Optimisation Algorithms Week 4 Assignment

(a) **Gradient Descent Updates**

(i) Algorithm 1 shows the gradient descent with Polyak step size. The update to the baseline gradient descent algorithm is the updating of alpha with each step. After initialising x and alpha, the algorithm loops through the number of allowed iterations. It first computes the step using the size given by alpha and the direction given by the derivative in that point. Alpha is then updated ahead of the next iteration to account for the gradient - the step size will get smaller when the function is steeper. Finally, the step is made, modifying the current point.

Algorithm 1 Gradient Descent with Polyak step size

$x \leftarrow x_0$
$\alpha \leftarrow \alpha_0$
for $k \in \overline{0, max_{iter}}$ **do**
  $step \leftarrow \alpha * np.array(df(*x))$
  $\alpha \leftarrow np.array(f(*x))/np.dot(np.array(df(*x)), np.array(df(*x)))$
  $x \leftarrow x - step$
end for

(ii) Algorithm 2 shows the gradient descent with the RMSProp steps. The update to the baseline gradient descent algorithm is the method for computing alpha. After initialising x and alpha, the algorithm loops through the number of allowed iterations. While doing so, it maintains and updates a dum that is used to compute alpha. It first computes the step using the size given by the previous alpha and the direction given by the derivative in that point. It then updates the sum by taking into account the gradient in that point, such that the sum becomes larger when the function is steeper. The parameter beta controls the size of the gradient's effect on the sum. Alpha is then updated ahead of the next iteration to account for the gradient - similarly to Polyak, the step size will get smaller when the function is steeper. Finally, the step is made, modifying the current point.

Algorithm 2 Gradient Descent with RMSProp

$x \leftarrow x_0$
$sum \leftarrow 0$
$\alpha \leftarrow \alpha_0$
for $k \in \overline{0, max_{iter}}$ **do**
  $step \leftarrow \alpha * np.array(df(*x))$
  $sum = \beta_1 * sum + (1 - \beta_1) * np.dot(np.array(df(*x)), np.array(df(*x)))$
  $\alpha \leftarrow \alpha_0/sqrt(sum + \epsilon)$
  $x \leftarrow x - step$
end for

(iii) Algorithm 3 shows the gradient descent with the HeavyBall steps. The update to the baseline gradient descent algorithm is the method for computing the steps by accounting for momentum. After initialising x, the algorithm loops through the number of allowed iterations. It computes the step using the size given by alpha and the direction given by the derivative in that point. It also adds a fraction of the previous step size. This causes the step sizes to accelerate with each iteration. Finally, the step is made, modifying the current point.

Algorithm 3 Gradient Descent with Heavy Ball

$x \leftarrow x_0$
for $k \in \overline{0, max_{iter}}$ **do**
  $step \leftarrow \beta_1 * step + \alpha * np.array(df(*x))$
  $x \leftarrow x - step$
end for

(iv) Algorithm 4 shows the gradient descent using Adam. The update to the baseline gradient descent algorithm is the method for computing the steps, and it combines the methods of RMSProp and Heavy Ball. The m variable keeps track of the previous step size, building momentum, and uses the gradient to determine step direction as well as making the step size smaller when close to a minimum. The v variable ensures that the step size doesn't get too big when the function is steep. After initialising x, the algorithm loops through the number of allowed iterations. Variables m and v are updated, and the step is computed. Finally, the step is made, modifying the current point.

Algorithm 4 Gradient Descent with Adam

$x \leftarrow x_0$
$m \leftarrow 0$
$v \leftarrow 0$
for $k \in \overline{0, max_{iter}}$ **do**
$m \leftarrow \beta_1 * m + (1 - \beta_1) * np.array(df(*x))/(1 - \beta_1^{k+1})$
$v \leftarrow \beta_2 * v + (1 - \beta_2) * np.dot(np.array(df(*x)), np.array(df(*x)))/(1 - \beta_2^{k+1})$
$step \leftarrow \alpha * m/(sqrt(v) + \epsilon)$
$x \leftarrow x - step$
end for

(b) **Parameter analysis**

(i) Figure 1 shows the effect of varying the parameters when running the gradient descent algorithm with RMSProp and a maximum of 5000 iterations on function 1. The legend shows the parameters used to generate each line. Alpha was chosen from 0.0025,0.025,0.25,0.5 and beta from 0.25,0.9 to show a wide range of outcomes while keeping the plot legible. For the runs where the algorithm converges, it generally holds that the descent is faster where the function is steeper, and slows down once it reaches the flat region. The red line, corresponding to the smallest alpha, reaches the flat region but runs out of iterations before getting close enough to the minimum, running for approx 40 milliseconds. The yellow line, for which alpha was increased, reaches the minimum in only 1 millisecond (it is not the exact minimum, instead the algorithm automatically stops if the decrease in function value is smaller than 10^5). The same holds for the light green solid line, representing a further increase in alpha and a small beta, although it oscillates significantly more than the yellow line. The dark green dashed line represents the same alpha as the light green line, but with a high beta. It ends up diverging and running out of iterations. The blue lines correspond to the highest alpha. Both of them diverge regardless of the choice of beta.

