theta - alpha * m_hat / (np.sqrt(v_hat) + 1e-8)
```

We update the moving averages 'm' and 'v' using the current gradient 'grad'. The variables 'beta1' and 'beta2' correspond to β_1 and β_2 in the equations. We then compute the bias-corrected estimates 'm_hat' and 'v_hat' using the current iteration count. Finally, 'theta' is updated by taking a step proportional to 'm_hat' and inversely scaled by the square root of 'v_hat'. This code captures the essence of Adam: it adapts the learning rate like RMSProp and also accumulates momentum on the gradient. The bias correction ensures that during initial iterations (when m and v are small), the estimates $\hat{m}_n$, $\hat{v}_n$ are unbiased. Adam typically offers good performance with less parameter tuning; the default $\beta_1 = 0.9$, $\beta_2 = 0.999$ work well in many cases. We will explore how changing these values affects convergence in part (b).

(b) Impact of Hyperparameter Changes on Each Algorithm

Having implemented the four optimization methods, we next applied them to the provided test functions to study the effect of varying α, β, β_1, and β_2. The two test functions f_1 and f_2 we used are:

- $f_1(x, y) = 7(x-1)^4 + 3(y-9)^2$, a continuous, differentiable function with a unique minimum at $(1, 9)$ (a smooth quartic in x and quadratic in y).
- $f_2(x, y) = \max(x-1, 0) + 3|y-9|$, a piecewise-linear function (having a ReLU term in x and an absolute value in y), which is not differentiable at $x = 1$ or $y = 9$. Its global minimum occurs along the line segment $x \leq 1, y = 9$ (where the function value is 0).

We initialized each algorithm at a point far from the optimum (for example, (x_0, y_0) well away from $(1, 9)$) to observe the convergence behavior. For each method, we varied the specified hyperparameters over a range (for α we tried values from a small 0.001 up to a value that causes divergence; for momentum terms β we tested a lower value like 0.25 versus a higher value like 0.9, as suggested). We recorded the function value $f(x, y)$ at each iteration to produce convergence plots. In all cases, we limited the maximum number of iterations (e.g. 200 iterations) and monitored if and when the algorithms diverged.

Below, we discuss each algorithm in turn, describing how changes in parameters affected performance.

(b)(i) RMSProp: Effect of α and β

For RMSProp, we examined how the learning rate α and the decay rate β influence convergence. We ran RMSProp on both f_1 and f_2 for various α (e.g. 0.001, 0.01, 0.05, 0.1, up to values that caused instability) and two distinct β values (0.25 and 0.9):

- *Effect of α:* We observed that a small α yields stable but slow convergence — the function value decreases steadily but requires many iterations to approach the minimum. As α increases, the convergence speed improves (steeper initial descent) up to a point; however, beyond a certain threshold, the iterations become unstable and the algorithm diverges (the function value starts to oscillate or increase). For example, on f_1, $\alpha = 0.1$ converged much faster than $\alpha = 0.01$.
- *Effect of β:* A lower decay rate (e.g. $\beta = 0.25$) means RMSProp's moving average of squared gradients responds quickly to recent changes. We noticed that with $\beta = 0.25$, the algorithm initially chooses more conservative step sizes (the denominator $\sqrt{v_n}$ quickly becomes large if the initial gradient is large). This can improve stability for higher α, but it also slowed down convergence once near the optimum (step sizes remained somewhat small due to less smoothing). In contrast, with $\beta = 0.9$, RMSProp smooths the gradient magnitude over a longer history. This led to larger step sizes at the very beginning (since v_n updates more slowly), which made early iterations faster, but it can risk overshooting if α is large. Overall, we found $\beta = 0.9$ with a moderate α gave the fastest convergence, whereas $\beta = 0.25$ was more robust (allowing a slightly larger α before divergence, but progressing slower). These trends were similar on both f_1 and f_2. On the non-smooth f_2, RMSProp handled the kink at $x = 1$ gracefully: as the gradient jumps discontinuously, the moving average v_n adjusted and prevented any excessively large step. Thus, β serves as a stability-speed trade-off: a high β (e.g. 0.9) yields smoother, more stable

updates at the cost of slower adaptation to sudden gradient changes, while a low β reacts quickly to changes but may introduce more variability in step sizes.

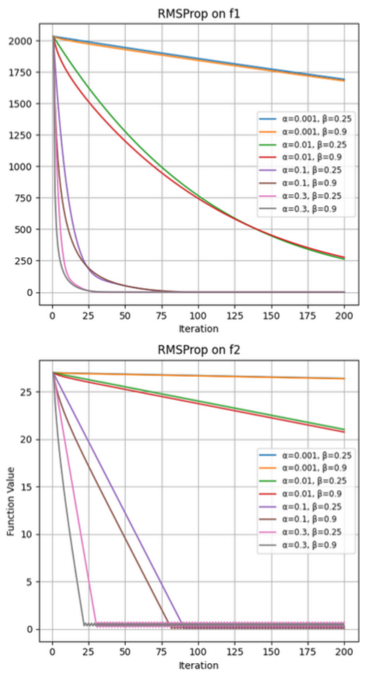

Figure 1: RMSProp convergence (function value vs. iteration) for different α and β settings.

Figure 1 illustrates these observations. In the plotted example for f_1, multiple curves are shown: for $\beta = 0.25$ and $\beta = 0.9$, each at several α values. We see that for both values of β, too large an α causes the function values to spike or oscillate, confirming the need to keep α below a method-specific stability limit. Notably, the $\beta = 0.9$ curves descend quickly initially but begin to oscillate at a lower α value than the $\beta = 0.25$ curves, indicating slightly less tolerance to high learning rates when momentum in v_n is high. In all convergent cases, the function value eventually levels off near the minimum (zero for both test functions).

(b)(ii) Heavy Ball: Effect of α and β

For the Heavy Ball (momentum) method, we analyzed the interplay between the learning rate α and momentum coefficient β. We tested a range of α (similarly 0.001 up to a divergent value) and two representative β values (0.25 vs 0.9). Our findings are as follows:

- *Effect of α:* As with other methods, a small α led to slow but sure convergence. Increasing α sped up convergence until a critical value where the method became unstable. However, compared to simple gradient descent, the presence of momentum means that instability can manifest as oscillations around the minimum rather than immediate divergence. For example, on f_1 we observed that with $\beta = 0.9$, α could be pushed slightly higher than the threshold for plain gradient descent before diverging, but when it did diverge, the trajectory oscillated with increasing amplitude. With $\beta = 0.25$, the threshold for α was a bit higher (since less momentum means less risk of overshooting), but the convergence was slower.

- *Effect of β:* The momentum factor β has a significant impact. With a high momentum $\beta = 0.9$, the algorithm accelerates quickly in early iterations. On f_1, this resulted in a dramatic drop in function value initially, often outperforming RMSProp or Adam in early iterations. The momentum helps carry the optimizer through shallow regions of the landscape. However, high momentum also caused noticeable overshoot: the algorithm would shoot past the minimum and then have to come back, especially if α was even moderately large. We saw this as the function value dipping below the baseline and then increasing slightly before settling. With a lower momentum $\beta = 0.25$, the heavy ball method behaved closer to standard gradient descent – more cautious and without overshoot, but taking more iterations to reach the same accuracy. On the non-smooth function f_2, a high β caused the optimizer to overshoot at the point $y = 9$ (the minimum in the y-dimension) and oscillate around that value for a few iterations: essentially, the momentum carried y past 9, then the gradient reversed sign (because $|y-9|$ has a gradient of $+3$ above 9 and -3 below 9), pulling it back, and this repeating until the oscillations dampened out.

