Week 4 Assignment

Brendan Jobus

18321740

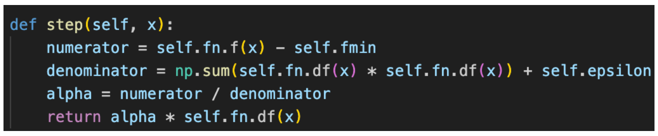
Function 1: 5*(x-8)^4 + 6*(y-3)^2

Function 2: Max(x-8, 0) + 6*|y-3|

Part A

For all of the algorithms, I set up a python class to implement them, where they are instantiated with the function that we are running the algorithms on, as well as any hyper parameters that the algorithms use(f*, alpha, beta, etc.). They all have a function called step, that does the relevant step calculation(i.e. for Polyak this would be $\alpha \frac{df}{dx} x$), this is just to make the code behave nicely with my gradient descent algorithm due to the fact that Adam and HeavyBall compute the step differently to the other two. Each function is also defined by a class, with a function for $f(x)$ and $\frac{df}{dx} x$ which calls a function for each partial derivative if the function is multivariate.

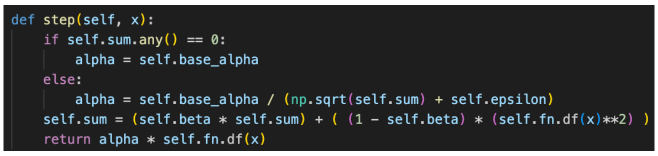
Polyak



```python
def step(self, x):
    numerator = self.fn.f(x) - self.fmin
    denominator = np.sum(self.fn.df(x) * self.fn.df(x)) + self.epsilon
    alpha = numerator / denominator
    return alpha * self.fn.df(x)
```

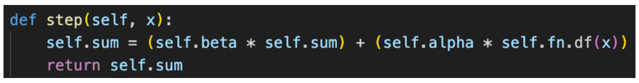
For polyak, the numerator is very basic, just a call to the function with the current x, then subtracted from the supplied fmin(or fstar). The denominator uses numpy ndarrays(to account for case of a multivariate function being run) to do element wise squaring, and then summing them, in order to implement $\nabla f(x)^T \nabla f(x)$.

RMSProp



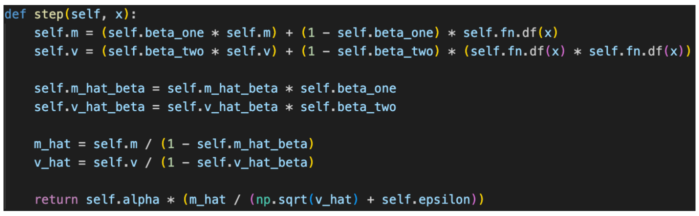
```python
def step(self, x):
    if self.sum.any() == 0:
        alpha = self.base_alpha
    else:
        alpha = self.base_alpha / (np.sqrt(self.sum) + self.epsilon)
    self.sum = (self.beta * self.sum) + ( (1 - self.beta) * (self.fn.df(x)**2) )
    return alpha * self.fn.df(x)
```

For RMSProp, I had to set up a first pass condition, due to np.sqrt, which I use to do similar element wise square rooting, spitting out an error when the sum value is 0. Other than that, its very basic, an update being added to sum after each update to alpha. Note that sum here holds a vector to handle the multivariate functions(θ_1, θ_2 in this case), and is the reason for np.sqrt being used.

HeavyBall 

```
def step(self, x):
    self.sum = (self.beta * self.sum) + (self.alpha * self.fn.df(x))
    return self.sum
```

HeavyBall is much simpler due to the lack of a square root, and since the sum itself is our step size. We simply do the same sum line from before, except without scaling by $\dfrac{1}{1-\beta}$, and multiplying the derivative by alpha instead of itself.

