CS7DS2: Week 6 Assignment

Conor McCauley - 17323203

March 16, 2022

N.B. The loss function used in this assignment is included in Appendix A.

Question (a)

(i) A single Python class, `SGD`, will implement the SGD algorithm for all methods of step size calculation. The constructor takes as parameters the function being minimised, the derivative functions, the starting value of x, an enum specifying the choice of step size, a dictionary of hyperparameters, the batch size and the training data.

The `run_iter_minibatch()` method runs an iteration of mini-batch SGD:

```
def run_iter_minibatch(self):
    np.random.shuffle(self.T)
    N = len(self.T)
    for i in range(0, N, self.batch_size):
        if i + self.batch_size > N: continue
        sample = self.T[i:(i + self.batch_size)]
        self.iter_function(sample)
```

The method shuffles the training data, iterates over batch size-length samples of the training data (making sure not to sample outside the bounds of the array) and calls the `iter_function()` method for that sample. This method varies depending on the choice of step size calculation but it is practically identical to the implementations provided in the previous assignment. For example, when using a constant step size `iter_function()` would call the following method:

```
def __iter_constant(self, sample):
    alpha = self.params['alpha']
    for i in range(self.n):
        self.x[i] -= alpha * self.__calc_approx_derivative(i, sample)

def __calc_approx_derivative(self, i, sample):
    return sum(
        self.df[i](*self.x, *sample[j]) for j in range(self.batch_size)
    ) / self.batch_size
```

As can be seen in the above code, the only noticeable change from the previous assignment is the replacement of calls to `df` with calls to a function that uses the mini-batch sample to calculate the approximate derivative.

(ii) Figure 1 contains a contour plot and a wireframe plot of $f(x, T)$ over the ranges $-15 \leq x_0 \leq 10, -15 \leq x_1 \leq 10$. These ranges were chosen so as to allow for a clear view of both of the function's minima while also showing its flatness when $x_1 < 0$ and steepness when $x_1 > 5$. The same training data, T, was used in both plots to maintain consistency:

(iii) The part of our function that we are interested in finding the derivative of is:

$$\min(18(z_0^2 + z_1^2), (z_0 + 10)^2 + (z_1 + 5)^2),\ z_i = x_i - w_i - 1, w \in T$$