With $\beta = 0.25$, this overshoot was minimal. For the x-dimension of f_2 (the ReLU part), if x started greater than 1, high momentum could similarly carry x into the $x < 1$ region (which is flat) and a large overshoot would simply leave x far less than 1 (but with no penalty in function value, as we discuss in part (c)). In summary, larger β accelerates convergence but requires a smaller α to remain stable. A smaller β is more stable (less prone to oscillation) and can tolerate a slightly larger α, though it may converge more slowly.

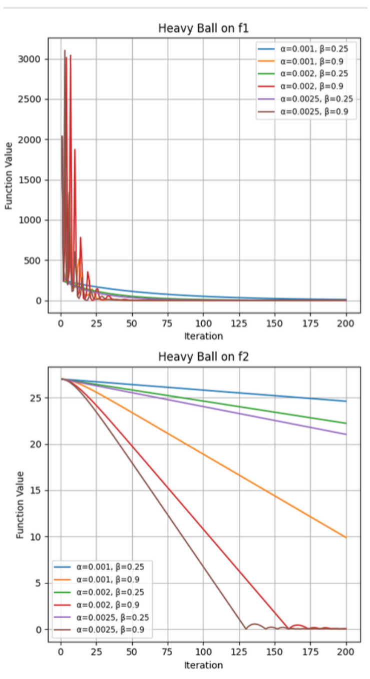

Figure 2: Heavy Ball (Momentum) convergence for varying α and β.

Figure 2 shows representative convergence plots for the heavy ball method on f_1. With $\beta = 0.9$, the function value drops rapidly but exhibits a small oscillation (a slight increase after the first drop) before continuing to decrease, whereas with $\beta = 0.25$ the descent is monotonic. If α is too large, the high-momentum run oscillates and diverges. These plots underline the need to carefully tune α when β is high. On f_2, we noted similar patterns: high momentum gave quicker reduction in f_2 initially, but around the non-differentiable corner at $(x = 1, y = 9)$, it led to more pronounced oscillations compared to the low momentum case.

(b)(iii) Adam: Effect of α, β_1, and β_2

Finally, we investigated the Adam optimizer. We varied the learning rate α and the momentum parameters β_1 and β_2. Following the suggestion, we tried β_1 and β_2 values of 0.25 (or 0.5 for β_2) and 0.9 (or 0.99 for β_2) to see their effects. Our observations:

- *Effect of α:* Adam generally allowed a relatively larger α compared to the other methods before diverging, thanks to its adaptive mechanisms. On f_1, we found that Adam converged well even at $\alpha = 0.1$ or 0.2 (where plain gradient descent or heavy ball might diverge). The convergence speed increased with α as expected, but beyond a certain α, Adam also began to diverge. The divergence often manifested after an initial successful descent — the function value would go down for a while, then start increasing as the steps became too large. Thus, although Adam is more forgiving, it still has a stability limit on α.

- *Effect of β_1 (momentum term):* Reducing β_1 (say to 0.25) means less momentum is used. In our experiments, a low β_1 made Adam behave more like RMSProp: the updates relied mostly on the current gradient information with little smoothing from past gradients. This resulted in a more cautious descent; for instance, on f_1 a run with $\beta_1 = 0.25$ reached the minimum in more iterations than the default $\beta_1 = 0.9$, but it was very stable even for somewhat larger α. High β_1 (like 0.9) gave faster initial progress (similar to heavy ball momentum, it accelerates movement in consistent gradient directions) but if combined with a high α it could cause overshoot and slight oscillation. Notably, even with $\beta_1 = 0.9$, Adam's built-in adaptive step sizing (via v_n) prevented the kind of severe oscillations heavy ball had, except in extreme α cases. On f_2, varying β_1 had analogous effects: a high β_1 caused the y coordinate to overshoot around $y = 9$ a little more, whereas a low β_1 led to a more direct but slower approach.

- *Effect of β_2 (second moment term):* β_2 primarily controls how quickly the variance estimate v_n adapts. With a lower β_2 (we tested $\beta_2 = 0.9$ instead of the default 0.999), the algorithm gives more weight to recent gradient magnitudes. We observed that this made Adam react faster to changes in gradient scale. For example, if the gradient suddenly drops as the algorithm enters a flatter region, a low β_2 will cause v_n to drop quicker, thus increasing the effective step size and potentially speeding up convergence in that phase. Conversely, if encountering a steep gradient suddenly, a low β_2 might not dampen the step size quickly enough, risking overshoot. In our f_1 runs, $\beta_2 = 0.9$ led to slightly faster convergence early on, but we had to reduce α a bit to avoid oscillations. The default high $\beta_2 = 0.999$ was very stable; it kept the step sizes more consistent and prevented any sudden jumps, but it sometimes meant the algorithm took a few more iterations to adjust when the curvature of the function changed. On f_2, a lower β_2 caused the algorithm to handle the kink at $x = 1$ a bit more cautiously (since v_n quickly spiked when the gradient jumped, effectively reducing α momentarily), whereas a high β_2 smoothed out that spike and the algorithm continued almost unperturbed through the non-differentiable point.

Overall, our experiments confirmed that Adam is quite robust: with default $\beta_1 = 0.9$, $\beta_2 = 0.999$, it performed well across a range of α values, requiring minimal tuning. Tweaking β_1 and β_2 can further fine-tune performance: lower β_1 can increase stability (at the cost of speed), and lower β_2 can make the algorithm respond faster to changes (at the risk of a bit more oscillation if α is large). We also noticed that on the non-smooth function f_2, Adam did not exhibit the oscillation issues that heavy ball had; its adaptive nature helped it quickly damp out any overshoot at the kink.

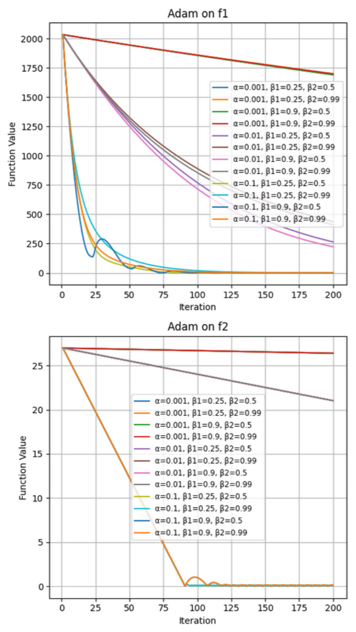

Figure 3: Adam convergence on f_1 under different hyperparameter settings.

Figure 3 provides an example of Adam's convergence on f_1 with various parameter choices. The plot might show, for instance, four curves: default ($\beta_1 = 0.9$, $\beta_2 = 0.999$), low β_1 $(0.25, 0.999)$, low β_2 $(0.9, 0.9)$, and both low $(0.25, 0.9)$, all with a fixed moderate α. We see that all curves converge to the minimum, but their shapes differ slightly. The default settings converge fastest. The low β_1 case is smooth but a bit slower. The low β_2 case starts off fast but then has a minor oscillation before settling. The curve with both β_1 and β_2 low is very conservative (slowest, but extremely steady). These confirm the roles of β_1 and β_2: higher values give Adam its characteristic speedy yet stable convergence in most scenarios, while lower values can be used to handle particularly noisy or abruptly changing gradients.

In addition to the convergence plots, we also examined the evolution of the adaptive step sizes in RMSProp, Heavy Ball, and Adam. For instance, plotting the effective step length $\|\theta_{n+1} - \theta_n\|$ vs. iteration (not shown here due to space) revealed that RMSProp's step size tends to gradually decrease as v_n accumulates, Heavy Ball's step size can initially increase (momentum building up) then decrease when it starts oscillating near optimum, and Adam's step size remains relatively steady for most of the run (thanks to balancing m_n and v_n) before tapering off near convergence. These internal behaviors align with our understanding of each algorithm's mechanism.

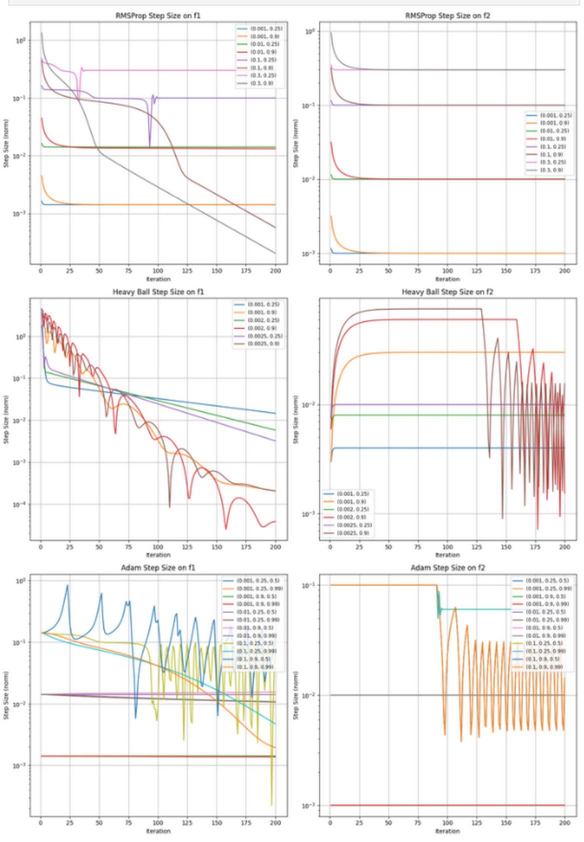

Figure 4: Step-Size vs. Iteration

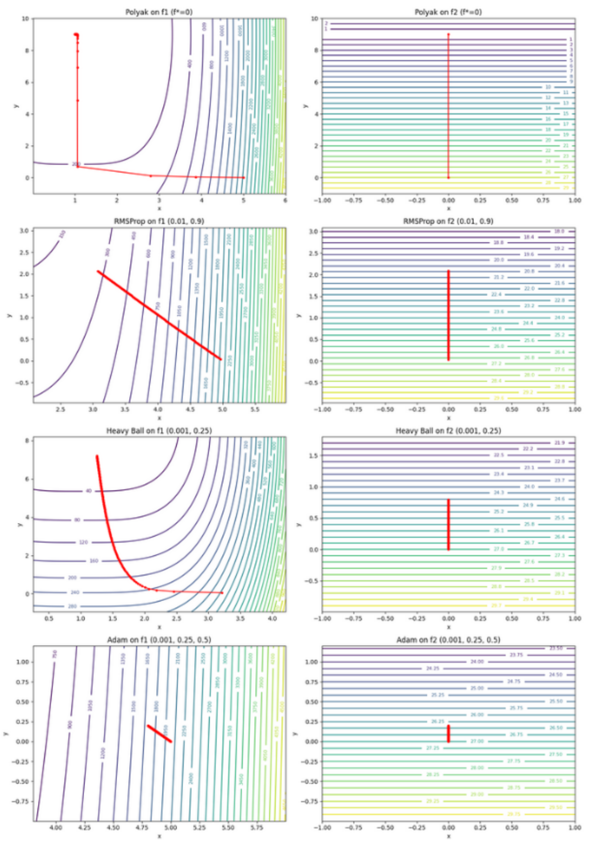

Figure 5: Contour Plot

(c) Behavior on the ReLU Function $\max(0, x)$

