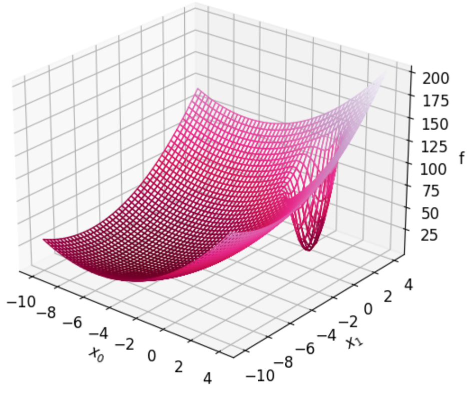
Optimisation Algorithms Week 6 Assignment

(a) (i) Algorithm 1 shows the basic steps of the minibatch SGD. The algorithm loops through the number of allowed/required iterations. For each iteration, it shuffles T and then splits it into subsets/mini-batches. The gradient is computed in the current point, x_k, using $\delta = 10^{-5}$, by taking each mini-batch and performing a finite difference approximation. The code for this is shown in algorithm 2. The step is then computed depending on the step option chosen (default, Polyak, RMSProp, HeavyBall or Adam), and x is updated. If the change in function value between x_k and x_{k-1} is less than 10^{-5}, the iteration loop is broken and the steps taken are returned to be plotted.

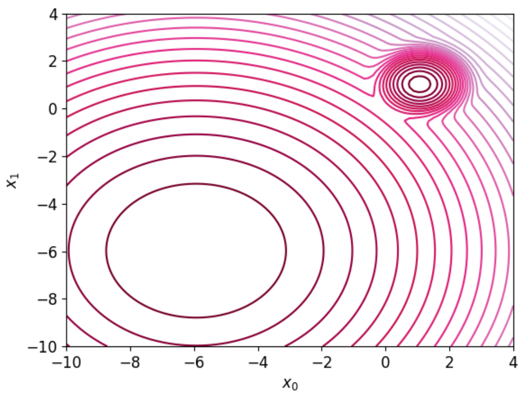
Algorithm 1 Minibatch Stochastic Gradient Descent

```
$n \leftarrow size(T)$
**for** $k \in \overline{0, iterations}$ **do**
    $np.random.shuffle(T)$
    **for** $i \in range(0, n, batchSize$ **do**
        $sample \leftarrow T[i : i + batchSize]$
        $gradient = finiteDifference(x, N)$
        $step \leftarrow$ gradient-based step update
        $x_{k+1} \leftarrow x_k - step$
    **end for**
**end for**
```

(ii) Figure 1 shows the 3D plot and contour plot of $f(x, T)$ where T is the full set of 25 training values. $X = [-10, 4] * [-10, 4]$ was used in order to show the two minima of the function. The function has its global minimum - in X = [-6,-6] and a local minimum in X = [1,1].



(a) 3D plot



(b) Contour plot

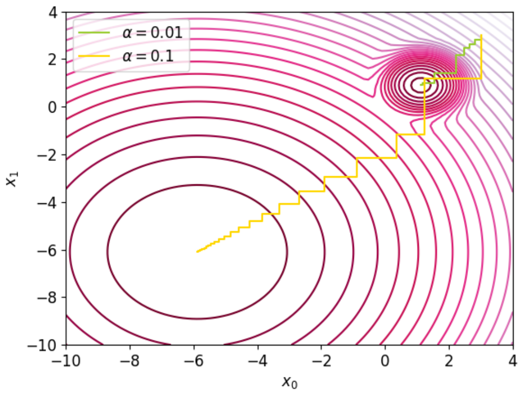
Figure 1: Function plots

(iii) The gradients were computed using a central difference estimation. The formula used was $gradient_k = \dfrac{f(x_k + \delta, sample) - f(x_k - \delta, sample)}{2\delta}$. When $\delta \to 0$, this formula describes the derivative of the function. For a very small delta, it can offer an approximation. The code for the function finiteDifference() referenced in algorithm 1 is given in algorithm 2.