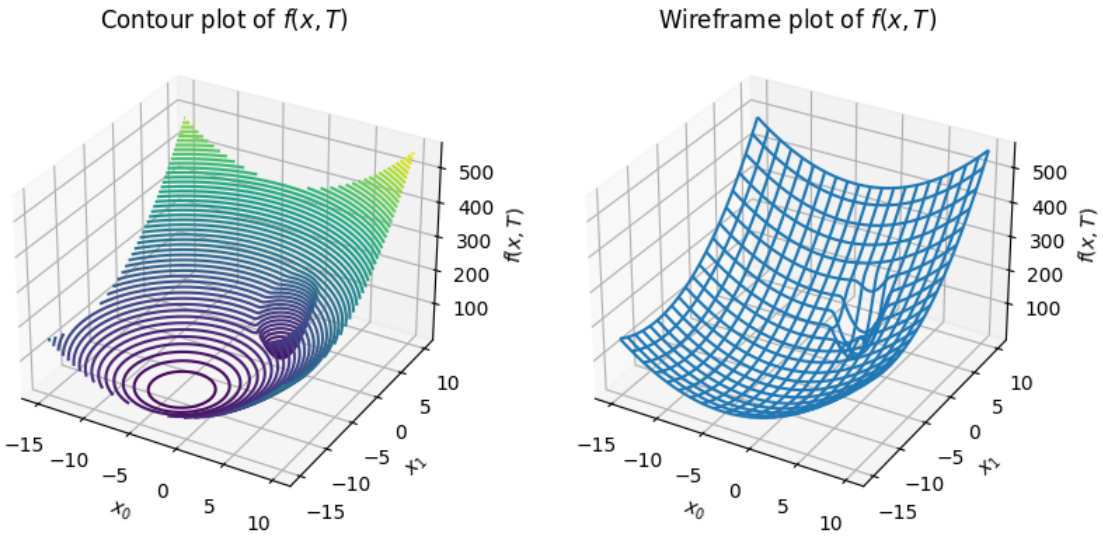

Figure 1: Contour and wireframe plots of $f(x, T)$

As such, we can write our function in SymPy using four symbols, x_0, x_1, w_0, w_1, and differentiate it once with respect to x_0 and once with respect to x_1:

```
x0, x1, w0, w1 = sp.symbols('x0 x1 w0 w1', real=True)
f = sp.Min(18*(((x0-w0-1)**2)+((x1-w1-1)**2)),(((x0-w0-1)+10)**2)+(((x1-w1-1)+5)**2))
df0 = sp.diff(f, x0)
df1 = sp.diff(f, x1)
```

This code is almost identical to the SymPy code used in both previous assignments except for the inclusion of a sample of training data, w. The resulting derivatives are too long to include here but can be found in Appendix A.

```
print(df0) -> (-36*w0 + 36*x0 - 36)*Heaviside(-18*(-w0 + x0 - 1)**2 + ...
print(df1) -> (-36*w1 + 36*x1 - 36)*Heaviside(-18*(-w0 + x0 - 1)**2 + ...
```

Question (b)

(i) Since running classical gradient descent (CGD) on our training data is the same as running SGD with a batch size equal to the length of our training data we can use the `SGD` class outlined in (a)(i) with the constant step size algorithm. In order to determine an appropriate step size we will experiment with a range of $\alpha \in \{0.1, 0.01, 0.001, 0.0001\}$. Running gradient descent for 100 iterations for each α produces the results shown in figure 2.

It is evident from these results that $\alpha = 0.1$ is optimal as it converges on a minimum the fastest with $\alpha = 0.01$ and $\alpha = 0.001$ not converging on the global minimum and $\alpha = 0.0001$ not converging at all. The reason for this lack of convergence on the global minimum is clear from the contour plot. The noise introduced to the training data causes the minimum that all but $\alpha = 0.1$ converge to be larger than zero (the function's noiseless minimum). The minimum that $\alpha = 0.1$ converges to is not exactly zero either however it is a good deal closer than the other minimum. We can thus conclude that $\alpha = 0.1$ is the optimal choice for our function.

(ii) Using a value of $\alpha = 0.1$ we can run five trials of our constant step size SGD each running for 10 iterations using the same set of training data. The results of each trial can be seen in figure 3.

From these results we can see that the randomness introduced during each iteration of SGD, i.e. the shuffling of the training data, has a very minor effect on the results from each of the trials. All trials converged on the minimum after around six iterations. When compared to the