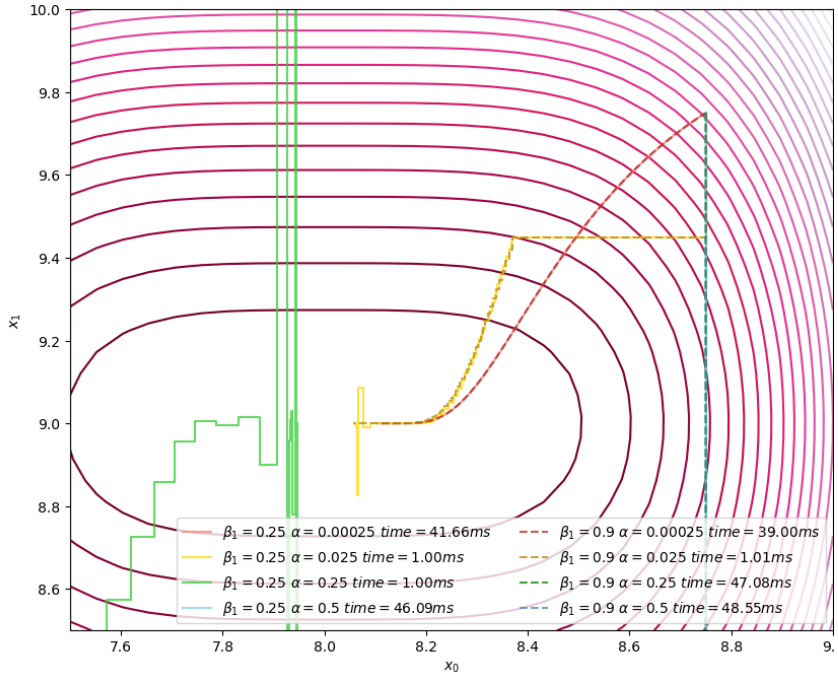

Figure 1: Effect of alpha and beta on the performance of RMSProp on function 1

Figure 2 shows the effect of varying the parameters when running the gradient descent algorithm with RMSProp and a maximum of 5000 iterations on function 2. The legend shows the parameters used to generate each line. Alpha was chosen from 0.00025,0.08,0.25,0.8 and beta from 0.25,0.9 to show a wide range of outcomes while keeping the plot legible. The algorithm stops if the function decrease falls under 10^5. Due to the non-smooth function, all of the lines show ocillation to some degree. The red lines do not get very close to the minimum due to alpha being too small and running out of iterations. The yellow and green lines both get close to the minimum, with green having oscillations of a higher amplitude. This shows that oscillation amplitude is proportional to the step size, which makes sense, since larger steps will cause the algorithm to make larger jumps from one side to the other. Higher beta also seems to cause larger oscillations, but the effect is smaller than that of alpha.