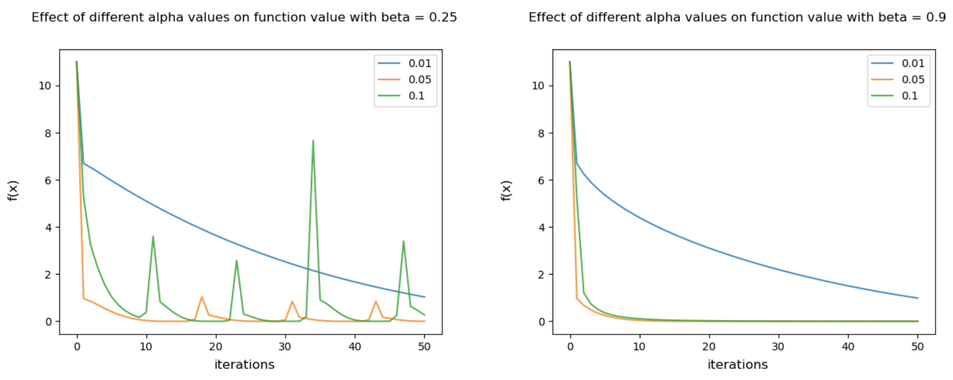
Adam 

```
def step(self, x):
    self.m = (self.beta_one * self.m) + (1 - self.beta_one) * self.fn.df(x)
    self.v = (self.beta_two * self.v) + (1 - self.beta_two) * (self.fn.df(x) * self.fn.df(x))

    self.m_hat_beta = self.m_hat_beta * self.beta_one
    self.v_hat_beta = self.v_hat_beta * self.beta_two

    m_hat = self.m / (1 - self.m_hat_beta)
    v_hat = self.v / (1 - self.v_hat_beta)

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

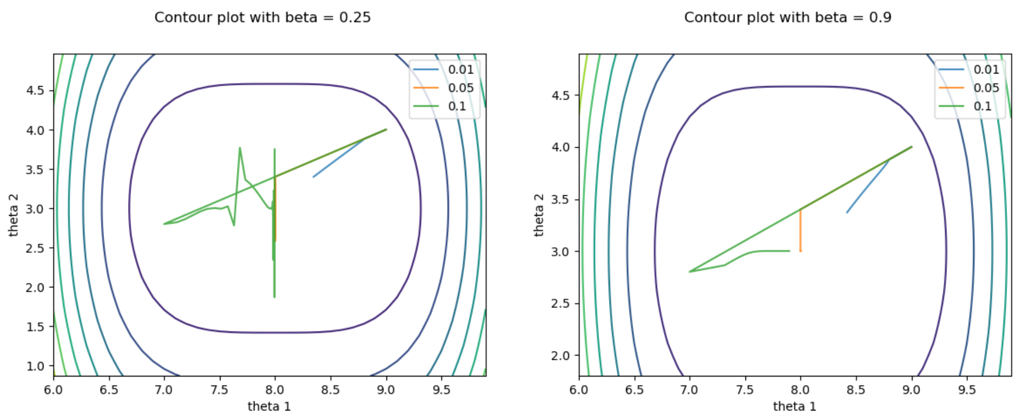
For Adam, m and v are basically the same as the sums from the last two functions, with m_hat and v_hat being basically HeavyBall and RMSProp respectively. To implement the denominator required to calculate m_hat, I didn't want to keep a count of the number of passes that have occurred, so instead chose to just store the previous β^t from each pass and update it during each pass(note that m_hat_beta is instantiated as 1 to account for the first pass of β^1. Then I use np.sqrt again as m_hat and v_hat are running totals of an ndarray of derivatives, since we have a partial derivative for each variable.

Part B

i)

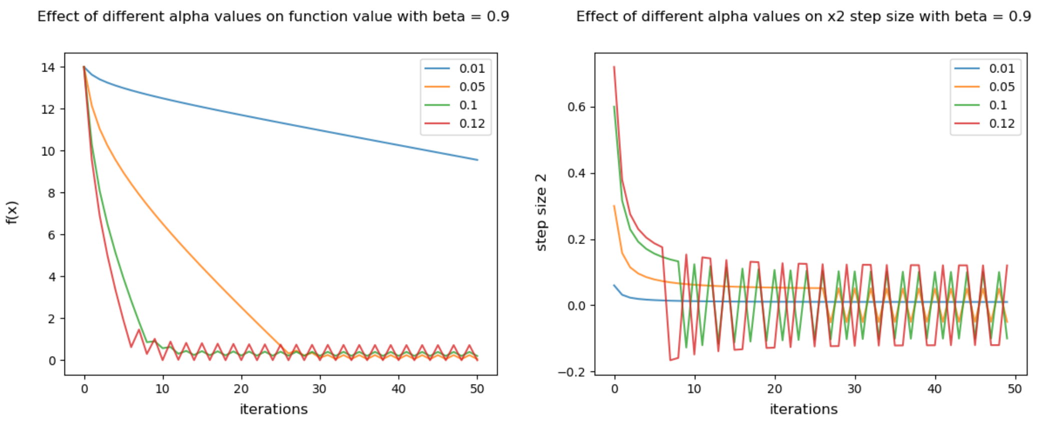


With these two plots, we see an obvious difference in that there are less spikes with higher alpha values with a larger beta value, logically, this should mean that using a smaller beta value forces us to use smaller step sizes. This makes sense due to the fact that our alpha value is being divided by a running average of the gradients weighted by beta, and with a smaller beta, the past results, and thus our denominator, gets smaller as our gradients get smaller, so if the function we are trying to minimise has a very flat area around the minimum, with a small beta, we need to have a smaller alpha as well, since the smaller beta will make our denominator smaller, increasing the size of our step sizes, perhaps too much. And if we look at a contour plot, thats what we see at our larger alphas with a smaller beta.

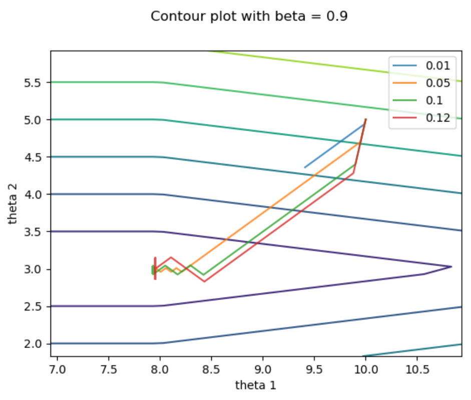


Here we can see that with beta = 0.25, very obviously alpha = 0.1 is not converging on theta 2, and the same can also be said for alpha = 0.05. Even with the larger beta = 0.9, we can see that the larger alpha doesn’t converge fast because it overshot the minimum for theta 2.

For the second function, we get similar results, with an increased convergence speed, but RMSProp begins to struggle when it encounters a non convex function, in this case the absolute function. And we can see this if we look at the function against iteration and step size 2 against iteration plots.

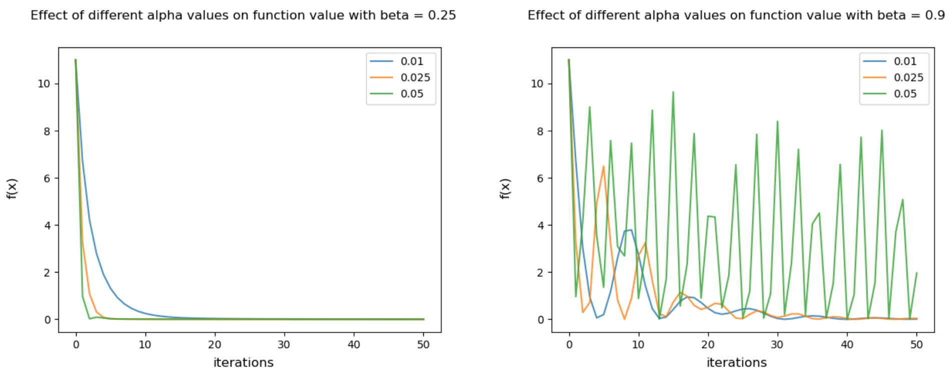


Here we see that, because of the constant gradient of the absolute functions derivative, we run we’re forced to keep the step size much smaller in order to reduce the oscillations, however, if our step size is too small, it takes far too long to converge.

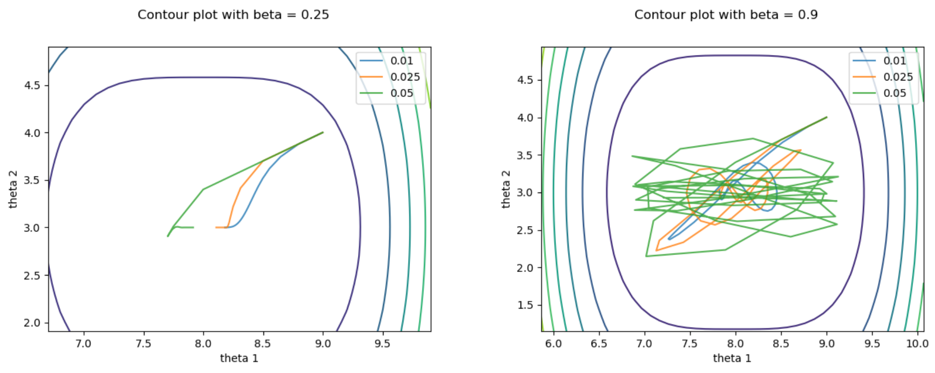