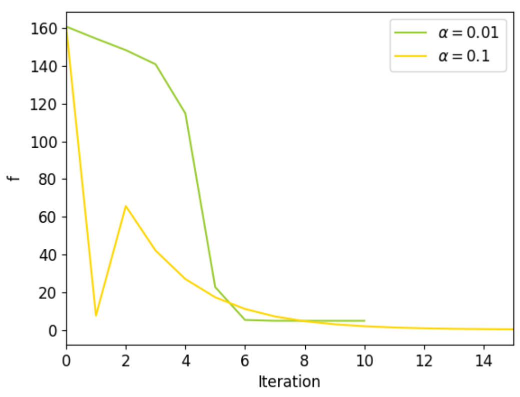
Algorithm 2 Finite Difference

```
$\delta \leftarrow 10^{-5}$
$g_0 \leftarrow (f([x_0 + \delta, x_1], N) - f([x_0 - \delta, x_1], N))/(2 * \delta)$
$g_1 \leftarrow (f([x_0, x_1 + \delta], N) - f([x_0, x_1 - \delta], N))/(2 * \delta)$
**return** $[g_0, g_1]$
```

(b) (i) Figure 2 (a) shows the steps taken by a GD with a constant step-size to minimise the loss function. The starting point is $x_0 = [3, 3]$, and two values for alpha are included to show two versions of neatly achieving optimality. The green line with $\alpha = 0.01$ is an example of gradient descent finding the [1, 1] local minimum. The yellow line with $\alpha = 0.1$ shows how even though the algorithm enters deep into the valley forming around [1, 1], the step size is too large, causing it to bounce out and instead continue towards the [-6, -6] global minimum. Figure 2 (b) shows the values of f for the first 5 epochs.



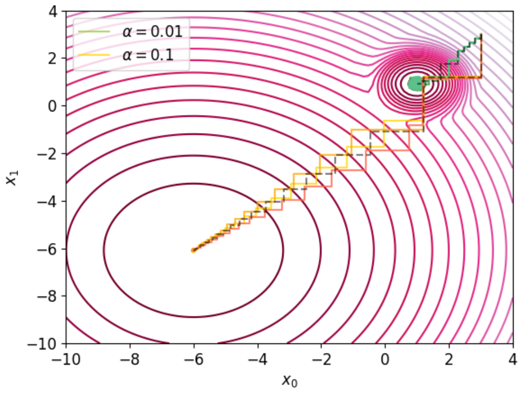
(a) Steps and contour plot



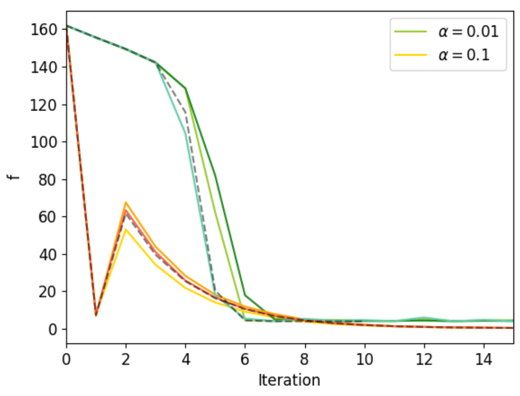
(b) Function values

Figure 2: Gradient descent

(ii) Figure 3 (a) shows the steps taken by a minibatch SGD with a mini-batch size of 5. The same starting point is chosen as in (b) (i), and the same two values for alpha are compared. To compare the two approaches, the black lines on each plot represent the same lines shown in (b)(i) generated using batch GD. The behaviour of the minibatch SGD and batch GD is similar for each value of alpha respectively. The SGD algorithm is run 3 times to see how the behaviour changes (shown in different hues of the same family). The small deviations from the GD black lines are caused by the random shuffling and sampling of T when computing the gradient. Because a batch gradient descent considers the entire training set on each iteration, there will be no change in behaviour for multiple runs, even when shuffling T. However, since only a fraction of the set is sampled in a minibatch SGD, the gradient in the same point is slightly different every time.