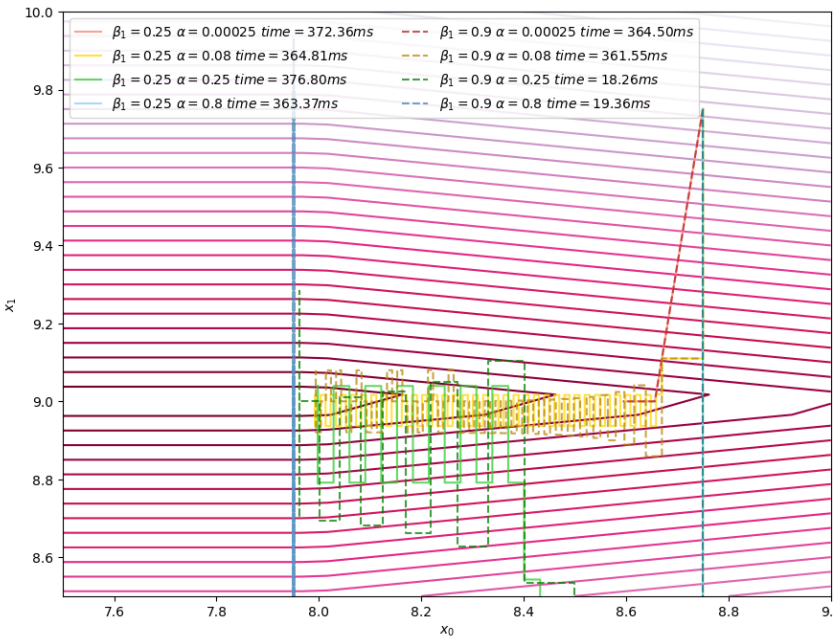

Figure 2: Effect of alpha and beta on the performance of RMSProp on function 2

(ii) Heavy Ball Figure 3 shows the effect of varying the parameters when running the gradient descent algorithm with HeavyBall and a maximum of 5000 iterations on function 1. The legend shows the parameters used to generate each line. Alpha was chosen from 0.002,0.0035,0.1,0.2 and beta from 0.25,0.9 to show a wide range of outcomes while keeping the plot legible. The algorithm stops if the function decrease falls under 10^5. Comparing the red lines to the yellow lines shows the effect of alpha, which is especially visible for the large beta. A smaller step size results in smaller curves, and also runs the risk of running out of iterations. Comparing the solid light coloured red and yellow lines shows the effect of beta, which controls momentum. Higher beta results in the algorithm spinning around more as it approaches the flat area, resembling a ball falling into a funnel. The light green line is very jagged due to its very large step size, but manages to converge the fastest. The dark green high beta counterpart, however, bounces around chaotically and never namages to converge, even after 5000 iterrations. The blue lines, with the highest alpha, diverge right away.

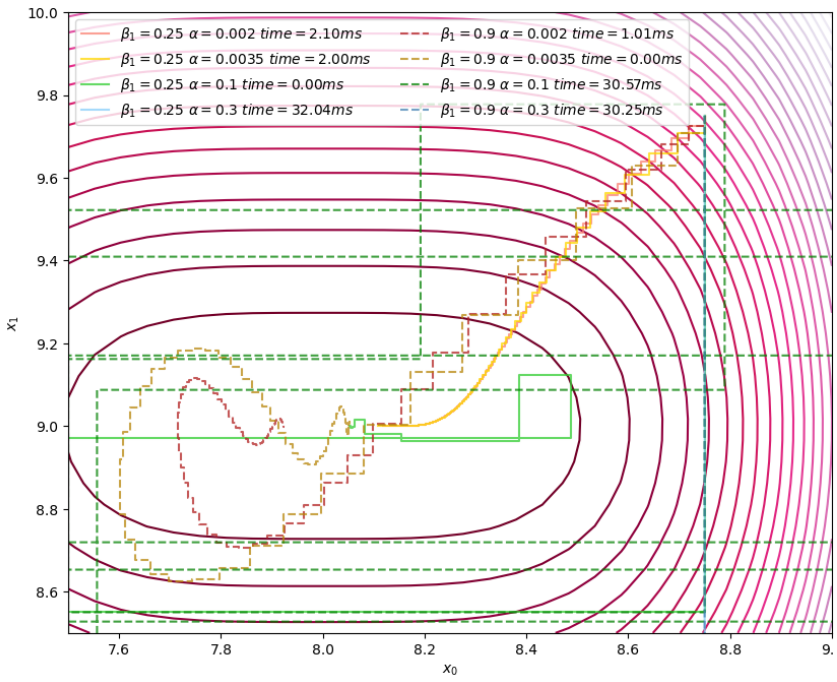

Figure 3: Effect of alpha and beta on the performance of HeavyBall on function 1

Figure 3 shows the effect of varying the parameters when running the gradient descent algorithm with HeavyBall and a maximum of 5000 iterations on function 2. The legend shows the parameters used to generate each line. Alpha was chosen from 0.00001,0.0035,0.1,0.3 and beta from 0.25,0.9 to show a wide range of outcomes while keeping the plot legible. The algorithm stops if the function decrease falls under 10^5. The red line corresponding to the lowest alpha never manages to reach close to the minimum. The light yellow line, with a small alpha and small beta oscillates a lot around the x_1 value corresponding to the minimum and manages to get close to the x_0 threshold corresponding to the minimum values of f2. The dark yellow line, corresponding to a higher beta, shows how the algorithm resembles a ball swirling around a valley. The light green and light blue, high alphas with low momentum, have high and jagged oscillations, characteristic of the large step size. The darker versions corresponding to the high betas diverge away very fast, with very large steps and high momentum.