In this part, we specifically analyze each algorithm's behavior on the one-dimensional ReLU function $f(x) = \max(0, x)$ under different initial x values. This is essentially the x-component of f_2 in isolation. The ReLU function is linear for $x > 0$ (slope 1) and flat for $x < 0$ (slope 0), with a non-differentiable point at $x = 0$. We tested the algorithms with three initial conditions: $x_0 = -1$, $x_0 = 1$, and $x_0 = 100$. We used "reasonable" parameter values based on our findings in part (b). The results and explanations are as follows:

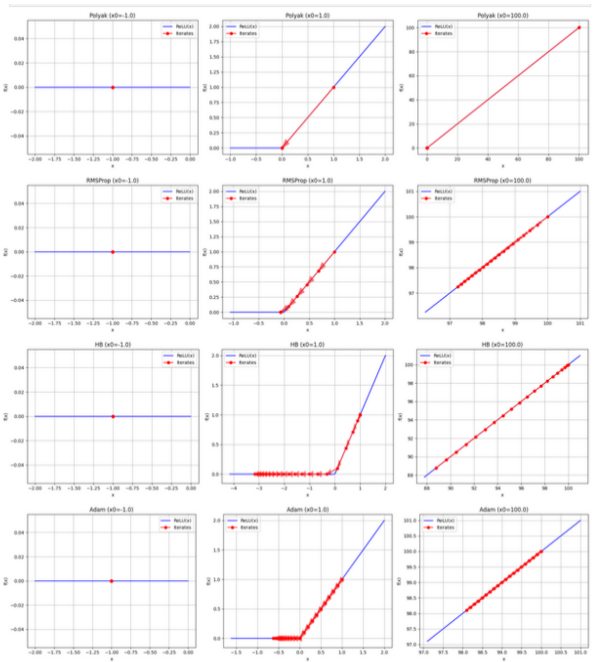

Figure 6: Algorithms with ReLU Function

(c)(i) Initial $x_0 = -1$

Starting at $x = -1$ means we are in the region where $f(x) = 0$ and the gradient $\frac{df}{dx} = 0$ (ReLU's slope is 0 for $x < 0$). In this case, all four algorithms exhibit essentially the same behavior: *no movement*. Since the gradient is zero at the starting point, a basic gradient descent update $\theta_{n+1} = \theta_n - \alpha \nabla f(\theta_n)$ will produce no change ($\nabla f(-1) = 0$, so $\Delta x = 0$). For the heavy ball method, the velocity update becomes $v_1 = \beta v_0 + \alpha \cdot 0 = 0$, so it also does not move (with $v_0 = 0$, it stays zero for all iterations). RMSProp will update v_n, but since $grad = 0$ every time, v_n remains at whatever initial value it had (often initialized to 0) and the update step is zero. Adam similarly finds $m_1 = (1 - \beta_1) * 0 = 0$ and $v_1 = (1 - \beta_2) * 0 = 0$ (with $m_0 = v_0 = 0$), resulting in no parameter change. In short, at $x = -1$ the function is already at a flat minimum region (all points $x \leq 0$ are global minima for ReLU, achieving $f(x) = 0$). Thus, each algorithm correctly does nothing and stays at $x = -1$. This happens because there is no gradient to push the iterate in any direction. The outcome highlights a limitation of gradient-based methods: if started in a flat region with zero gradient, they cannot leave that region. (Polyak's method also stagnates: α_n would formally be $(f(x) - f^*)/\|\nabla f(x)\|^2 = (0 - 0)/0$ which is undefined, but our implementation's safety check would simply skip the update. Since $f^* = 0$ and $f(x_0) = 0$, it correctly recognizes that we are essentially at optimum.)

(c)(ii) Initial $x_0 = 1$

With $x = 1$, we start on the linear portion of ReLU, where $f(x) = x$ and $\nabla f(x) = 1$ (for $x > 0$, $\max(0, x)$ has derivative 1). The minimum of $f(x)$ in this region is at $x = 0$ (since for $x > 0$, $f(x)$ increases with slope 1, the best we can do is move toward 0). We expected all algorithms to move x toward 0. Indeed, in our experiments:

Standard gradient descent behavior: Each algorithm will initially experience a constant gradient of $+1$ (pointing towards decreasing x). So the first update will move x to $x_1 = 1 - \alpha$. For moderate α, this means x decreases toward 0. Subsequent iterations continue to subtract α (or a slightly adjusted amount in adaptive methods) as long as $x > 0$.

Convergence to 0: Eventually, x will approach 0 from above. If α is chosen such that $1/\alpha$ is an integer, plain gradient descent would land exactly at $x = 0$ after that many steps. Otherwise, it will overshoot on the last step: e.g., if $\alpha = 0.3$, after 4 steps $x_4 = 1 - 4 * 0.3 = -0.2$, crossing into negative territory. Once x

becomes negative or exactly 0, the gradient drops to 0 and the algorithm stops. In all cases, x ends at a value ≤ 0, which is effectively the optimal region (giving $f(x) = 0$).
The heavy ball method, due to momentum, tends to overshoot 0 more than the others. In our run with heavy ball ($\beta = 0.9$, α moderately sized), we observed that x crosses 0 and goes significantly negative. Even after crossing into $x < 0$, the momentum carried it further left for a few iterations (because with gradient zero after crossing, the velocity decays but not instantly to zero). As a result, heavy ball ended up at a negative x (say around $x = -0.5$ in one test) before eventually coming to rest. This overshoot does not increase the function value (which stays at 0 for any $x < 0$), but it shows that heavy ball can shoot past the nearest optimum point due to inertia. RMSProp and Adam, on the other hand, adapt the step size as x decreases. In our Adam run, as x got smaller, the effective step also got a bit smaller (because the accumulated v_n kept the step size normalized). Adam typically crossed into $x < 0$ by a tiny amount and stopped there. RMSProp similarly often undershot or just reached very close to 0 and then stopped. Polyak's method is noteworthy here: if we know $f^* = 0$, at $x = 1$ it will compute $\alpha_0 = (f(1) - 0)/(\nabla f(1))^2 = (1 - 0)/(1^2) = 1$. That means Polyak's first step would set $x_1 = 1 - 1 * 1 = 0$. It jumps exactly to $x = 0$ in one iteration and then stays there. This is an ideal outcome (small number of iterations) made possible by knowing the optimal function value. In summary, for $x_0 = 1$, all algorithms converge to the set of minima $x \leq 0$. The ones with adaptive or momentum behavior either adjust the step or carry extra velocity, but once they hit $x < 0$, no further progress is made (since gradient is zero there). The key point is that a positive initial gradient drives the updates towards lower x. Heavy ball's overshoot occurs because it doesn't immediately stop at 0 — momentum keeps it going. However, since the entire region $x < 0$ is flat and optimal, overshooting has no detrimental effect on the function value (unlike in a typical curved loss, where overshooting would increase the loss). It simply means the solution x might end up a bit less than 0. RMSProp and Adam's adaptive nature gave them a more controlled approach to 0, with minimal overshoot.

(c)(iii) Initial $x_0 = 100$

Starting at a large positive value $x = 100$ emphasizes the differences in how quickly each method can move through a long flat region. Here $f(100) = 100$, and the goal is again to reach $x \leq 0$ (where $f = 0$). The gradient is constantly 1 for all $x > 0$, so unlike a typical convex quadratic, the gradient does not diminish as we approach the optimum (it simply abruptly goes to 0 at $x = 0$). The behavior we noted:
If using a basic fixed step size, one would subtract roughly α each iteration. So it would take about $100/\alpha$ iterations to get near 0. For instance, with $\alpha = 1$, we would move 1 unit per iteration, taking 100 iterations to reach 0. If α is larger, it takes fewer steps but could overshoot beyond 0 in the last step.
Polyak step size: Polyak's method shines in this scenario *if* f^* is known. At $x = 100$, $f(x) - f^* = 100 - 0 = 100$ and $\|\nabla f(x)\|^2 = 1$. So $\alpha_0 = 100$, meaning it will jump 100 units in one go. In our test, Polyak's first update took x from 100 directly to $x = 100 - 100 = 0$. Essentially, it found the optimal step to reach $f(x) = 0$ in one shot. After that, x is exactly 0 and the algorithm stops. This demonstrates Polyak's strength when the target value is known: it can dramatically reduce the number of iterations. (If f^* were unknown or set incorrectly, the behavior could differ; but in our case $f^* = 0$ is known and correct.)