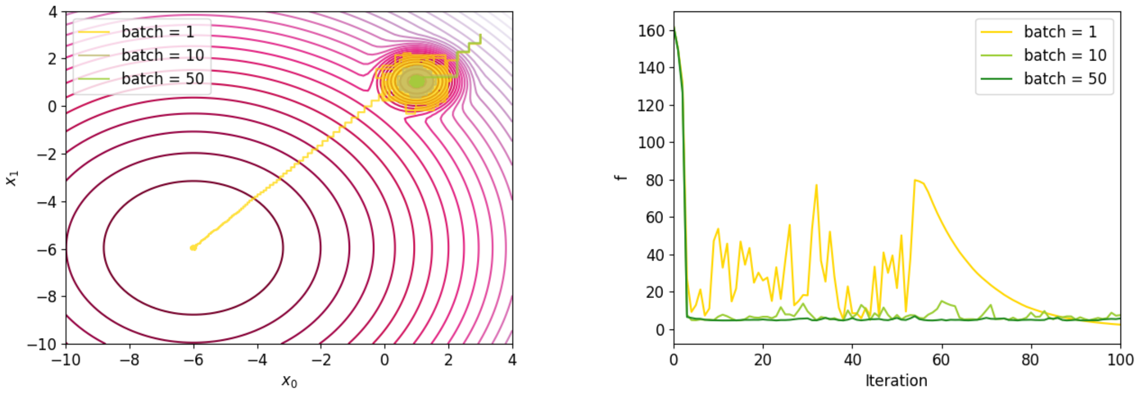
(a) Steps and contour plot



(b) Function values

Figure 3: Minibatch SGD

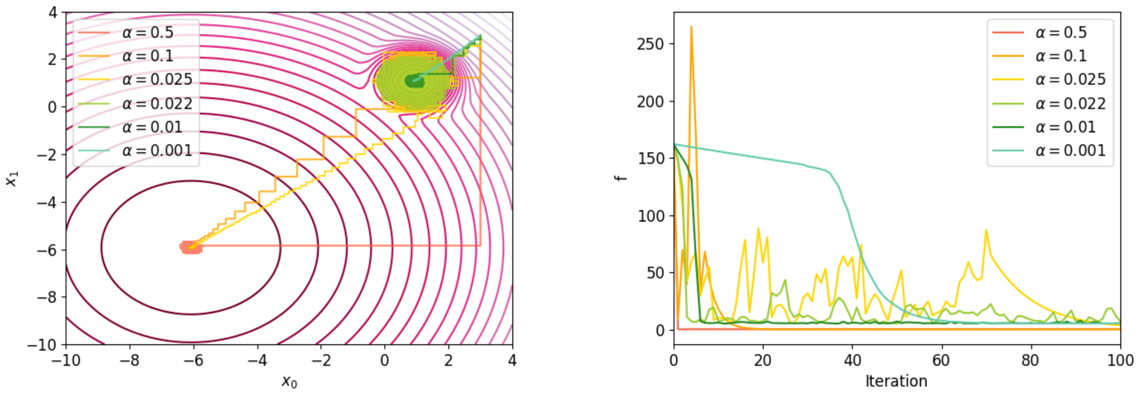
(iii) Figure 4 shows the results of varying the batch size of the minibatch SGD. The parameters of this experiment are different from previous plots, in order to best show the effects of different batch sizes. The size of the training set T is 100 this time, and the algorithm is run for a single value of alpha equal to 0.02. The batch sizes observed are 1, 10, and 50. The plots show how smaller batch sizes lead to more noise. The very small step size would normally cause SGD to get trapped in the local minimum - which it does, for the minibatches equal to 1/2 or even 1/10 of the training set size. However, it can be seen that when noise is large enough, such as the amount caused by batch size = 1, SGD escapes the local minimum and is able to converge towards the global minimum instead. On figure (b) this can be seen around iteration 50 (which in this case is also epoch 50) as the yellow line starts to converge towards the global minimum which end up being slightly smaller than the local minimum that the green lines are stuck in.