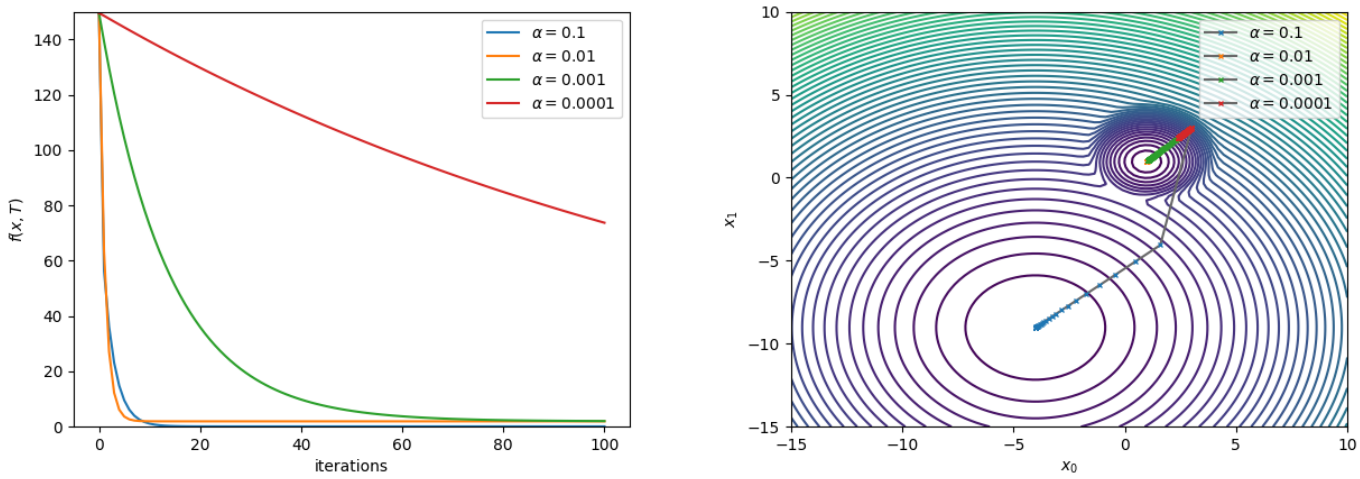

Figure 2: Change in x and $f(x, T)$ over iterations for different α (CGD + constant)

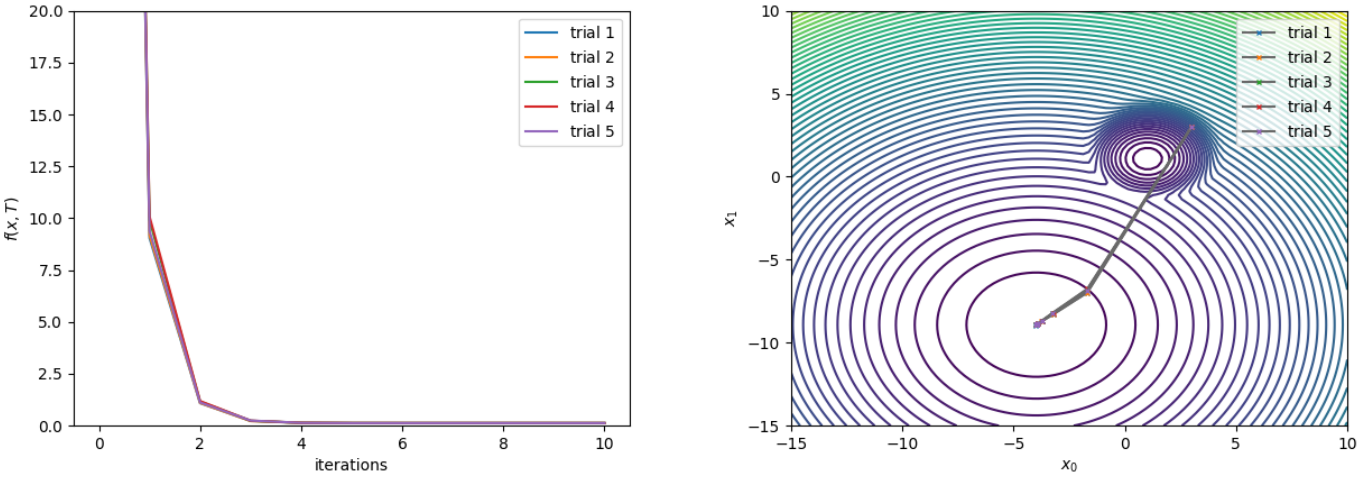

Figure 3: Change in x and $f(x, T)$ over iterations for $\alpha = 0.1$ (SGD + constant)

optimal results from (b)(i) the algorithm converged on the minimum in about half the number of iterations. Every trial also converged on the global minimum (given the noisy training data) as opposed to a local minimum.

(iii) We will use a range of batch sizes, n, from 1 to the size of our training data $|T|$. These values will be $n \in \{1, 3, 5, 10, 25\}$. Each trial will be run for 25 iterations. The results can be seen in figure 4.

All of the trials converge on the global minimum regardless of the batch size. This is probably due to α being large enough so as to allow the early iterations to expand x far enough away from the local minimum that x is initially closer to.