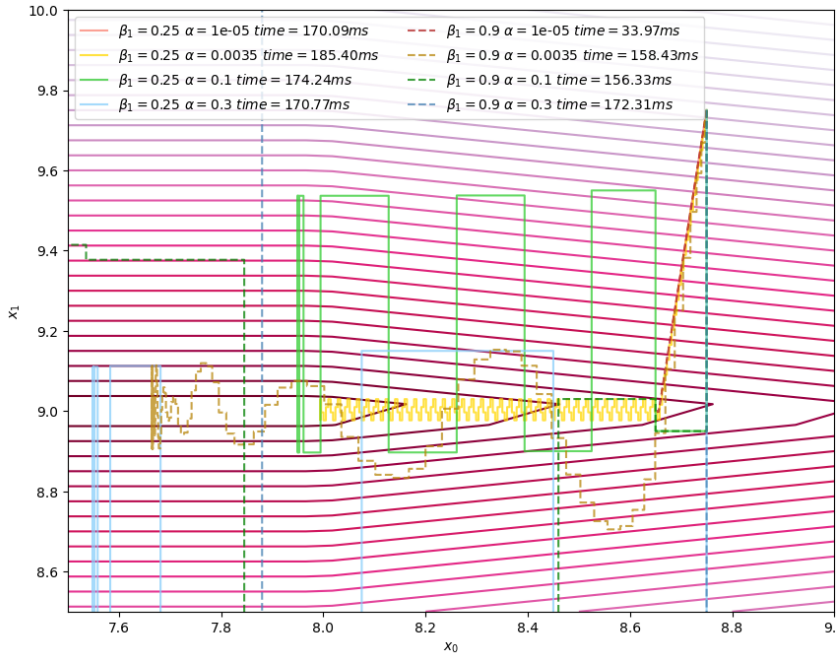

Figure 4: Effect of alpha and beta on the performance of HeavyBall on function 2

Figure 5 shows the effect of varying the parameters when running the gradient descent algorithm with Adam and a maximum of 5000 iterations on function 1. The legend shows the parameters used to generate each line. Alpha was chosen from 0.01,0.025, and $beta_1$ and $beta_1$ from 0.25,0.9 to show a wide range of outcomes while keeping the plot legible. The algorithm stops if the function decrease falls under 10^5. All of the solid line functions, corresponding to low $beta_1$, converge without oscillating, although it takes longer for the small step sized ones to do so. When looking at the dark dashed lines, multiple patters emerge. First of all, the low alpha, high $beta_1$, high $beta_2$ line runs out of iterations, and it does so very fast. Second, The high $beta_2$ lines - red and green - show larger oscillations than their low $beta_2$ counterparts. The green line is the one that takes the longest to find the minimum due to the combination of its parameters. The high $beta_1$ gives it a high momentum, high alpha causes it to have a large step size, and the low $beta_2$, which is supposedto prevent the step size from getting too big, is too small to counteract this. Even so, it still manages to converge.