(a) Steps and contour plot (b) Function values

Figure 4: Batch size experiments

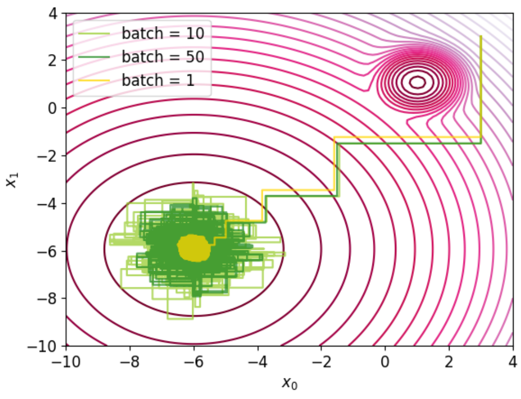
(iv) Figure 5 shows the results of varying the step size of the minibatch SGD. This time, the training value set is back to its original size of 25, and the minibatch is set to 5. The plots show how the step size alpha also influences the amount of noise, similar to the batch size at (b) (iii). Specifically, the larger the step size, the more noise there is. In this example, somewhere between alpha = 0.2 and 0.3, the step size become large enough to introduce enough noise to escape the local minimum and converge to the global minimum. The turquoise heads straight for the local minimum in very small steps. The dark green also ends up in the local minimum although it takes larger steps. The light green is more noisy but it never manages to escape the local minimum (although the same step size did find the global minimum in some of the runs). By slightly increasing the step size, the yellow line also circles around the local minimum but eventually exists the valley around iteration 70/epoch 14, and converges to the global minimum. The golden line passes through the valley but its step size is too small to get caught in it even for a few epochs. The orange line has such a large step size that it jumps directly past the local valley.



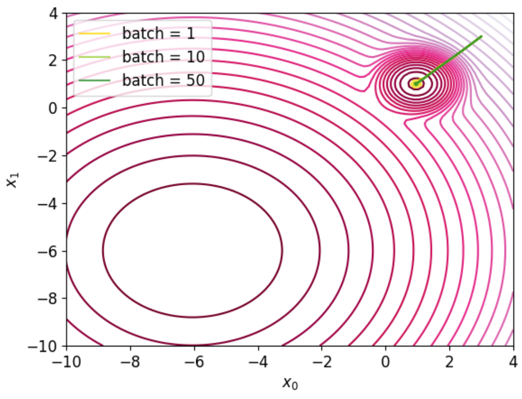
(a) Steps and contour plot (b) Function values

Figure 5: Step size experiments

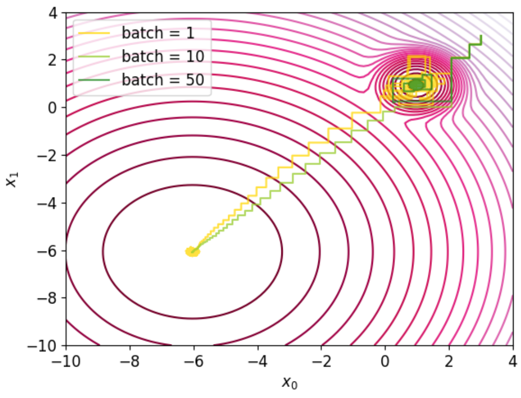
(c) The experiment is once again run with training set T of size 100, on batch sizes 1, 10, and 50. Figure 6 shows the steps taken by SGD with these parameters for the four step updates: Polyak, RMSProp, Heavyball, and Adam. The $beta_1$ parameter of the HeavyBall and Adam momentum is 0.6. The $beta_2$ parameter used by RMSProp and Adam is 0.1. The baseline SGD with the default steps for the same parameters is shown in figure 4.