This contour plot shows us this much better, and shows that, when we choose a larger alpha, we control not just how fast we converge, but also how far we deviate from the minimum during the oscillations.

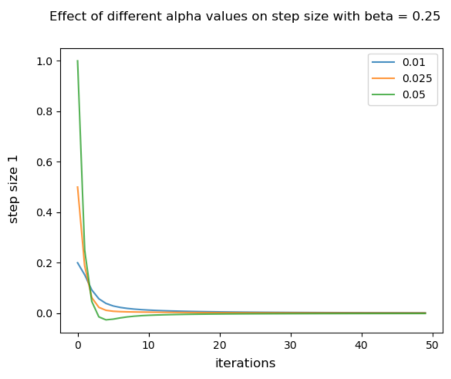
ii)

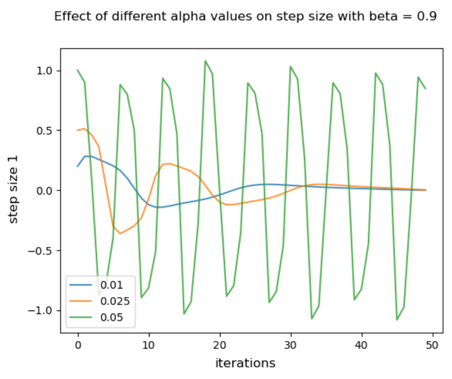


With HeavyBall, we see that there has been a flip from the RMSProp, where now, the higher beta value appears to have much more trouble converging for higher alpha values than for the smaller beta value. And if we look at the contour plot we can see that that's true.

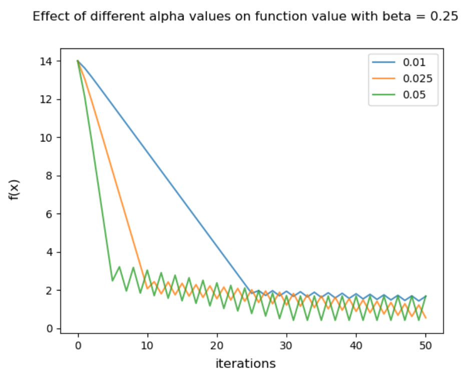


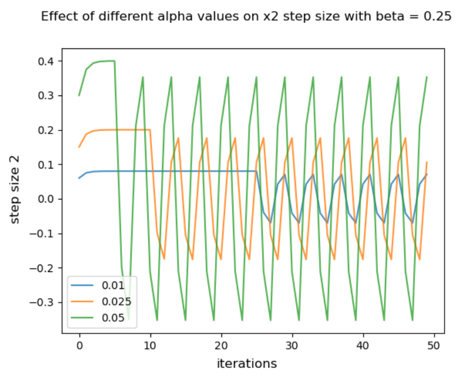
Here we can see that the larger beta value cause us to overshoot the local minimum with every alpha value. We find this issue because unlike with RMSProp which uses a weighted running average of the square gradient, HeavyBall uses a weighted running average of the step sizes instead, meaning that our average now takes into account the sign of past gradients. This can result in a situation where it takes longer to recover from overshooting the minimum, where the momentum of the running average keeps our steps positive(or negative) for longer than we would want, making us overshoot the minimum even more. This explains why the larger beta results in more overshooting, since it increases the impact of past steps on the current, and the largest alpha would therefore increase the size of those steps as well. If we look at the step size of variable 1 against iterations, we see this stops us from converging for an alpha = 0.05 with beta = 0.9.



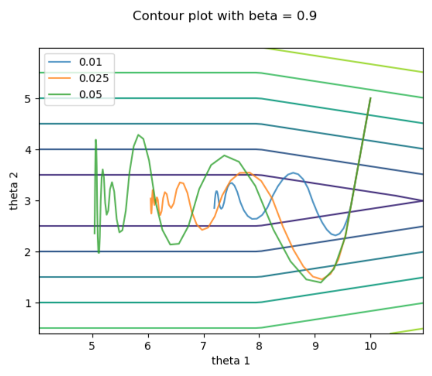


For the second function, we see the overshooting has occurred slightly, even with the smaller beta value, but has landed us in a never ending loop for the largest alpha and beta value.



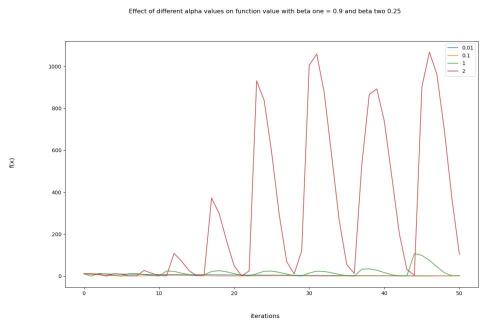


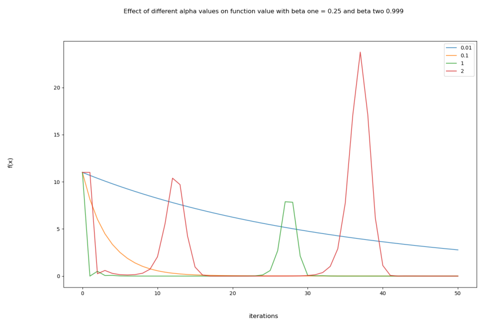
HeavyBall has similar issues, for similar reasons, dealing with the non convex, constant gradient of the absolute function. I've chosen to use beta = 0.9 here because the momentum feature of heavy ball actually causes further issues with the part of the function that we didn't have much of a problem with before, now, the momentum causes us to overshoot the minimum of theta 1 as well.

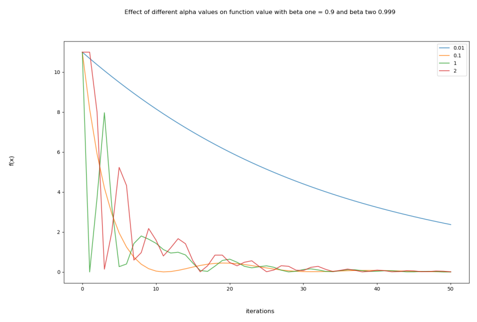


Keeping in mind that the minimum of theta 1 is 8, you can see that we heavily overshoot the minimum, and that this overshoot scales with the value we choose for alpha and beta. This is due to the interaction between momentum based approaches and the derivative of the maximum function, the Heaviside function, which is essentially a one sided function, where we can move on one side but not the other, this is covered more in part C, but is essentially due to the momentum taking time to cause the earlier values in the running average to become non-influential on the steps we take, while the current input is 0. Essentially, momentum based approaches need a convex function that will allow the algorithm to bounce back and forth past the minimum, getting closer every pass.