(iii) Adam

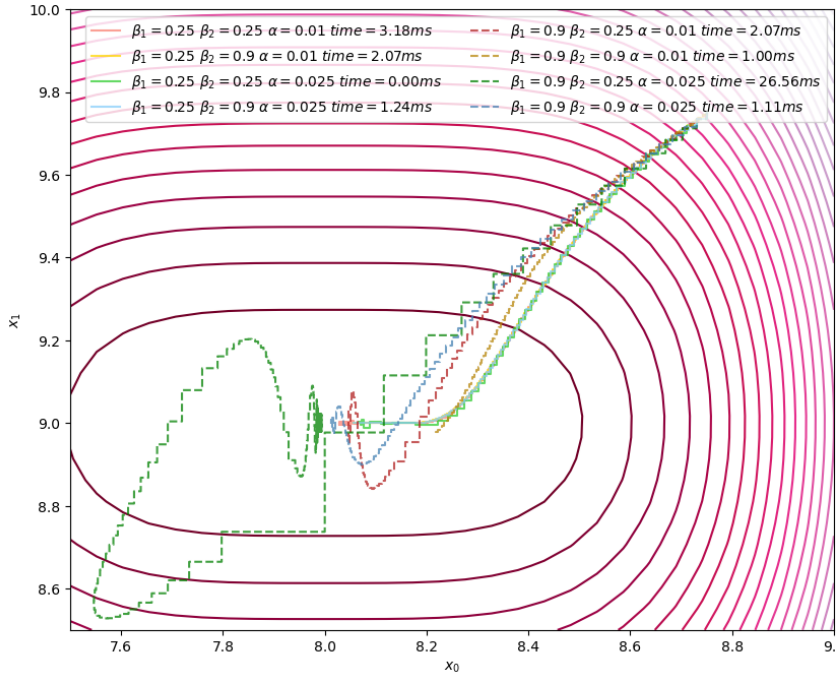

Figure 5: Effect of alpha and beta on the performance of Adam on function 1

Figure 6 shows the effect of varying the parameters when running the gradient descent algorithm with Adam and a maximum of 5000 iterations on function 2. The legend shows the parameters used to generate each line. Alpha was chosen from 0.025,0.8, and $beta_1$ and $beta_2$ from 0.25,0.9 to show a wide range of outcomes while keeping the plot legible. Once again, all of the lines show oscillation around the nondiferrentiable line. The red and yellow lines are very similar, with the distinction of the red lines - low $beta_2$ - having a slightly higher oscillation amplitude. The solid lines - low $beta_1$ - have small, even oscillations, while the dashed lines reflect more of the momentum that is built as the algorithm descends down the valley. The green and blue lines - high alpha - show very large step sizes, with the oscillations once again being slightly larger for the low $beta_2$, and much larger and more rounded for high $beta_1$.

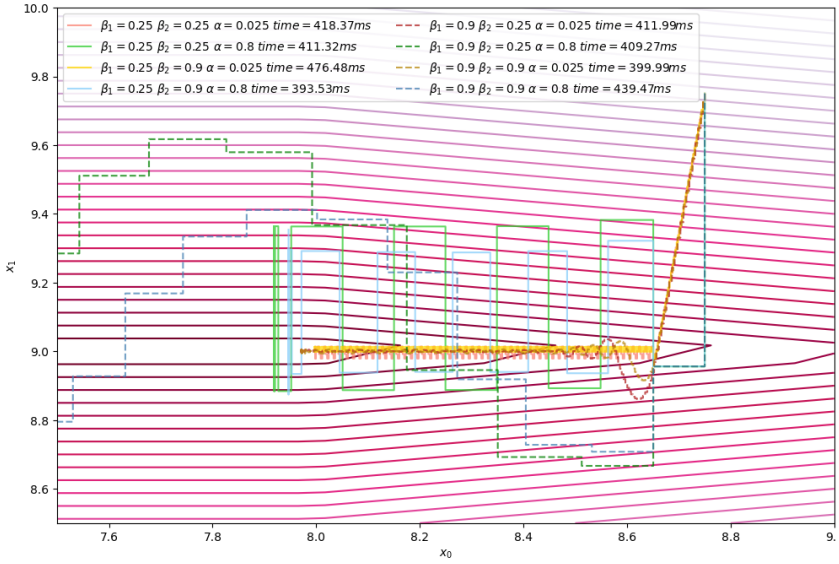

Figure 6: Effect of alpha and beta on the performance of Adam on function 2

(c) **ReLu**

(i) Figure 7 shows the effect of starting at x = -1 when attempting to optimise. Nothing happens because x = -1 already corresponds to a minimum and the gradient is 0.

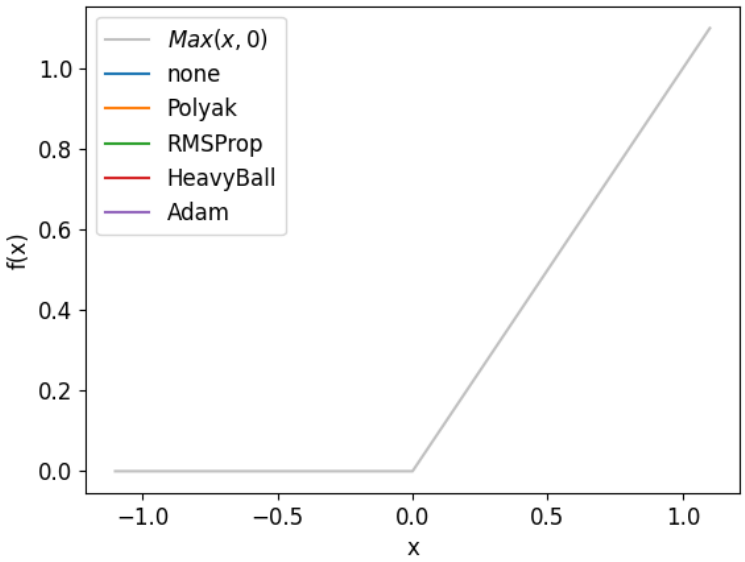

Figure 7: Optimising ReLu starting at -1

(ii) Figure 9 shows the effect of starting at x = 1 when attempting to optimise. The gradient all along the x > 0 portion is equal to 1. All of the algorithms manage to reach a minimum, although HeavyBall overshoots.