The number of iterations required to converge on the minimum seems to increase as n is increased. However, when n is very small ($n = 1$) the large number of changes to x that occur during each iteration, perhaps due to the noise occurring in each individual sample, seems to lead to fluctuations in $f(x, T)$ which causes poorer results. This effect occurs to a lesser degree when $n = 3$ and does not appear to occur at all when $n = 5$.

(iv) We will run each use the same range of values for α as we used in (b)(i) and run each trial for 25 iterations. The results can be seen in figure 5.

Only the trial where $\alpha = 0.1$ converges on the global minimum while the other trials all

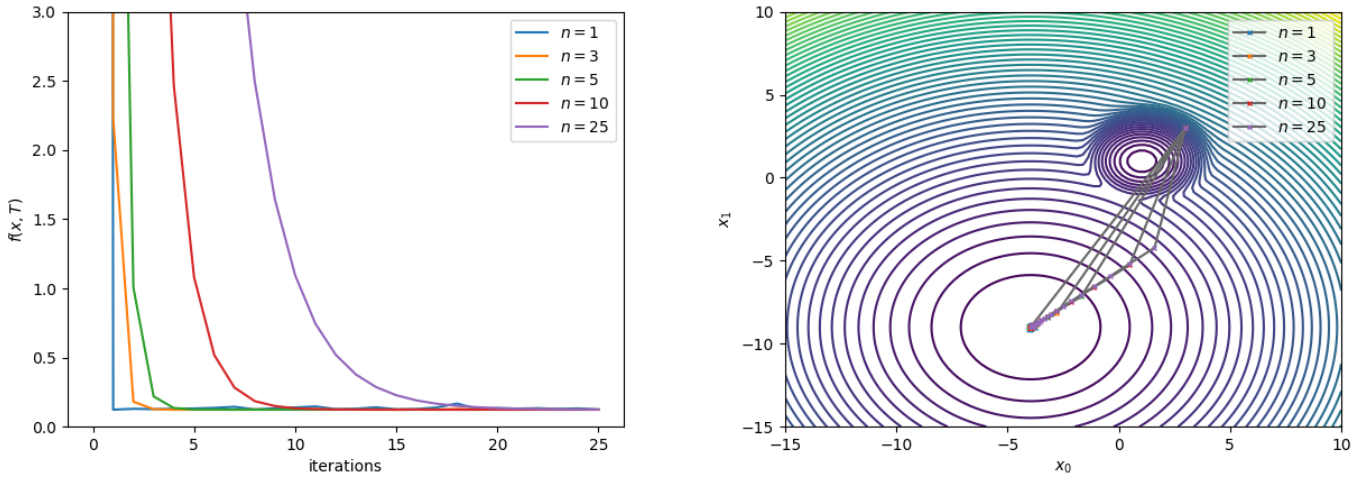

Figure 4: Change in x and $f(x, T)$ over iterations for different n and $\alpha = 0.1$ (SGD + constant)