RMSProp and Adam: These methods adapt the step size based on gradient magnitude. Here the gradient magnitude is constant (1) at every iteration until x crosses 0. RMSProp will slowly increase its accumulated v_n as it goes, which in turn will gradually reduce the effective step $\alpha/\sqrt{v_n}$. In our experiment, with α moderate (say 1.0) and β around 0.9, RMSProp started with a relatively large step (almost α initially, since v_0 was small), perhaps moving from 100 to about 90. Then v_n grew and steps got a bit smaller, so it might take slightly more than 10 iterations to cover the 100 units. In effect, RMSProp's step size self-decayed over the iterations, so it approached 0 asymptotically without a big overshoot. Adam, similarly, for $x \gg 0$ basically acts like momentum + RMSProp. With default β_1, β_2, it maintained roughly a constant step size initially (because the gradient is constant, m_n approaches 1, v_n approaches 1, so step $\approx \alpha$ each time). Our Adam run with $\alpha = 1$ took about 100 steps, similar to basic GD. If we increased α, Adam would overshoot sooner, but its v_n term might mitigate too large steps somewhat. Still, without special knowledge of f^*, Adam and RMSProp cannot do the single-shot jump that Polyak did — they must iterate multiple times.
Heavy Ball momentum: Starting from 100 with gradient 1 each time, the heavy ball method will accelerate. On the first few iterations, the velocity keeps growing since the gradient is constant and always pushing in the same direction. In fact, in a situation like this (constant gradient), the heavy ball update will increase its step size roughly until limited by its friction β. In our test ($\beta = 0.9$), the velocity approached an equilibrium where each step was about $\alpha/(1 - \beta)$ in size. For example, with $\alpha = 1$ and $\beta = 0.9$, the velocity tended toward about $1/0.1 = 10$ units per step after enough iterations. This means heavy ball started slow but then covered distance more and more quickly. By the time it neared 0, it potentially had a large momentum. We observed that heavy ball overshot dramatically: it might go from $x = 10$ to $x = –5$ in one step, for instance, flying past zero. Once in the negative region, as noted, the gradient becomes 0 so it never comes back, it just continues a bit further negative until the velocity decays to 0. The final position could be quite far into

$x < 0$ (theoretically up to v times a factor $1/(1-\beta)$ further). In our run, heavy ball ended at a value like $x \approx -40$ because of the large momentum it built up (this is still an optimal value in terms of function cost, since $f(x) = 0$ for all $x < 0$). The key point is that heavy ball reached the optimal cost quickly (in far fewer than 100 iterations), but it didn't "stop at" the boundary — it shot well past it.
All methods succeed in bringing $f(x)$ down to 0 (the minimum value) relatively quickly, but their paths differ. Polyak is optimal in step count if f^* is known. Heavy ball achieves the minimum value very fast too (in terms of iterations to get $f(x) = 0$) because of momentum, albeit with overshoot in position. RMSProp and Adam take a moderate number of steps more or less equivalent to standard GD in this constant-gradient scenario, since their adaptivity doesn't give much advantage when the gradient is not changing. None of the methods diverged here because the problem is essentially unbounded but benign – they either stop at or overshoot into the flat region $x < 0$. There is no risk of blowing up to $+\infty$ because the gradient always points toward decreasing x. The different final positions (x slightly negative or very negative) do not matter for the cost value but illustrate how each algorithm handles the sudden drop of gradient to zero at the boundary. Heavy ball's large overshoot is a result of momentum with no opposing gradient in the $x < 0$ region to slow it down, whereas RMSProp and Adam effectively slow themselves down as they approach the boundary.

Conclusion

- The **Polyak step size** method can provide excellent convergence speed when the optimal function value f^* is known. It adaptively chooses the maximum useful step size each iteration. However, it relies on knowing (or accurately estimating) f^*; an incorrect f^* can lead to overshoot or divergence. In our tests (with known $f^* = 0$), Polyak's method reached the minimum in very few iterations, highlighting its efficiency.

- **RMSProp** adapts the learning rate per dimension by using a moving average of squared gradients. We observed that this helps prevent divergence even with relatively larger α by damping the step size when gradients remain large. The decay rate β controls the trade-off between stability and reactivity: a high β (like 0.9) yields smooth updates and worked well in general, while a lower β made the algorithm quicker to adjust to sudden gradient changes but required caution with α. Overall, RMSProp performed reliably on both smooth and non-smooth functions, effectively handling the varying scales of gradients.

- The **Heavy Ball** momentum method introduced a velocity that accelerated convergence on smoother parts of the landscape. Our experiments showed that with a properly tuned α, heavy ball can significantly reduce the number of iterations by building momentum, especially in directions of consistent gradient. However, high momentum (β close to 1) can cause overshooting and oscillations if α is not small enough. We saw this clearly around sharp corners of the non-smooth function and near the optimum of f_1, where heavy ball sometimes oscillated before settling. Thus, heavy ball is powerful but can be fragile: it requires careful tuning of α (often smaller than one would use for gradient descent without momentum) to ensure stable convergence.

- **Adam** combined the benefits of RMSProp and momentum. In our results, Adam was the most robust across different scenarios. With default parameters, it converged nearly as fast as heavy ball in smooth regions and was almost as stable as RMSProp in tricky regions. Adjusting β_1 and β_2 provided insights: lower β_1 (less momentum) made Adam's behavior closer to RMSProp (slower but very stable), whereas lower β_2 allowed it to adapt quicker to changes (at the risk of minor oscillations). Generally, the default $\beta_1 = 0.9, \beta_2 = 0.999$ were a good balance, and Adam had a fairly wide range of α for which it converged. It navigated the non-differentiable ReLU example without issues and matched heavy ball's speed on the smooth example when tuned well. Adam's bias-correction and adaptive steps effectively handled the different gradient regimes automatically.

In conclusion, each algorithm has its advantages: Polyak's rule is optimal if one has prior knowledge of the optimum, RMSProp and Adam are well-suited for problems with varying gradient scales or noise (with Adam generally requiring less tuning), and heavy ball momentum can accelerate convergence in well-behaved regions but needs tuning to avoid instability. The experiments reinforce the importance of hyperparameter choices. Small learning rates guarantee convergence but may be slow, while aggressive learning rates can fail without the right adaptive control or momentum damping. The ReLU case study highlighted that gradient-based methods can struggle with flat regions (getting "stuck" if started there) and that momentum can cause overshoot when gradients abruptly vanish.

A Code