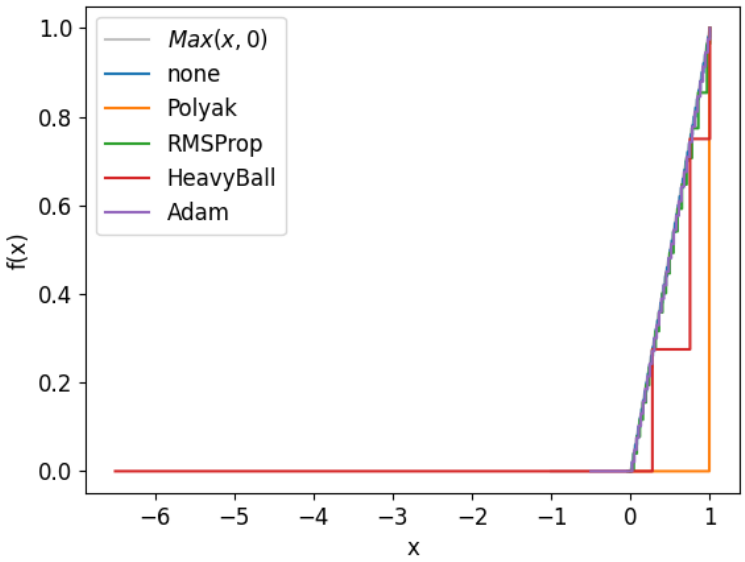

Figure 8: Optimising ReLu starting at 1

(iii) Figure 8 shows the effect of starting at x = 100 when attempting to optimise. The gradient all along the x > 0 portion is equal to 1. All of the algorithms manage to reach a minimum. Momentum causes some step sizes to get progressively bigger. Figure 10 shows algorithm starting at 100 with the plot limits capped at 1. Comparing this to figure 9 shows the effects of momentum, which overshot by a lot, and got so big in som that the last steps of HeavyBall are not visible.