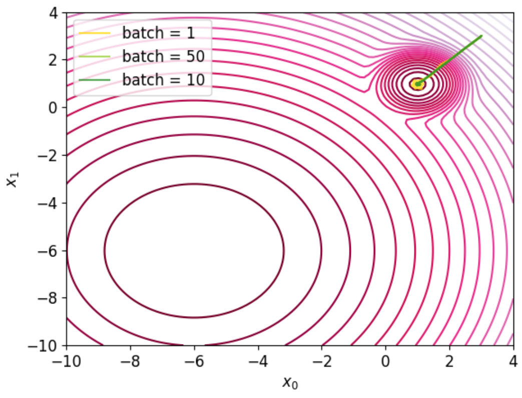
(a) Polyak



(b) RMSProp values



(c) HeavyBall

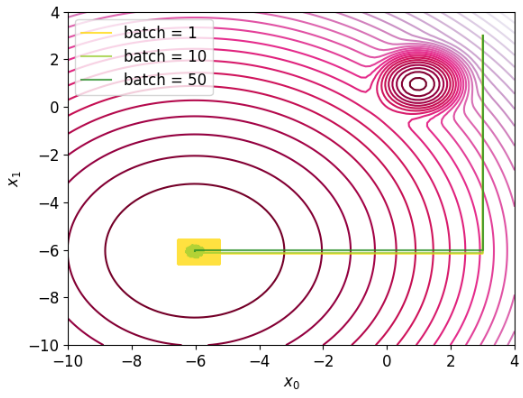


(d) Adam

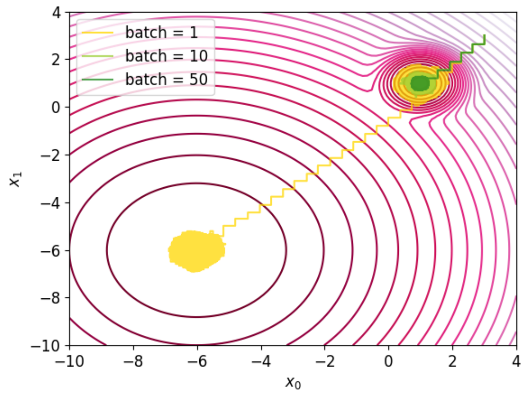
Figure 6: Step update experiments

(i) Polyak modifies the baseline algorithm by updating alpha on each iteration to account for the function steepness. The step size around the flat area, where the gradient is very low, becomes very large and the line becomes very noisy. This effect is slightly larger for the medium batch size than the large batch size. One explanation is that the larger batch size causes less noise, but another likely possibility is that the larger batch size causes less steps to be taken in total. Setting the batch size to 1 results in surprisingly less noise.

(ii) RMSProp keeps a sum of fractions of previous gradients and uses it to adjust the step size. This is done to remove noise and oscillations, and in this case results in the algorithm going for the local minimum regardless of the batch size. Figure 7 shows the same experiment but with an initial alpha of 0.5. The larger step size, combined with the noise caused by the small step size cause RMSProp to exit the local minimum valley and find the global minimum. Comparing this to the baseline algorithm with alpha = 0.5 shows that this step size is huge, allowing any batch size, even the ones that previously got stuck in the local minimum for alpha = 0.02, to jump straight to the global minimum.



(a) Baseline

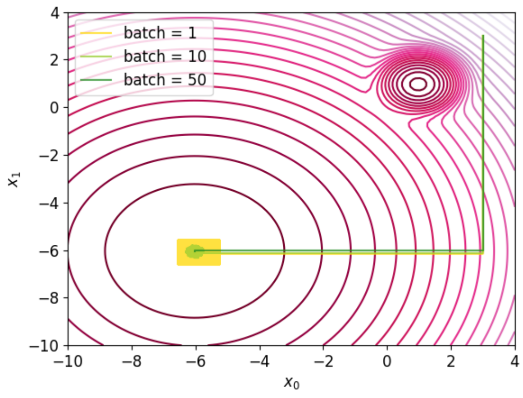


(b) RMSProp values

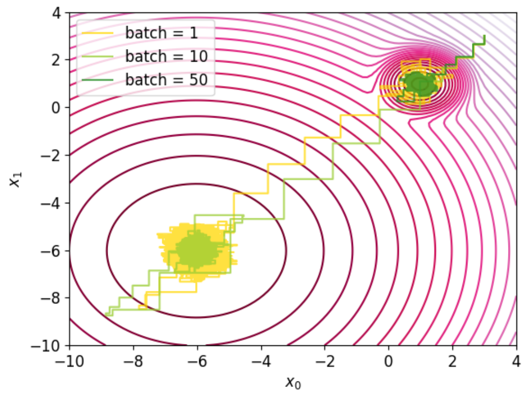
Figure 7: Step update experiments with alpha = 0.5