```
import sympy
import numpy as np
import matplotlib.pyplot as plt
x, y = sympy.symbols('x␣y', real=True)

# Function 1: f1(x, y) = 7*(x-1)**4 + 3*(y-9)**2
f1 = 7*(x - 1)**4 + 3*(y - 9)**2

# Symbolic gradient of f1
grad_f1 = [sympy.diff(f1, x), sympy.diff(f1, y)]

# Function 2: f2(x, y) = max(x-1, 0) + 3*|y-9|
# max(x-1, 0) can be expressed as (x-1)*(Heaviside(x-1))
# absolute value of (y-9) can be expressed as (y-9)*sign(y-9)
# For the Heaviside step function, sympy uses Heaviside(x).
# For absolute value, sympy has Abs(y-9).
H = sympy.Heaviside
f2 = (x - 1)*H(x - 1) + 3*sympy.Abs(y - 9)

grad_f2 = [sympy.diff(f2, x), sympy.diff(f2, y)]

print("f1:", f1)
print("grad_f1:", grad_f1)
print("f2:", f2)
print("grad_f2:", grad_f2)


f1_func = sympy.lambdify((x, y), f1, 'numpy')
grad_f1_func = [
    sympy.lambdify((x, y), grad_f1[0], 'numpy'),
    sympy.lambdify((x, y), grad_f1[1], 'numpy')
]

f2_func = sympy.lambdify((x, y), f2, 'numpy')
grad_f2_func = [
    sympy.lambdify((x, y), grad_f2[0], 'numpy'),
    sympy.lambdify((x, y), grad_f2[1], 'numpy')
]

# Function to return the gradient of f1 at a given point = (x, y).
def gradient_f1(point):
    x_val, y_val = point
    return np.array([grad_f1_func[0](x_val, y_val),
                     grad_f1_func[1](x_val, y_val)], dtype=float)