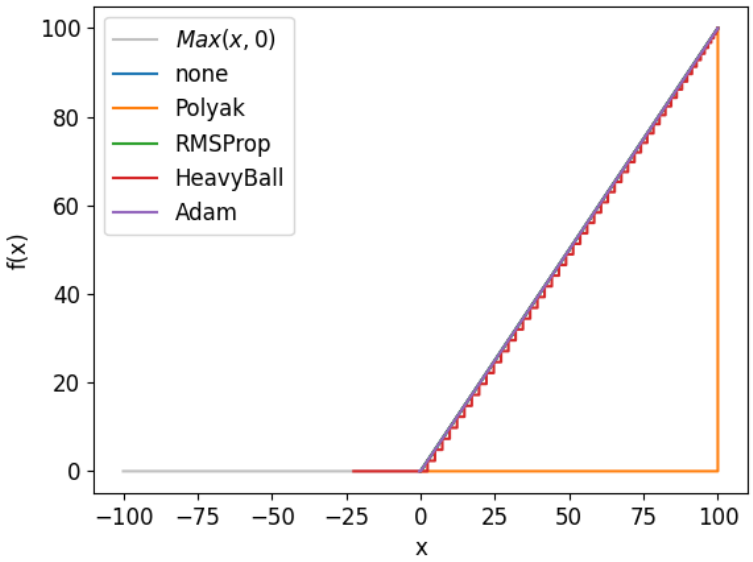

Figure 9: Optimising ReLu starting at 100

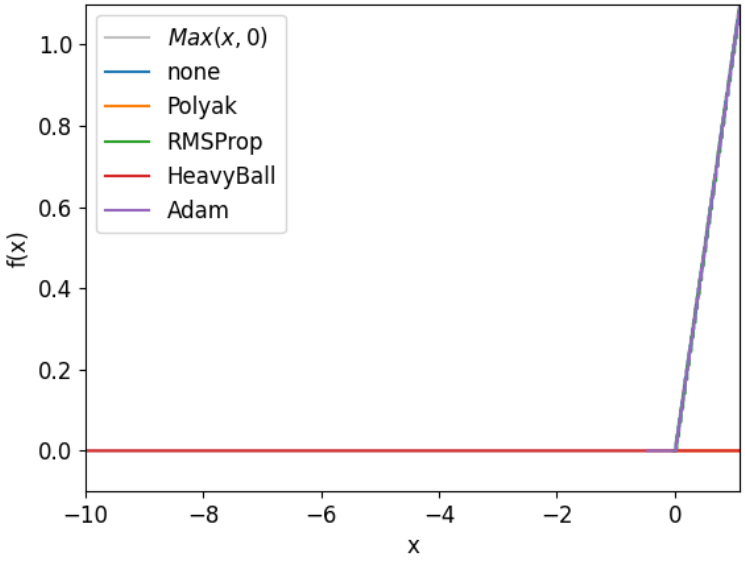

Figure 10: Optimising ReLu starting at 100

Appendix A - Code

```
# function: 9*(x-8)^4+8*(y-9)^2
# function: Max(x-8,0)+8*|y-9|
import time

import sympy
from sympy import Max, Abs
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np
from math import *

# Obtaining derivatives
x0, x1 = sympy.symbols('x0, x1', real=True)
x = sympy.Array([x0, x1])

f1 = 9 * (x[0] - 8) ** 4 + 8 * (x[1] - 9) ** 2
df1dx = sympy.diff(f1, x)
f1 = sympy.lambdify(x, f1)
df1dx = sympy.lambdify(x, df1dx)

f2 = Max(x[0] - 8, 0) + 8 * Abs(x[1] - 9)
df2dx = sympy.diff(f2, x)
f2 = sympy.lambdify(x, f2)
df2dx = sympy.lambdify(x, df2dx)

x = sympy.symbols('x', real=True)
f3 = Max(x, 0)
df3dx = sympy.diff(f3, x)
f3 = sympy.lambdify(x, f3)
df3dx = sympy.lambdify(x, df3dx)


def gradDescentReLu(f, df, x0, alpha0, beta1, beta2, epsilon=0.00001, iters=5000, update='none'):
    x = x0
    X = [x]
    alpha = alpha0
    sum, step, m, v = [0, 0, 0, 0]
    fPrev = f(x)

    for k in range(iters):
        if update == 'Adam':
            m = beta1 * m + (1 - beta1) * np.array(df(x)) / (1 - beta1 ** (k + 1))  # Adam
            v = beta2 * v + (1 - beta2) * np.dot(np.array(df(x)), np.array(df(x))) / (1 - beta2 ** (
            step = alpha * m / (sqrt(v) + epsilon)  # Adam
        elif update == 'HeavyBall':
            print(df(x))
            step = beta1 * step + alpha * np.array(df(x))  # HeavyBall
        else:
            step = alpha * np.array(df(x))  # Default
        if update == 'Polyak':
            alpha = np.array(f(x)) / (epsilon + np.dot(np.array(df(x)), np.array(df(x))))  # Polyak
        elif update == 'RMSProp':
            sum = beta1 * sum + (1 - beta1) * np.dot(np.array(df(x)), np.array(df(x)))  # RMSProp
            alpha = alpha0 / sqrt(sum + epsilon)  # RMSProp
        fPrev = f(x)
        step = alpha * df(x)
```

```
        x = x - step
        X.append(x)
    return X


def gradDescent(f, df, x0, alpha0, beta1, beta2, epsilon=0, iters=5000, update='none'):
    x = np.array(x0)
    X0, X1 = [x[0]], [x[1]]
    alpha = alpha0
    sum, step, m, v = [0, 0, 0, 0]
    fPrev = f(*x)
    print(update)
    for k in range(iters):
        if update == 'Adam':
            m = beta1 * m + (1 - beta1) * np.array(df(*x)) / (1 - beta1 ** (k + 1))  # Adam
            v = beta2 * v + (1 - beta2) * np.dot(np.array(df(*x)), np.array(df(*x))) / (1 - beta2 **
            step = alpha * m / (sqrt(v) + epsilon)  # Adam
        elif update == 'HeavyBall':
            step = beta1 * step + alpha * np.array(df(*x))  # HeavyBall
        else:
            step = alpha * np.array(df(*x))  # Default
        if update == 'Polyak':
            alpha = np.array(f(*x)) / np.dot(np.array(df(*x)), np.array(df(*x)))  # Polyak step
        elif update == 'RMSProp':
            sum = beta1 * sum + (1 - beta1) * np.dot(np.array(df(*x)), np.array(df(*x)))  # RMSProp
            alpha = alpha0 / sqrt(sum + epsilon)  # RMSProp
        fPrev = f(*x)
        x = x - step
        X0.append(x[0])
        X1.append(x[1])
        if abs(fPrev - f(*x)) < 0.00001:
            break
    return [X0, X1]


def plot3D(X, Y, Z):
    norm = plt.Normalize(Z.min(), Z.max())
    colors = cm.PuRd_r(norm(Z))
    ax = plt.axes(projection='3d')