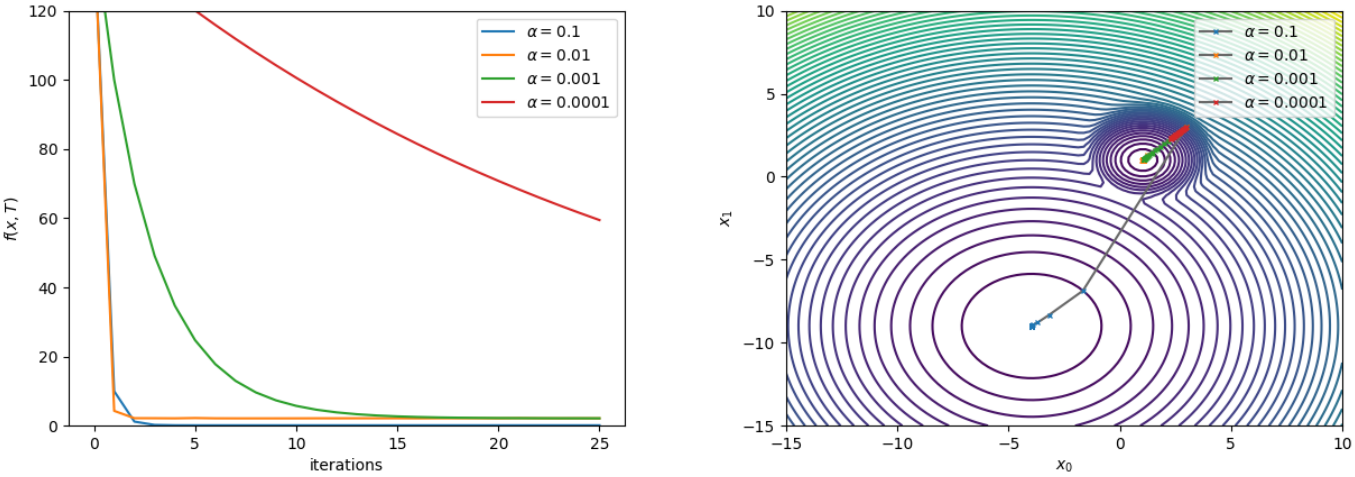

Figure 5: Change in x and $f(x, T)$ over iterations for different α (SGD + constant)

converge or begin to converge on the local minimum which is closer to the initial value of x. As discussed in (b)(iii), this is probably due to smaller values of α not expanding x far enough away from its initial value in early iterations.

Due to global minimum being much flatter and consistent than the local minimum which is much steeper and confined it might be expected that the more noise is introduced to the training data the less likely the algorithm is to converge on the limited local minimum and the more likely it is to converge on the much broader global minimum. However, perhaps due to there not being 'enough' noise in the training data all but one trial converges on the local minimum.

As was the case in previous trials, smaller values of α seem to take longer to converge. The optimal value of α still appears to be 0.1 because, although the quickest convergence occurs when $\alpha = 0.01$, it does not reach the global minimum.

Question (c)

(i) Polyak step size: No hyperparameters needed to be chosen for this choice of step size. SGD was run for 100 iterations for batch sizes $n \in \{1, 3, 5, 10, 25\}$ with the results from (b)(ii) being used as a baseline. The results can be seen in figure 6.

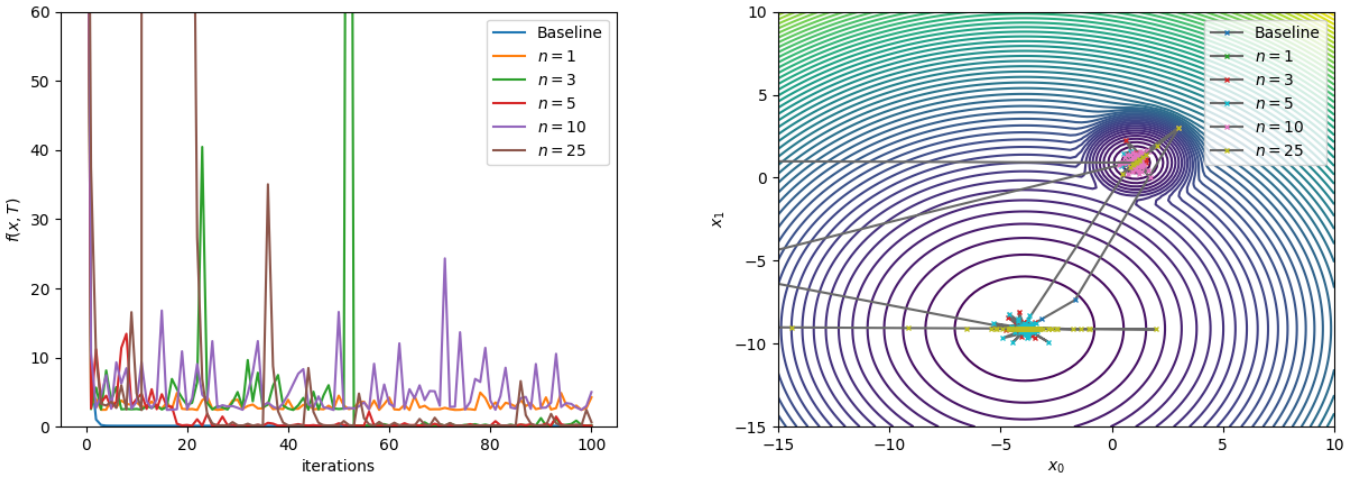

Figure 6: Change in x and $f(x, T)$ over iterations for different n (SGD + Polyak)