(iii) HeavyBall builds momentum by adding a fraction $beta_1$ of the previous step to the current step. This causes even the low-noise batch 50 run to roll around the local minimum before converging. The small and medium batch runs both jump out of the local minimum due to extra noise and converge to the global minimum

(iv) Adam, much like RMSProp, removes noise too well to be able to escape the local minimum for a small step size when the parameter $beta_2$ is any larger than 0. Figure 8 shows the behaviour of Adam for a larger step size of 0.5, along with the baseline algorithm with the same step size. This time, both the small and medium sized batches introduce enough noise to escape the local minimum. The momentum built as they head for the global minimum causes them to overshoot, but they are then able to roll back to it.



(a) Baseline



(b) Adam values

Figure 8: Step update experiments with alpha = 0.5

Appendix A - Code

```
import numpy as np
from math import *
import matplotlib.pyplot as plt
from matplotlib import cm

def generate_trainingdata(m=25):
    return np.array([0,0])+0.25*np.random.randn(m,2)

def f(x, minibatch):
    # loss function sum_{w in training data} f(x,w)
    y=0; count=0
    for w in minibatch:
        z=x-w-1
        y=y+min(42*(z[0]**2+z[1]**2), (z[0]+7)**2+(z[1]+7)**2)
        count=count+1
    return y/count

# Estimating the derivative of f(x,N)
def finiteDifference(x, N, delta = 10 ** (-5)):
    g0 = (f([x[0] + delta,x[1]], N) - f([x[0] - delta,x[1]], N))/(2 * delta)
    g1 = (f([x[0], x[1] + delta], N) - f([x[0], x[1] - delta], N)) / (2 * delta)
    return np.array([g0,g1])
# Computing the values to plot
def makeZ(X, Y, T):
    Z=np.zeros_like(X)
    for i in range(len(X)):
        for j in range(len(X[0])):
            Z[i, j] = f([X[i, j], Y[i, j]], T)
    return Z

# Plotting the wireframe (using a surface plot as it allows colourmaps)
def plot3D(X, Y, Z):
    plt.rcParams['font.size'] = 12
    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()

# Creating the contour plot; Generating and plotting the steps
def plotContour(X, Y, Z,colours,lines=1,alphaRange=[0.01,0.1],batchRange=[5],update='None'):
    plt.rcParams['font.size'] = 12
    fig,ax=plt.subplots(1,1)
    ax.set(xlim=(-10, 4), ylim=(-10,4))
    ax.contour(X, Y, Z, levels=30, cmap='PuRd_r')
    for i in range(lines):
        for a, alpha in enumerate(alphaRange):
            for b, batch in enumerate(batchRange):
                steps = SGDminibatch(x0=[3,3], T=T, alpha0 = alpha, batchSize = batch,
                update=update)
                ax.step(steps[0], steps[1], alpha=0.75, color = colours[a][b],
                label = 'batch = '+str(batch))
    # Uncomment to show the GD behaviour
    '''for a, alpha in enumerate(alphaRange):
        steps = SGDminibatch(x0=[3, 3], T=T, alpha0=alpha, batchSize=25, update='None')
        ax.step(steps[0], steps[1], alpha=0.5, linestyle='--',color='black')'''
    #ax.legend(labels = [r'$\alpha = $'+str(0.01),r'$\alpha = $'+str(0.1)]) (b i,ii)
    ax.legend()
    ax.set_xlabel('$x_{0}$')
    ax.set_ylabel('$x_{1}$')
    plt.show()

def plotLine(T,colours,lines=1,alphaRange=[0.01,0.1],batchRange=[5],update='None'):