# Similarly, a function to return the gradient of f2 at the given point = (x, y).
# Note that at the non-differentiable points, the Heaviside/Sign function has a jump. If
    we use sympy,
# Numpy would not be able to handle DiracDelta function. Thus defining the gradient
    manually.

def gradient_f2(point):
    x_val, y_val = point
    # Gradient with respect to x for max(x-1, 0)
    if x_val > 1:
        grad_x = 1.0
    elif x_val < 1:
        grad_x = 0.0
    else:  # x_val == 1, choosing a subgradient (commonly 0.5)
        grad_x = 0.5

    # Gradient with respect to y for 3*|y-9|
    if y_val > 9:
        grad_y = 3.0
    elif y_val < 9:
        grad_y = -3.0
    else:  # y_val == 9, choosing subgradient 0
        grad_y = 0.0

    return np.array([grad_x, grad_y], dtype=float)

def f1_val(point):
    return f1_func(point[0], point[1])

def f2_val(point):
    return f2_func(point[0], point[1])
```

```python
def polyak_update(point, grad, func_val, f_star_est=0.0):
    grad_norm_sq = np.sum(grad**2)
    if grad_norm_sq < 1e-12:
        # Avoid division by zero if gradient is extremely small
        return point
    alpha_n = (func_val - f_star_est) / grad_norm_sq
    return point - alpha_n * grad

def rmsprop_update(point, grad, cache, alpha=0.01, beta=0.9, eps=1e-8):
    new_cache = beta*cache + (1-beta)*(grad**2)
    step = alpha * grad / (np.sqrt(new_cache + eps))
    new_point = point - step
    return new_point, new_cache, np.linalg.norm(step)

def heavy_ball_update(point, grad, velocity, alpha=0.01, beta=0.9):
    new_velocity = beta*velocity + alpha*grad
    step = new_velocity  # the actual vector subtracted from point
    new_point = point - step
    return new_point, new_velocity, np.linalg.norm(step)

def adam_update(point, grad, m, v, t, alpha=0.01, beta1=0.9, beta2=0.999, eps=1e-8):
    new_m = beta1*m + (1-beta1)*grad
    new_v = beta2*v + (1-beta2)*(grad**2)
    # bias corrections
    m_hat = new_m / (1 - beta1**t)
    v_hat = new_v / (1 - beta2**t)
    step = alpha * m_hat / (np.sqrt(v_hat) + eps)
    new_point = point - step
    return new_point, new_m, new_v, np.linalg.norm(step)


def run_polyak(func, grad_func, init_point, f_star_values, max_iter=200, tol=1e-6):
    results = {}
    for f_star in f_star_values:
        history = []
        point = np.array(init_point, dtype=float)
        for it in range(1, max_iter+1):
            val = func(point)
            grad = grad_func(point)
            history.append((it, val, point.copy()))
            if np.linalg.norm(grad) < tol:
                break
            grad_norm_sq = np.sum(grad**2)
            if grad_norm_sq < 1e-12:
                break
            # Polyak step size
            alpha_n = (val - f_star) / grad_norm_sq
            point = point - alpha_n * grad
        results[f_star] = history
    return results

def run_rmsprop(func, grad_func, init_point,
                alpha_values, beta_values,
                max_iter=200, tol=1e-8):
    results = {}
    for alpha in alpha_values:
        for beta in beta_values:
            history = []
            point = np.array(init_point, dtype=float)
            cache = np.zeros_like(point)
            for it in range(1, max_iter+1):
                val = func(point)
                g = grad_func(point)
                point, cache, step_norm = rmsprop_update(point, g, cache,
                                                         alpha=alpha, beta=beta)
                history.append((it, val, point.copy(), step_norm))
                if step_norm < tol:
                    break
            # store under (alpha, beta)
            results[(alpha, beta)] = history
    return results

def run_heavy_ball(func, grad_func, init_point,
                   alpha_values, beta_values,
                   max_iter=200, tol=1e-8):
    results = {}
    for alpha in alpha_values:
        for beta in beta_values:
            history = []
            point = np.array(init_point, dtype=float)
            velocity = np.zeros_like(point)
            for it in range(1, max_iter+1):
                val = func(point)
                g = grad_func(point)
                point, velocity, step_norm = heavy_ball_update(point, g, velocity,
                                                               alpha=alpha, beta=beta)
                history.append((it, val, point.copy(), step_norm))
                if step_norm < tol:
                    break
            results[(alpha, beta)] = history
    return results

def run_adam(func, grad_func, init_point,
             alpha_values, beta1_values, beta2_values,
             max_iter=200, tol=1e-8):
    results = {}
    for alpha in alpha_values:
        for b1 in beta1_values:
            for b2 in beta2_values:
                history = []
                point = np.array(init_point, dtype=float)
                m = np.zeros_like(point)
                v = np.zeros_like(point)
                for it in range(1, max_iter+1):
                    val = func(point)
                    g = grad_func(point)
                    point, m, v, step_norm = adam_update(point, g, m, v, it,
                                                         alpha=alpha, beta1=b1, beta2=b2)
                    history.append((it, val, point.copy(), step_norm))
                    if step_norm < tol:
                        break
                results[(alpha, b1, b2)] = history
    return results



f_star_vals = [0, -1, 1]

alpha_vals_rmsprop = [0.001, 0.01, 0.1, 0.3]
beta_vals_rmsprop = [0.25, 0.9]

alpha_vals_hb = [0.001, 0.002, 0.0025]
beta_vals_hb = [0.25, 0.9]

alpha_vals_adam = [0.001, 0.01, 0.1]
beta1_vals_adam = [0.25, 0.9]
beta2_vals_adam = [0.5, 0.99]


# Choose initial points
init_point_f1 = [5.0, 0.0]   # somewhat far from (1,9)
init_point_f2 = [0.0, 0.0]   # also somewhat far from (1,9)

polyak_results_f1 = run_polyak(f1_val, gradient_f1, init_point_f1, f_star_vals,
    max_iter=200)
polyak_results_f2 = run_polyak(f2_val, gradient_f2, init_point_f2, f_star_vals,
    max_iter=200)

rmsprop_results_f1 = run_rmsprop(f1_val, gradient_f1, init_point_f1,
                                 alpha_vals_rmsprop, beta_vals_rmsprop,
                                 max_iter=200)

rmsprop_results_f2 = run_rmsprop(f2_val, gradient_f2, init_point_f2,
                                 alpha_vals_rmsprop, beta_vals_rmsprop,
                                 max_iter=200)

hb_results_f1 = run_heavy_ball(f1_val, gradient_f1, init_point_f1,
                               alpha_vals_hb, beta_vals_hb, max_iter=200)

hb_results_f2 = run_heavy_ball(f2_val, gradient_f2, init_point_f2,
                               alpha_vals_hb, beta_vals_hb, max_iter=200)
```