All of the trials appear to perform worse than the baseline. When $n = 1$ and $n = 10$ the functions converge on the local minimum but are relatively unstable. Each of the three cases that result in correct convergence remain somewhat unstable when compared to the baseline. A clear difference between the baseline result and the three aforementioned cases is that the latter all began converging on the local minimum before correcting themselves and converging correctly.

In this example, Polyak step size produces unstable results that do not always converge, or remain converged, on the global minimum. This is due to the step sizes during each iteration tending to be too large due to the squares of the approximate derivatives being much smaller than the value of the function itself which results in a larger than necessary step size in flatter regions of the function.

(ii) RMSProp: Hyperparameters were chosen in the ranges $\alpha_0 \in \{0.1, 0.01, 0.001\}$, $\beta \in \{0.25, 0.9\}$ and SGD was run for 100 iterations. The results, alongside the baseline, can be seen in figure 7.

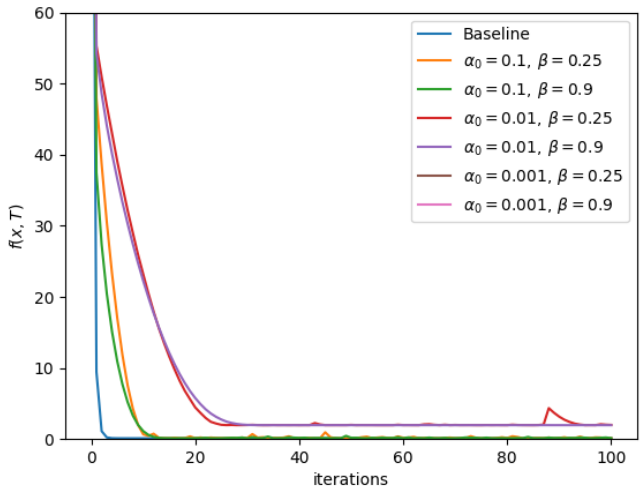

Figure 7: Change in $f(x, T)$ over iterations for different α_0, β (SGD + RMSProp)

Correct convergence occurs in the cases where $\alpha_0 = 0.1$ with the most stable results occurring when $\beta = 0.9$. As such, these values will be used when varying the batch size in figure 8.