iii)

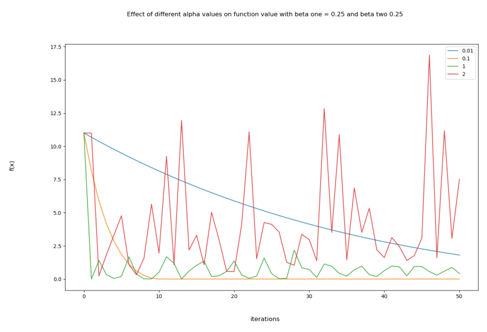


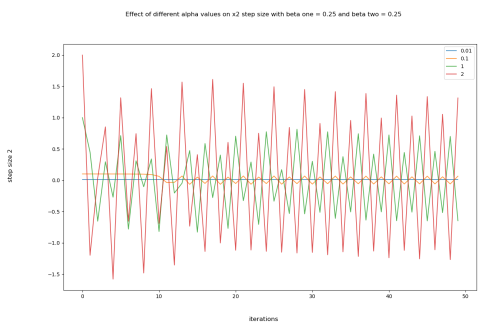


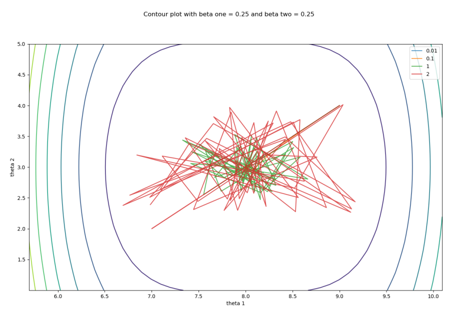


There is a clear pattern here that smaller beta one or two values causes the irregularities in larger alpha values. Having a smaller version of either causes us difficulties due to one being a weight for the running average of gradients, and the other a weight for the running average of square gradients. So if beta one is small, then we will respond better to a change in gradient signs, since we won't be past gradients won't be giving as much input to our next step, and if beta two is small, we will respond faster to a change in shape(i.e. going from a steeper to flatter section). Both have different effects depending on the function we are applying it to, as if we are better at changing sign or shape, we may respond too quickly and end up reducing the size of our steps too much. This can be seen in the next three plots together.

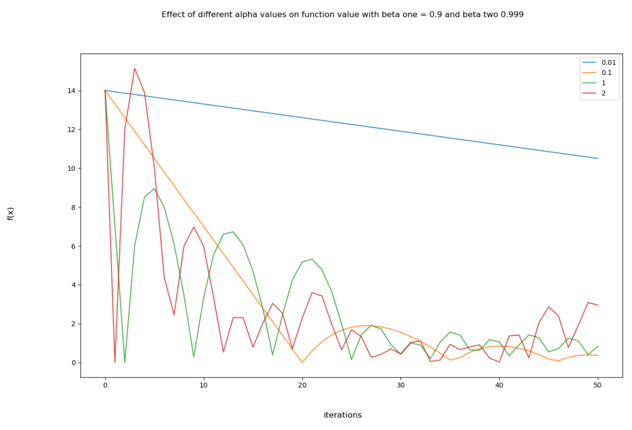
Here, where both beta one and two are small, we are overshooting massively all the time, with the same values of alpha as the previous plots. Since past

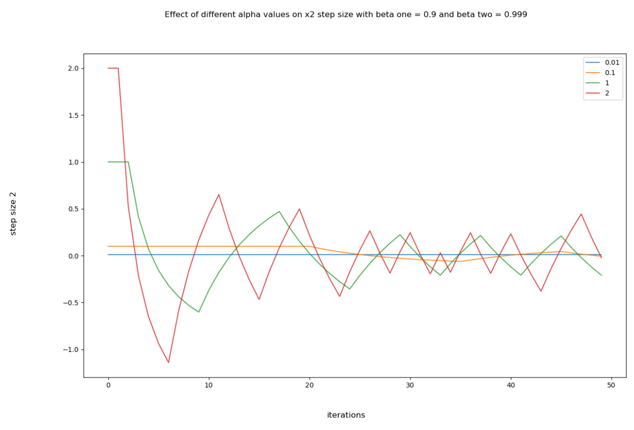




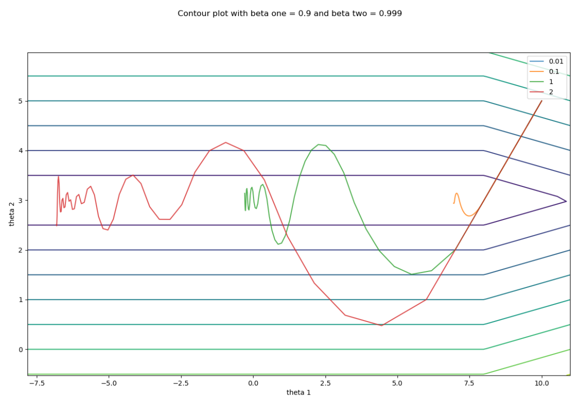


gradients and square gradients aren't being weighted, now if we overshoot, instead of it being the weighted average of our past lets say 5 values, its now just our current and last value. So if the past 5 we're all positive, we would turn around slower, but would take a smaller step in the other direction when we did turn around, whereas if we weight less, we might turn around quicker, but we'll also take a larger step in the other direction. This causes us to be less stable at dealing with overshooting, and thus we must reduce our value of alpha.





With these graphs, we can see that we appear to be dealing similarly with the problem of theta 2 in the second function, even with the larger alpha value, however, the same issue with HeavyBall and theta 1 is present here, but since our step sizes are orders of magnitude larger, we overshoot by orders of magnitude more as well.

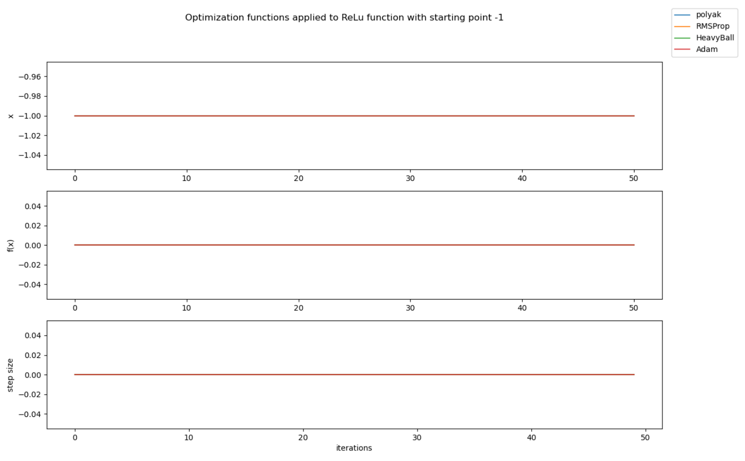


Part C

To choose the hyper parameters for this section, I looked at two graphs from the second function, function value against iterations and step size 1 against iterations. I did this because, since all of these algorithms create separate step sizes for each variable, and the function applied to the first variable in the second function is the same as Relu's function (i.e. Max), then function value against iteration should be a decent indicator of how fast these algorithms converge with this function(although they will be heavily affected by the second function, which as shown above, they struggled to converge with), by looking at the step size 1 against iterations as well, we should get a good idea as to how stable they are for that function, though we still cant be entirely sure. (I ran out of space here for the plots of step size 1 against iterations for the second function)