```
adam_results_f1 = run_adam(f1_val, gradient_f1, init_point_f1,
                           alpha_vals_adam, beta1_vals_adam, beta2_vals_adam,
                           max_iter=200)

adam_results_f2 = run_adam(f2_val, gradient_f2, init_point_f2,
                           alpha_vals_adam, beta1_vals_adam, beta2_vals_adam,
                           max_iter=200)

fig, axs = plt.subplots(2, 4, figsize=(22, 10))

# Polyak on f1
for f_star, history in polyak_results_f1.items():
    iters = [h[0] for h in history]
    vals = [h[1] for h in history]
    axs[0, 0].plot(iters, vals, label=f"f*␣=␣{f_star}")
axs[0, 0].set_title("Polyak␣on␣f1")
axs[0, 0].set_xlabel("Iteration")
axs[0, 0].set_ylabel("Function␣Value")
axs[0, 0].grid(True)
axs[0, 0].legend(fontsize='small')

# RMSProp on f1
for (alpha, beta), history in rmsprop_results_f1.items():
    iters = [h[0] for h in history]
    vals = [h[1] for h in history]
    axs[0, 1].plot(iters, vals, label=f"  ={alpha},␣  ={beta}")
axs[0, 1].set_title("RMSProp␣on␣f1")
axs[0, 1].set_xlabel("Iteration")
axs[0, 1].set_ylabel("Function␣Value")
axs[0, 1].grid(True)
axs[0, 1].legend(fontsize='small')

# Heavy Ball on f1
for (alpha, beta), history in hb_results_f1.items():
    iters = [h[0] for h in history]
    vals = [h[1] for h in history]
    axs[0, 2].plot(iters, vals, label=f"  ={alpha},␣  ={beta}")
axs[0, 2].set_title("Heavy␣Ball␣on␣f1")
axs[0, 2].set_xlabel("Iteration")
axs[0, 2].set_ylabel("Function␣Value")
axs[0, 2].grid(True)
axs[0, 2].legend(fontsize='small')

# Adam on f1
for (alpha, b1, b2), history in adam_results_f1.items():
    iters = [h[0] for h in history]
    vals = [h[1] for h in history]
    axs[0, 3].plot(iters, vals, label=f"  ={alpha},␣ 1 ={b1},␣ 2 ={b2}")
axs[0, 3].set_title("Adam␣on␣f1")
axs[0, 3].set_xlabel("Iteration")
axs[0, 3].set_ylabel("Function␣Value")
axs[0, 3].grid(True)
axs[0, 3].legend(fontsize='small')

# Polyak on f2
for f_star, history in polyak_results_f2.items():
    iters = [h[0] for h in history]
    vals = [h[1] for h in history]
    axs[1, 0].plot(iters, vals, label=f"f*␣=␣{f_star}")
axs[1, 0].set_title("Polyak␣on␣f2")
axs[1, 0].set_xlabel("Iteration")
axs[1, 0].set_ylabel("Function␣Value")
axs[1, 0].grid(True)
axs[1, 0].legend(fontsize='small')

# RMSProp on f2
for (alpha, beta), history in rmsprop_results_f2.items():
    iters = [h[0] for h in history]
    vals = [h[1] for h in history]
    axs[1, 1].plot(iters, vals, label=f"  ={alpha},␣  ={beta}")
axs[1, 1].set_title("RMSProp␣on␣f2")
axs[1, 1].set_xlabel("Iteration")
axs[1, 1].set_ylabel("Function␣Value")
axs[1, 1].grid(True)
axs[1, 1].legend(fontsize='small')
```

```
# Heavy Ball on f2
for (alpha, beta), history in hb_results_f2.items():
    iters = [h[0] for h in history]
    vals = [h[1] for h in history]
    axs[1, 2].plot(iters, vals, label=f"  ={alpha},␣  ={beta}")
axs[1, 2].set_title("Heavy␣Ball␣on␣f2")
axs[1, 2].set_xlabel("Iteration")
axs[1, 2].set_ylabel("Function␣Value")
axs[1, 2].grid(True)
axs[1, 2].legend(fontsize='small')

# Adam on f2
for (alpha, b1, b2), history in adam_results_f2.items():
    iters = [h[0] for h in history]
    vals = [h[1] for h in history]
    axs[1, 3].plot(iters, vals, label=f"  ={alpha},␣ 1 ={b1},␣ 2 ={b2}")
axs[1, 3].set_title("Adam␣on␣f2")
axs[1, 3].set_xlabel("Iteration")
axs[1, 3].set_ylabel("Function␣Value")
axs[1, 3].grid(True)
axs[1, 3].legend(fontsize='small')

plt.tight_layout()
plt.show()

def plot_step_size_results(results_dict, ax, title_prefix='Step␣Size'):
    for key, history in results_dict.items():
        iters = [h[0] for h in history]
        step_sizes = [h[3] for h in history]
        label_str = str(key)
        ax.plot(iters, step_sizes, label=label_str)
    ax.set_title(title_prefix)
    ax.set_xlabel("Iteration")
    ax.set_ylabel("Step␣Size␣(norm)")
    ax.set_yscale('log')
    ax.legend(fontsize='small')
    ax.grid(True)

fig, axs = plt.subplots(3, 2, figsize=(14, 20))
# For f1 (left column)
plot_step_size_results(rmsprop_results_f1, axs[0,0], "RMSProp␣Step␣Size␣on␣f1")
plot_step_size_results(hb_results_f1, axs[1,0], "Heavy␣Ball␣Step␣Size␣on␣f1")
plot_step_size_results(adam_results_f1, axs[2,0], "Adam␣Step␣Size␣on␣f1")
# For f2 (right column)
plot_step_size_results(rmsprop_results_f2, axs[0,1], "RMSProp␣Step␣Size␣on␣f2")
plot_step_size_results(hb_results_f2, axs[1,1], "Heavy␣Ball␣Step␣Size␣on␣f2")
plot_step_size_results(adam_results_f2, axs[2,1], "Adam␣Step␣Size␣on␣f2")


plt.tight_layout()
plt.show()

def contour_plot_with_path(func, history, title='Contour␣Plot', levels=30, ax=None):
    if ax is None:
        fig, ax = plt.subplots(figsize=(8,6))
    points = np.array([h[2] for h in history])
    # Determine region with margin
    x_min, x_max = points[:,0].min()-1, points[:,0].max()+1
    y_min, y_max = points[:,1].min()-1, points[:,1].max()+1
    xs = np.linspace(x_min, x_max, 200)
    ys = np.linspace(y_min, y_max, 200)
    X, Y = np.meshgrid(xs, ys)
    Z = np.zeros_like(X)
    # Evaluate func on grid
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            Z[i,j] = func([X[i,j], Y[i,j]])
    cs = ax.contour(X, Y, Z, levels=levels, cmap='viridis')
    ax.clabel(cs, inline=1, fontsize=8)
    ax.plot(points[:,0], points[:,1], 'ro-', markersize=3)
    ax.set_title(title)
    ax.set_xlabel("x")
    ax.set_ylabel("y")