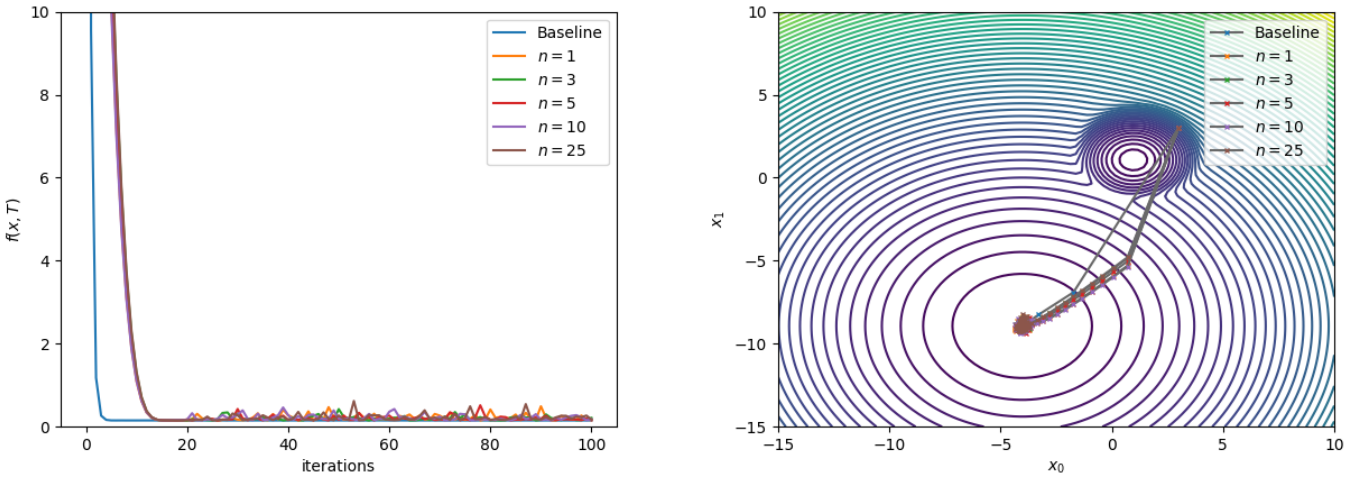

Figure 8: Change in x and $f(x,T)$ over iterations for different n and $\alpha_0 = 0.1, \beta = 0.9$ (SGD + RMSProp)

All of the trials appear to perform much better than when Polyak step size was used but they still fail to outperform the baseline. Convergence on the global minimum takes about three times longer than the baseline with every trial failing to converge on a stable value. The cases where $n = 3$ and $n = 5$ appear to be the most stable which is consistent with the results from (b)(iii).

Unlike Polyak, flatter regions of the function do not result in an increased step size. However, the step size does not completely stabilise as it is dependent on the moving average of square gradient sums which is constantly changing during iterations.

(iii) Heavy Ball: Hyperparameters were chosen in the ranges $\alpha \in \{0.01, 0.001\}$, $\beta \in \{0.25, 0.5, 0.9\}$ and SGD was run for 100 iterations. The results can be seen in figure 9.

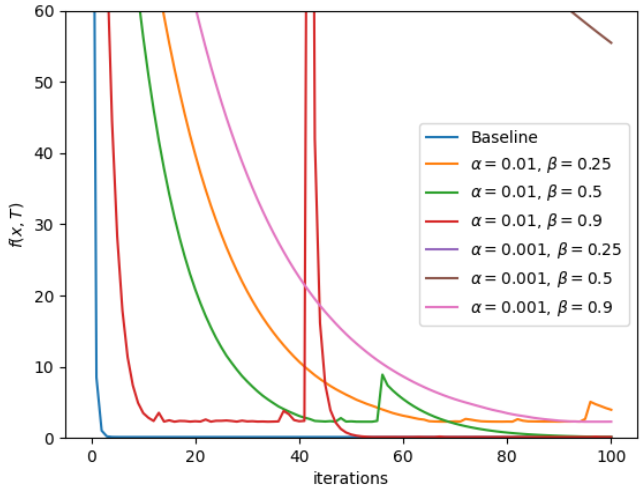

Figure 9: Change in $f(x,T)$ over iterations for different α, β (SGD + HB)

Correct convergence occurs when $\alpha = 0.01$ and $\beta \in \{0.5, 0.9\}$. Since convergence occurs the quickest when $\beta = 0.9$ these values will be used when varying the batch size in figure 10.

All of the trials initially begin to converge on the local minimum before eventually moving toward the global minimum. All but one of the cases appear to remain stable around the global minimum before briefly diverging away for a number of iterations before returning to