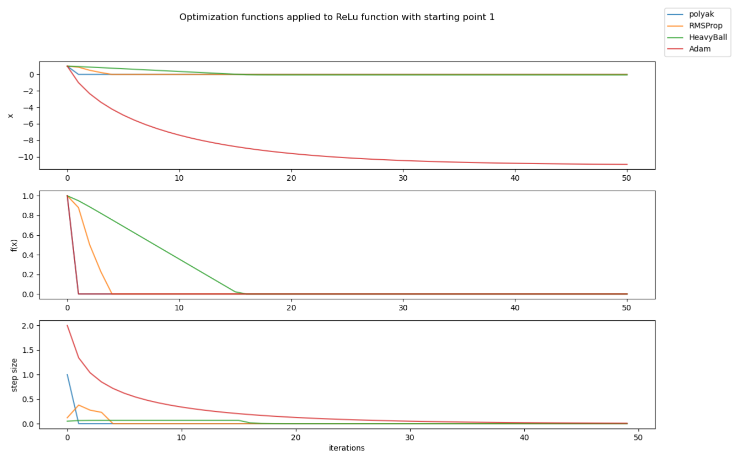
For RMSProp, I chose alpha = 0.12 and beta = 0.9. For HeavyBall, alpha = 0.05 and beta = 0.25 and for Adam, alpha = 2, beta one = 0.9 and beta two = 0.999

Starting point -1



With starting point -1, we run into a very similar situation to the dead neuron problem, even though we aren't working with a neural net. Since the derivative of Max() function is Heaviside, when our x is negative, the derivative is -1, so our functions can never move, since we essentially have a completely flat area from 0 to -∞.

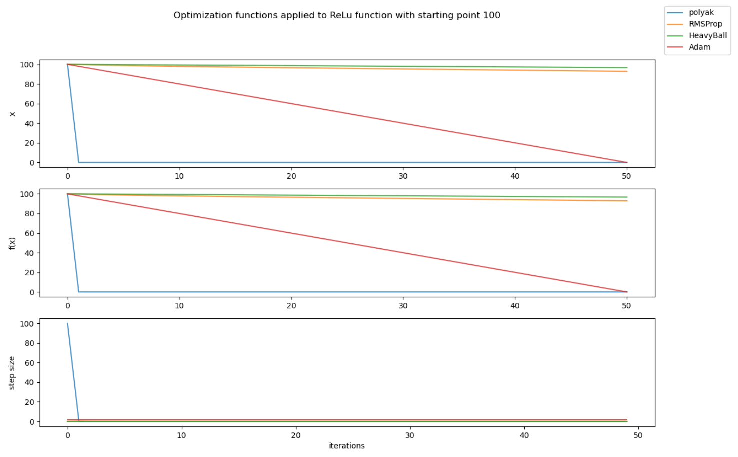
Starting point 1



When the starting point is 1, polyak and Adam converge the fastest with RMSProp not too far behind and HeavyBall lagging quite a bit. Whats interesting however, is that as can be see from the first plot, Adam continues to move even after reaching the minimum. This is due to the fact that Adam uses previous gradients and step sizes to help choose its next step, so once it reaches the min, and the gradient is 0, it will still move due to the previous steps and gradients. This is an issue here as the Max function is not convex, and has no opposing gradients on the other side of the minimum to balance out this effect, so we have to wait for the effect of the scaling betas to make the previous steps and gradients irrelevant. Both RMSProp and HeavyBall also have this issue, although it can't be seen here because their step sizes are so small that the steps peter out much quicker.

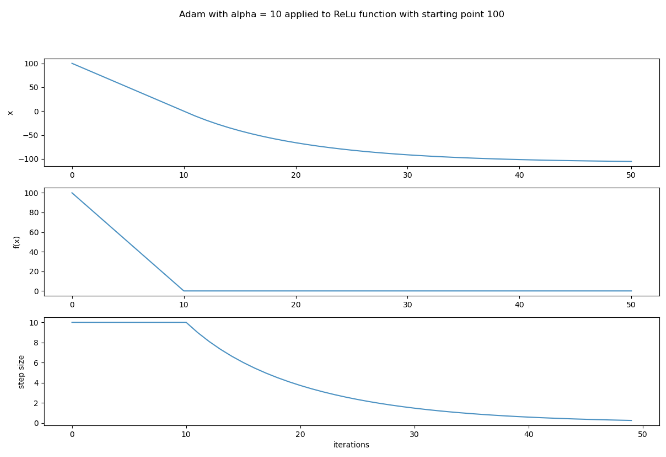
In comparison, polyak doesn't struggle at all, since it bases its step sizes off of its current function value in comparison to the minimum, and once it reaches the minimum or passes it, the function value is always 0, and so it will always be dividing something into 0, which gives 0, and therefore the step sizes will always be 0, and halt the algorithm.

Starting point 100



Finally, with the starting point of 100, and the minimum being 0, we are now looking at a situation in which we start extremely far away from the minimum. Polyak, doesn't struggle at all here, due to the function value in comparison to the minimum being very large, but the derivative being very small, it will take very large steps, and again, halt at the minimum.

Here, we see that RMSProp and HeavyBall have a hugely difficult time with being so far away, due to the fact that we have a constant gradient of simply 1, and they have a relatively small step size. In comparison, Adam, which is able to use a much larger step size, actually manages to converge, thought it takes a long time. However, while it might look like if we just crank up the step sizes then Adam could rival polyak, it actually just ends up exacerbating the issue we ran into in the last section as seen here, when we increase our alpha value to 10. This would also happen in the previous plots, if we had gone through more iterations.



Appendix