    plt.rcParams['font.size'] = 12
    fig,ax=plt.subplots(1,1)
    ax.set(xlim=(0, 100),ylim=(0,300))
    for i in range(lines):
        for a, alpha in enumerate(alphaRange):
            for b, batch in enumerate(batchRange):
                steps = SGDminibatch(x0=[3, 3], T=T, alpha0=alpha, batchSize =batch,
                update=update)
                steps = zip(steps[0], steps[1])
                Z = [f(s, T) for s in steps]
                ax.plot(Z,c = colours[a][b],
                label = 'batch = '+str(batch))
    #Uncomment to show the GD behaviour
    '''for a, alpha in enumerate(alphaRange):
        steps = SGDminibatch(x0=[3, 3], T=T, alpha0=alpha, batchSize=25, update='None')
        steps = zip(steps[0], steps[1])
        Z = [f(s, T) for s in steps]
        ax.plot(Z, c='black',alpha =0.5,linestyle='--',label = 'GD')'''
    #ax.legend(labels = [r'$\alpha = $'+str(0.01),r'$\alpha = $'+str(0.1)]) #(b i,ii)
    ax.legend() #b iii
    ax.set_xlabel('Iteration')
    ax.set_ylabel('f')
    plt.show()

# SGD minibatch finds the minimum and returns the steps taken
def SGDminibatch(x0, T, alpha0=0.01, beta1 = 0.6, beta2=0.1, epsilon = 10**-3,iters=10000,
batchSize=5, update = 'None'):
    x = np.array(x0)
    X0, X1 = [x[0]], [x[1]]
    n = len(T)
    alpha = alpha0
    sum, step, m, v = [0, 0, 0, 0]
    for k in range(iters):
        np.random.shuffle(T)
        for i in np.arange(0, n, batchSize):
            sample = T[i:i + batchSize]
            gradient = finiteDifference(x, sample)
            if update == 'Adam':
                m = beta1 * m + (1 - beta1) * np.array(gradient) / (1 - beta1 ** (k + 1))  # Adam
                v = beta2 * v + (1 - beta2) * np.dot(np.array(gradient), np.array(gradient)) / (
                            1 - beta2 ** (k + 1))  # Adam
                step = alpha * m / (sqrt(v) + epsilon)  # Adam
            elif update == 'HeavyBall':
                step = beta1 * step + alpha * gradient  # HeavyBall
            else:
                step = alpha * gradient  # Default
            if update == 'Polyak':
                alpha = f(x, sample) / (epsilon+np.dot(gradient, gradient))  # Polyak step
                step = alpha * gradient
            elif update == 'RMSProp':
                sum = beta2 * sum + (1 - beta2) * np.dot(gradient, gradient) # RMSProp
                alpha = alpha0 / sqrt(sum + epsilon)  # RMSProp
                step = alpha * gradient
            fPrev = f(x, T)
            x = x - step
            print(gradient, np.dot(gradient,gradient), x, fPrev)
            X0.append(x[0])
            X1.append(x[1])
        if abs(fPrev - f(x,T)) < 10**(-5):
            break
    return [X0, X1]


T = generate_trainingdata(m=25)
x = np.linspace(-10, 4, 100)
y = np.linspace(-10, 4, 100)
X, Y = np.meshgrid(x, y)
Z = makeZ(X, Y, T)


colours = [['yellowgreen', 'forestgreen', 'mediumaquamarine'], ['gold', 'orange', 'tomato']]
#colours = [['gold', 'yellowgreen','forestgreen',]]
#colours = [['tomato'],['orange'],['gold'],['yellowgreen'],['forestgreen'],['mediumaquamarine']]
plot3D(X,Y,Z)
plotContour(X,Y,Z,colours,lines=1,alphaRange=[0.1],batchRange=[5])
plotLine(T,colours,lines=1,alphaRange=[0.1],batchRange=[5])
```