    ax.plot_surface(X, Y, Z, facecolors=colors, shade=False).set_facecolor((0, 0, 0, 0))
    ax.set_xlabel('$x_{0}$')
    ax.set_ylabel('$x_{1}$')
    ax.set_zlabel('f')
    plt.show()


def plotSteps(f, df, x0, linestyle, colour, update='none'):
    if f == f1:
        if update == 'RMSProp':
            beta1List = [0.25, 0.9]
            beta2List = [0]
            alphaList = [0.00025, 0.025, 0.25, 0.5]
        elif update == 'HeavyBall':
            beta1List = [0.25, 0.9]
            beta2List = [0]
            alphaList = [0.002, 0.0035, 0.1, 0.3]
        elif update == 'Adam':
            beta1List = [0.25, 0.9]
```

```
            beta2List = [0.25, 0.9]
            alphaList = [0.01, 0.025]
        else:
            beta1List = [0]
            beta2List = [0]
            alphaList = [0.01]
    else:
        if update == 'RMSProp':
            beta1List = [0.25, 0.9]
            beta2List = [0]
            alphaList = [0.00025, 0.08, 0.25, 0.8]
        elif update == 'HeavyBall':
            beta1List = [0.25, 0.9]
            beta2List = [0]
            alphaList = [0.00001, 0.0035, 0.1, 0.3]
        elif update == 'Adam':
            beta1List = [0.25, 0.9]
            beta2List = [0.25, 0.9]
            alphaList = [0.025, 0.8]
        else:
            beta1List = [0]
            beta2List = [0]
            alphaList = [0.01]
    for b1, beta1 in enumerate(beta1List):
        for a, alpha in enumerate(alphaList):
            for b2, beta2 in enumerate(beta2List):
                start = time.time() * 1000
                steps = gradDescent(f, df, x0, alpha, beta1, beta2, update=update)
                stop = time.time() * 1000
                ax.step(steps[0], steps[1], linestyle=line[b1], color=colour[2 * a + b2][b1], alpha=(
                            r'$\beta_{1} = $' + str(beta1) + r' $\beta_{2} = $' + str(beta2) + r' $\
                        alpha) + r' $time = %.2f ms$' % (stop - start)))  #


def plotReLu(x0, f, df, colours):
    plt.rcParams['font.size'] = 12
    fig, ax = plt.subplots()
    x = np.linspace(-x0, x0, 3)
    y = np.maximum(x, 0)
    # print(xx)
    # ax.plot(x, y, 'r')
    ax.plot(x, np.maximum(x, 0), color='silver', label=(r'$Max(x,0)$'))
    alpha = [0.02, 0.015, 0.07, 0.5, 0.05]
    beta1 = [0.01, 0.01, 0.999, 0.999, 0.999]
    beta2 = [0.01, 0.01, 0.01, 0.01, 0.9]
    #ax.set_xlim(-1, 1)
    #ax.set_ylim(-0.1, 1)
    for u, update in enumerate(['none', 'Polyak', 'RMSProp', 'HeavyBall', 'Adam']):
        X = gradDescentReLu(f, df, x0, alpha[u], beta1[u], beta2[u], update=update)
        ax.step(X, np.maximum(X, 0), label=update)

    ax.set_xlabel('x')
    ax.set_ylabel('f(x)')
    ax.legend()
    plt.show()


#fig, ax = plt.subplots()
```

```
colour = [['salmon', 'firebrick'], ['gold', 'darkgoldenrod'], ['limegreen', 'green'], ['lightskyblue
line = ['-', '--']

x0, df = [8.75, 9.75], df1dx  # f1
x = np.linspace(7.5, 9, 30)
y = np.linspace(8.5, 10, 30)
X, Y = np.meshgrid(x, y)

f = f3
df = df3dx
# Z = f(X, Y) #f1
# Z = np.maximum(X - 8, 0) + 8 * np.abs(Y - 9) #f1

'''ax.contour(X, Y, Z, levels=30, cmap='PuRd_r')
plotSteps(f,df,x0,line,colour,update ='Adam') #Select the update

ax.legend(ncol=2)
ax.set_xlabel('$x_{0}$')
ax.set_ylabel('$x_{1}$')
ax.set_xlim(7.5, 9)
ax.set_ylim(8.5, 10)
plt.show()'''

# plot3D(X,Y,Z)

plotReLu(100, f, df, colour)
```