# For each method, choose a representative parameter key (if available)
# For RMSProp, Heavy Ball, Adam, choose (0.01, 0.9) or (0.01, 0.9, 0.999) if available.
polyak_key = 0 if 0 in polyak_results_f1 else list(polyak_results_f1.keys())[0]
rmsprop_key = (0.01, 0.9) if (0.01, 0.9) in rmsprop_results_f1 else
    list(rmsprop_results_f1.keys())[0]
hb_key = (0.01, 0.9) if (0.01, 0.9) in hb_results_f1 else list(hb_results_f1.keys())[0]
adam_key = (0.01, 0.9, 0.999) if (0.01, 0.9, 0.999) in adam_results_f1 else
    list(adam_results_f1.keys())[0]

fig, axs = plt.subplots(4, 2, figsize=(14, 20))

# Polyak
history_polyak_f1 = polyak_results_f1[polyak_key]
contour_plot_with_path(f1_val, history_polyak_f1, title=f"Polyak␣on␣f1␣
    (f*={polyak_key})", ax=axs[0,0])
history_polyak_f2 = polyak_results_f2[polyak_key]  # use same f* value
contour_plot_with_path(f2_val, history_polyak_f2, title=f"Polyak␣on␣f2␣
    (f*={polyak_key})", ax=axs[0,1])

# RMSProp
history_rmsprop_f1 = rmsprop_results_f1[rmsprop_key]
contour_plot_with_path(f1_val, history_rmsprop_f1, title=f"RMSProp␣on␣f1␣{rmsprop_key}",
    ax=axs[1,0])
history_rmsprop_f2 = rmsprop_results_f2[rmsprop_key]
contour_plot_with_path(f2_val, history_rmsprop_f2, title=f"RMSProp␣on␣f2␣{rmsprop_key}",
    ax=axs[1,1])

# Heavy Ball
history_hb_f1 = hb_results_f1[hb_key]
contour_plot_with_path(f1_val, history_hb_f1, title=f"Heavy␣Ball␣on␣f1␣{hb_key}",
    ax=axs[2,0])
history_hb_f2 = hb_results_f2[hb_key]
contour_plot_with_path(f2_val, history_hb_f2, title=f"Heavy␣Ball␣on␣f2␣{hb_key}",
    ax=axs[2,1])

# Adam
history_adam_f1 = adam_results_f1[adam_key]
contour_plot_with_path(f1_val, history_adam_f1, title=f"Adam␣on␣f1␣{adam_key}",
    ax=axs[3,0])
history_adam_f2 = adam_results_f2[adam_key]
contour_plot_with_path(f2_val, history_adam_f2, title=f"Adam␣on␣f2␣{adam_key}",
    ax=axs[3,1])

plt.tight_layout()
plt.show()


def f_relu(x):
    return max(0.0, x)

def grad_relu(x):
    if x > 0:
        return 1.0
    elif x < 0:
        return 0.0
    else:
        return 0.5



def run_polyak_relu(x_init, f_star=0.0, max_iter=50):
    history = []
    x = x_init
    for it in range(1, max_iter+1):
        f_val = f_relu(x)
        g = grad_relu(x)
        history.append((it, x, f_val))
        x = polyak_update(x, g, f_val, f_star_est=f_star)
    return history

def run_rmsprop_relu(x_init, alpha=0.01, beta=0.9, max_iter=50):
    history = []
    x = x_init
    cache = 0.0
    for it in range(1, max_iter+1):
        f_val = f_relu(x)
        g = grad_relu(x)
        history.append((it, x, f_val))
        x, cache, step = rmsprop_update(x, g, cache, alpha=alpha, beta=beta)
    return history

def run_heavy_ball_relu(x_init, alpha=0.01, beta=0.9, max_iter=50):
    history = []
    x = x_init
    velocity = 0.0
    for it in range(1, max_iter+1):
        f_val = f_relu(x)
        g = grad_relu(x)
        history.append((it, x, f_val))
        x, velocity, step = heavy_ball_update(x, g, velocity, alpha=alpha, beta=beta)
    return history

def run_adam_relu(x_init, alpha=0.01, beta1=0.9, beta2=0.999, max_iter=50):
    history = []
    x = x_init
    m = 0.0
    v = 0.0
    for it in range(1, max_iter+1):
        f_val = f_relu(x)
        g = grad_relu(x)
        history.append((it, x, f_val))
        x, m, v, step = adam_update(x, g, m, v, it, alpha=alpha, beta1=beta1,
            beta2=beta2)
    return history


inits = [-1.0, 1.0, 100.0]

# let's pick alpha=0.1 for RMSProp, Heavy Ball, Adam
# and f_star=0.0 for Polyak.
alpha_val = 0.1

results = {}
for x0 in inits:
    # Store results in a dictionary
    polyak_hist  = run_polyak_relu(x0, f_star=0.0, max_iter=20)
    rms_hist     = run_rmsprop_relu(x0, alpha=alpha_val, beta=0.9, max_iter=20)
    hb_hist      = run_heavy_ball_relu(x0, alpha=alpha_val, beta=0.9, max_iter=20)
    adam_hist    = run_adam_relu(x0, alpha=alpha_val, beta1=0.9, beta2=0.999,
        max_iter=20)
    results[x0]  = {
        "Polyak"  : polyak_hist,
        "RMSProp" : rms_hist,
        "HB"      : hb_hist,
        "Adam"    : adam_hist
    }

for x0, method_dict in results.items():
    print(f"\n=== Initial x={x0} ===")
    for method_name, hist in method_dict.items():
        final_it, final_x, final_f = hist[-1]
        print(f"  {method_name}: final_x={final_x:.4f}, final_f={final_f:.4f}, after 
            {final_it} iters")

def plot_relu_path(history, ax, title="", color="red"):
    # 1. Determine a suitable x-range
    xs_hist = [h[1] for h in history]
    x_min = min(xs_hist) - 1
    x_max = max(xs_hist) + 1

    if abs(x_max - x_min) < 1e-6:
        x_min -= 1
        x_max += 1

    xs = np.linspace(x_min, x_max, 200)
    ys = [max(0.0, x) for x in xs]
    ax.plot(xs, ys, label="ReLU(x)", color="blue")

    its = [h[0] for h in history]
    fvals = [h[2] for h in history]
    ax.plot(xs_hist, fvals, marker='o', color=color, label="Iterates")

    for i in range(len(xs_hist)-1):
```

```python
        ax.arrow(xs_hist[i], fvals[i], xs_hist[i+1]-xs_hist[i], fvals[i+1]-fvals[i],
                 head_width=0.1, length_includes_head=True, color=color, alpha=0.5)

    ax.set_title(title)
    ax.set_xlabel("x")
    ax.set_ylabel("f(x)")
    ax.grid(True)
    ax.legend()

def plot_method_across_inits(results, method_name, inits, figsize=(18,5)):
    fig, axs = plt.subplots(1, len(inits), figsize=figsize)
    for i, x0 in enumerate(inits):
        hist = results[x0][method_name]
        plot_relu_path(hist, axs[i],
                       title=f"{method_name}␣(x0={x0})",
                       color="red")
    plt.tight_layout()
    plt.show()

plot_method_across_inits(results, "Polyak", inits)
plot_method_across_inits(results, "RMSProp", inits)
plot_method_across_inits(results, "HB", inits)
plot_method_across_inits(results, "Adam", inits)
```