```
import numpy as np
import matplotlib.pyplot as plt
from sympy import *

# function: 5*(x-8)^4+6*(y-3)^2
# function: Max(x-8,0)+6*|y-3|

# x1, x2 = symbols("x1 x2")
# f1 = 5 * (x1 - 8)**4 + 6 * (x2 - 3)**2
# f2 = Max(x1 - 8, 0) + 6 * Abs(x2 - 3)
# f1 = function()
# f2 = function(f2)

def vars():
    x, y = symbols("x y", real = True)
    f1 = 5 * (x - 8)**4 + 6 * (y - 3)**2
    f2 = Max(x - 8, 0) + 6 * Abs(y - 3)

    df1dx = diff(f1, x)
    df1dy = diff(f1, y)
    df2dx = diff(f2, x)
    df2dy = diff(f2, y)

    print("df1/dx:", df1dx)
    print("df1/dy:", df1dy)
    print("df2/dx:", df2dx)
    print("df1/dy:", df2dy)

class function_one():
    def __init__(self):
        self.name = "5(x - 8)^4 + 6(y - 4)^2"

    def f(self, x):
        return 5 * (x[0] - 8)**4 + 6 * (x[1] - 3)**2

    def fx(self, x, y):
        return 5 * (x - 8)**4 + 6 * (y - 3)**2

    def df(self, x):
        return np.array([self.dfdx0(x[0]), self.dfdx1(x[1])])

    def dfdx0(self, x0):
        return 20 * (x0 - 8)**3

    def dfdx1(self, x1):
        return 12 * x1 - 36

class function_two():
    def __init__(self):
        self.name = "Max(x - 8, 0) + 6|y - 3|"
    def f(self, x):
        return np.maximum(x[0] - 8, 0) + 6 * np.abs(x[1] - 3)

    def fx(self, x, y):
        return np.maximum(x - 8, 0) + 6 * np.abs(y - 3)

    def df(self, x):
        return np.array([self.dfdx0(x[0]), self.dfdx1(x[1])])

    def dfdx0(self, x0):
        return np.heaviside(x0 - 8, 0.5)

    def dfdx1(self, x1):
        return 6 * np.sign(x1 - 3)

class ReLu_function():
    def __init__(self):
        self.name = "ReLu"

    def f(self, x):
        return np.maximum(0, x[0])

    def df(self, x):
        return np.heaviside(x, 0.5)

class Polyak():
    def __init__(self, fn, minf, epsilon = 0.0001):
        self.fn = fn
        self.fmin = minf
        self.epsilon = epsilon

    def step(self, x):
        numerator = self.fn.f(x) - self.fmin
        denominator = np.sum(self.fn.df(x) * self.fn.df(x)) +
self.epsilon
        alpha = numerator / denominator
        return alpha * self.fn.df(x)

class RMSProp():
    def __init__(self, fn, base_alpha, beta, base_sum =
np.array([0, 0]), epsilon = 0.001):
        self.fn = fn
        self.base_alpha = base_alpha
        self.beta = beta
        self.sum = base_sum
        self.epsilon = epsilon

    def step(self, x):
        if self.sum.any() == 0:
            alpha = self.base_alpha
        else:
            alpha = self.base_alpha / (np.sqrt(self.sum) +
self.epsilon)
```

```python
        self.sum = (self.beta * self.sum) + ( (1 - self.beta)
* (self.fn.df(x)**2) )
        return alpha * self.fn.df(x)

class HeavyBall():
    def __init__(self, fn, base_alpha, beta):
        self.fn = fn
        self.alpha = base_alpha
        self.beta = beta
        self.sum = 0

    def step(self, x):
        self.sum = (self.beta * self.sum) + (self.alpha *
self.fn.df(x))
        return self.sum

class Adam():
    def __init__(self, fn, alpha, b1, b2, epsilon = 0.00001):
        self.fn = fn
        self.alpha = alpha
        self.beta_one = b1
        self.beta_two = b2
        self.m_hat_beta = 1
        self.v_hat_beta = 1
        self.m = 0
        self.v = 0
        self.epsilon = epsilon

    def step(self, x):
        self.m = (self.beta_one * self.m) + (1 -
self.beta_one) * self.fn.df(x)
        self.v = (self.beta_two * self.v) + (1 -
self.beta_two) * (self.fn.df(x) * self.fn.df(x))

        self.m_hat_beta = self.m_hat_beta * self.beta_one
        self.v_hat_beta = self.v_hat_beta * self.beta_two

        m_hat = self.m / (1 - self.m_hat_beta)
        v_hat = self.v / (1 - self.v_hat_beta)

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

def gradientDescent(fn, x0, alpha, single_variable = False,
num_iters=50):
    xt = x0
    if not single_variable:
        X1 = np.array([xt[0]])
        X2 = np.array([xt[1]])
        F = np.array(fn.f(xt))
        A1 = np.array([])
        A2 = np.array([])
    else:
        X = np.array(xt)
        F = np.array(fn.f(xt))
        A = np.array([])

    for _ in range(num_iters):
        step = alpha.step(xt)
        xt1 = xt - step
        xt = xt1
        if not single_variable:
            X1 = np.append(X1, [xt[0]], axis = 0)
            X2 = np.append(X2, [xt[1]], axis = 0)
            F = np.append(F, fn.f(xt))
            A1 = np.append(A1, step[0])
            A2 = np.append(A2, step[1])
        else:
            X = np.append(X, [xt[0]], axis = 0)
            F = np.append(F, fn.f(xt))
            A = np.append(A, step)

    if not single_variable:
        return(X1, X2, F, A1, A2)
    else:
        return (X, F, A)

def basic_polyak(fn, x, fmin):
    alpha = Polyak(fn, fmin)
    (X, F, A) = gradientDescent(fn, x, alpha, single_variable
= True)
    return (X, F, A)

def basic_RMSProp(fn, x):
    b = 0.9
    a0 = 0.12

    alpha = RMSProp(fn, a0, b, base_sum = np.array([0]))
    (X, F, A) = gradientDescent(fn, x, alpha, single_variable
= True)

    return (X, F, A)

def basic_HB(fn, x):
    a0 = 0.05
    b = 0.25
    alpha = HeavyBall(fn, a0, b)
    (X, F, A) = gradientDescent(fn, x, alpha, single_variable
= True)
    return(X, F, A)

def basic_Adam(fn, x, alpha = 2, beta_one = 0.9, beta_two =
0.999):
    alpha = Adam(fn, alpha, beta_one, beta_two)
    return gradientDescent(fn, x, alpha, single_variable =
True)

def plot_basic_implementations(fn, x, fmin):
    (polyX, polyF, polyA) = basic_polyak(fn, x, fmin)
    (rmsX, rmsF, rmsA) = basic_RMSProp(fn, x)
    (hbX, hbF, hbA) = basic_HB(fn, x)
    (adamX, adamF, adamA) = basic_Adam(fn, x)

    fig = plt.figure()
    iterations = range(51)
    stepsize_iters = range(50)
    ax1 = fig.add_subplot(311)
    ax2 = fig.add_subplot(312)
    ax3 = fig.add_subplot(313)

    ax1.plot(iterations, polyX, alpha=0.8)
    ax2.plot(iterations, polyF, alpha=0.8)
    ax3.plot(stepsize_iters, polyA, alpha=0.8)

    ax1.plot(iterations, rmsX, alpha=0.8)
    ax2.plot(iterations, rmsF, alpha=0.8)
    ax3.plot(stepsize_iters, rmsA, alpha=0.8)

    ax1.plot(iterations, hbX, alpha=0.8)
    ax2.plot(iterations, hbF, alpha=0.8)
    ax3.plot(stepsize_iters, hbA, alpha=0.8)

    ax1.plot(iterations, adamX, alpha=0.8)
    ax2.plot(iterations, adamF, alpha=0.8)
    ax3.plot(stepsize_iters, adamA, alpha=0.8)

    ax3.set_xlabel('iterations')
    ax1.set_ylabel('x')
    ax2.set_ylabel('f(x)')
    ax3.set_ylabel('step size')
    fig.legend(["polyak", "RMSProp", "HeavyBall", "Adam"])
    #fig.suptitle("Effect of each iteration of gradient 
descent on x and f(x)")
    fig.suptitle("Optimization functions applied to {} 
function with starting point {}".format(fn.name, x[0]))
    plt.show()

    if x[0] == 100:
        # try Adam with larger alpha on x = 100
        X, F, A = basic_Adam(fn, x, alpha = 10)
        fig = plt.figure()
        iterations = range(51)
        stepsize_iters = range(50)
        ax1 = fig.add_subplot(311)
        ax2 = fig.add_subplot(312)
        ax3 = fig.add_subplot(313)

        ax1.plot(iterations, X, alpha=0.8)
        ax2.plot(iterations, F, alpha=0.8)
        ax3.plot(stepsize_iters, A, alpha=0.8)

        ax3.set_xlabel('iterations')
        ax1.set_ylabel('x')
        ax2.set_ylabel('f(x)')
        ax3.set_ylabel('step size')
        fig.suptitle("Adam with alpha = 10 applied to {}
function with starting point {}".format(fn.name, x[0]))
        plt.show()
```

```
def plot_optimization_comparison(fn, algorithm_name, F, A1,
A2, theta1s, theta2s, alpha, beta, beta_two = 0):
    # Plot f(x) vs iteration
    iterations = range(51)
    fig = plt.figure()
    plt.plot(iterations, F.T, alpha=0.8)
    fig.supxlabel("iterations")
    fig.supylabel("f(x)")
    plt.legend(alpha)
    if algorithm_name == "Adam":
        fig.suptitle("Effect of different alpha values on
function value with beta one = {} and beta two
{}".format(beta, beta_two))
    else:
        fig.suptitle("Effect of different alpha values on
function value with beta = {}".format(beta))
    plt.show()
```

```
    # Plot step size vs iteration
    # Step size 1
    iterations = range(50)
    fig = plt.figure()
    plt.plot(iterations, A1.T, alpha=0.8)
    fig.supxlabel("iterations")
    fig.supylabel("step size 1")
    plt.legend(alpha)
    if algorithm_name == "Adam":
        fig.suptitle("Effect of different alpha values on step
size with beta one = {} and beta two {}".format(beta,
beta_two))
    else:
        fig.suptitle("Effect of different alpha values on step
size with beta = {}".format(beta))
    plt.show()
```

```
    # Step size 2
    fig = plt.figure()
    plt.plot(iterations, A2.T, alpha=0.8)
    fig.supxlabel("iterations")
    fig.supylabel("step size 2")
    plt.legend(alpha)
    if algorithm_name == "Adam":
        fig.suptitle("Effect of different alpha values on x2
step size with beta one = {} and beta two = {}".format(beta,
beta_two))
    else:
        fig.suptitle("Effect of different alpha values on x2
step size with beta = {}".format(beta))
    plt.show()
```

```
    # Plot contour plot
    buffer = 1
    xMin = np.amin(theta1s) - buffer
    xMax = np.amax(theta1s) + buffer
    yMin = np.amin(theta2s) - buffer
    yMax = np.amax(theta2s) + buffer
    x = np.arange(xMin, xMax, 0.1)
    y = np.arange(yMin, yMax, 0.1)
    X, Y = np.meshgrid(x, y)
    Z = fn.fx(X, Y)

    plt.contour(X, Y, Z)
    plt.xlabel("theta 1"); plt.ylabel("theta 2")
    plt.plot(theta1s.T, theta2s.T, alpha=0.8)
    plt.legend(alpha)
    if algorithm_name == "Adam":
        plt.suptitle("Contour plot with beta one = {} and beta
two = {}".format(beta, beta_two))
    else:
        plt.suptitle("Contour plot with beta = 
{}".format(beta))
    plt.show()

def optimization_hyperparameter_comparison(fn, theta0,
algorithm_name, alphas, beta, beta_two = [], default_alpha =
0.05, default_beta_one = 0.9, default_beta_two = 0.25,
plot_contour = False):
    if algorithm_name == "RMSProp":
        alphaAl = lambda f, x, y : RMSProp(f, x, y)
    elif algorithm_name == "HeavyBall":
        alphaAl = lambda f, x, y : HeavyBall(f, x, y)
    else:
        alphaAl = lambda f, x, y : Adam(f, x, y,
default_beta_two)

    for b in beta:
        theta1s = np.array([])
        theta2s = np.array([])
        function_values = np.array([])
        theta1_stepsizes = np.array([])
        theta2_stepsizes = np.array([])
        for a in alphas:
            alphaAlgorithm = alphaAl(fn, a, b)
            (THETA1, THETA2, F, A1, A2) = gradientDescent(fn,
theta0, alphaAlgorithm)
            if function_values.size == 0:
                theta1s = np.array([THETA1])
                theta2s = np.array([THETA2])
                function_values = np.array([F])
                theta1_stepsizes = np.array([A1])
                theta2_stepsizes = np.array([A2])
            else:
                theta1s = np.vstack([theta1s, THETA1])
                theta2s = np.vstack([theta2s, THETA2])
                function_values = np.vstack([function_values,
F])
```

```
                theta1_stepsizes =
np.vstack([theta1_stepsizes, A1])
                theta2_stepsizes =
np.vstack([theta2_stepsizes, A2])
        if algorithm_name == "Adam":
            plot_optimization_comparison(fn, algorithm_name,
function_values, theta1_stepsizes, theta2_stepsizes, theta1s,
theta2s, alphas, b, default_beta_two)
        else:
            plot_optimization_comparison(fn, algorithm_name,
function_values, theta1_stepsizes, theta2_stepsizes, theta1s,
theta2s, alphas, b)

    if algorithm_name == "Adam":
        alphaAl = lambda f, x, y : Adam(f, x,
default_beta_one, y)
        for b in beta_two:
            del function_values
            function_values = np.array([])
            theta1_stepsizes = np.array([])
            theta2_stepsizes = np.array([])
            for a in alphas:
                alphaAlgorithm = alphaAl(fn, a, b)
                (_, _, F, A1, A2) = gradientDescent(fn,
theta0, alphaAlgorithm)
                if function_values.size == 0:
                    function_values = np.array([F])
                    theta1_stepsizes = np.array([A1])
                    theta2_stepsizes = np.array([A2])
                else:
                    function_values =
np.vstack([function_values, F])
                    theta1_stepsizes =
np.vstack([theta1_stepsizes, A1])
                    theta2_stepsizes =
np.vstack([theta2_stepsizes, A2])

            plot_optimization_comparison(fn, algorithm_name,
function_values, theta1_stepsizes, theta2_stepsizes, theta1s,
theta2s, alphas, default_beta_one, b)

def Part_B():
    # Function One
    fn = function_one()
    x = [9, 4]
    optimization_hyperparameter_comparison(fn, x, "RMSProp",
alphas = [0.01, 0.05, 0.1], beta = [0.25, 0.9])
    optimization_hyperparameter_comparison(fn, x, "HeavyBall",
alphas=[0.01, 0.025, 0.05], beta=[0.25, 0.9])
    optimization_hyperparameter_comparison(fn, x, "Adam",
alphas=[0.01, 0.1, 1, 2], beta=[0.25, 0.9], beta_two=[0.25,
0.999])

    # Function Two
    fn = function_two()
    x = [10, 5]
    optimization_hyperparameter_comparison(fn, x, "RMSProp", 
alphas = [0.01, 0.05, 0.1, 0.12], beta = [0.25, 0.9], 
plot_contour=True)
    optimization_hyperparameter_comparison(fn, x, "HeavyBall", 
alphas=[0.01, 0.025, 0.05], beta=[0.25, 0.9], 
plot_contour=True)
    optimization_hyperparameter_comparison(fn, x, "Adam", 
alphas=[0.01, 0.1, 1, 2], beta=[0.25, 0.9], beta_two=[0.25, 
0.999], plot_contour=True)

def Part_C():
    fn = ReLu_function()

    x = np.array([-1])
    plot_basic_implementations(fn, x, 0)

    x = np.array([1])
    plot_basic_implementations(fn, x, 0)

    x = np.array([100])
    plot_basic_implementations(fn, x, 0)

vars()
fn = function()
x = np.array([1])
plot_basic_implementations(fn, x, 0)
Part_B()
Part_C()
```