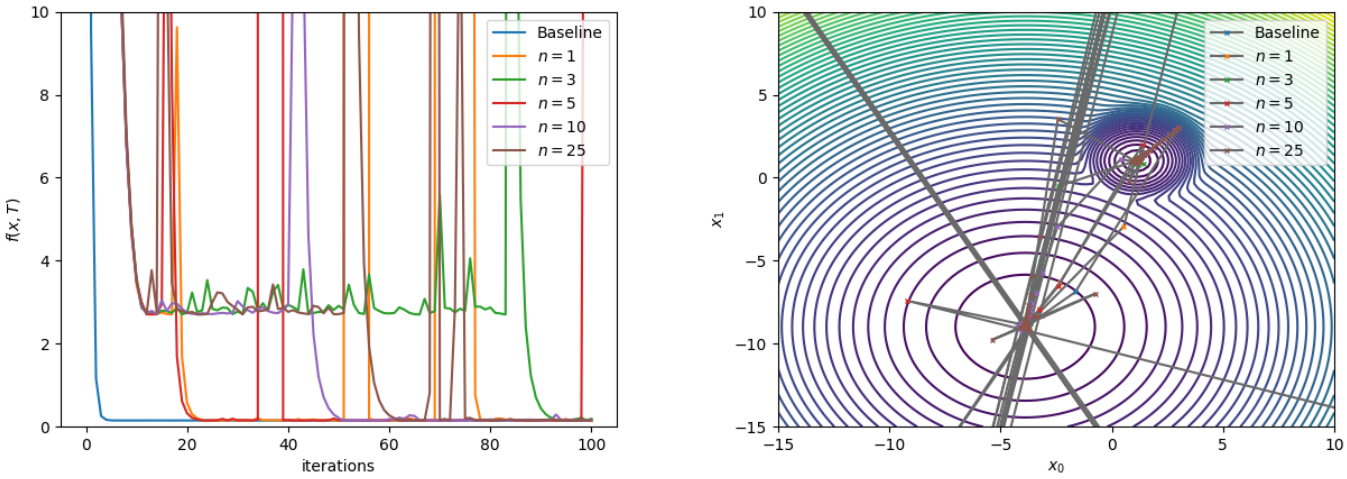

Figure 10: Change in x and $f(x, T)$ over iterations for different n and $\alpha_0 = 0.01, \beta = 0.9$ (SGD + HB)

the minimum. Only the case when $n = 10$ appears to be stable and it is not clear if this result is reliable. It is clear that none of these results are comparable to the baseline.

Like with Polyak, the instability is due to the step size in flatter regions of the function being too large and, although this effect is reduced by α, the repeated additions of $z\beta$ during each iteration can cause large fluctuations to occur.

(iv) Adam: Hyperparameters were chosen in the ranges $\alpha \in \{10, 1, 0.1\}$, $\beta_1 \in \{0.25, 0.9\}$ while β_2 was kept constant at 0.999. SGD was run for 100 iterations and the results can be seen in figure 11.

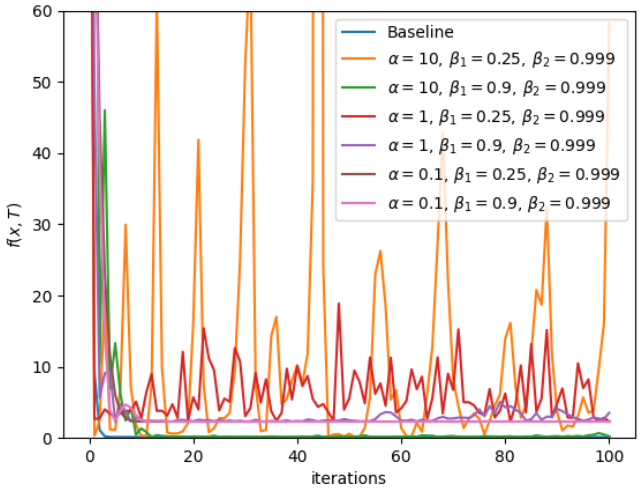

Figure 11: Change in $f(x, T)$ over iterations for different α, β_1, β_2 (SGD + Adam)

Smaller values of β_1 appear to lead to unstable results while smaller values of α fail to converge on the global minimum. The only trial that managed to converge on the global minimum while also remaining stable had values of $\alpha = 10$ and $\beta_1 = 0.9$. As such, these values will be used when varying the batch in figure 12.

All of the trials converge on the global minimum after around the same number of iterations but none manage to remain stable. It is difficult to tell which choice of batch size produced the best results but it appears that $n = 1$ was the worst.

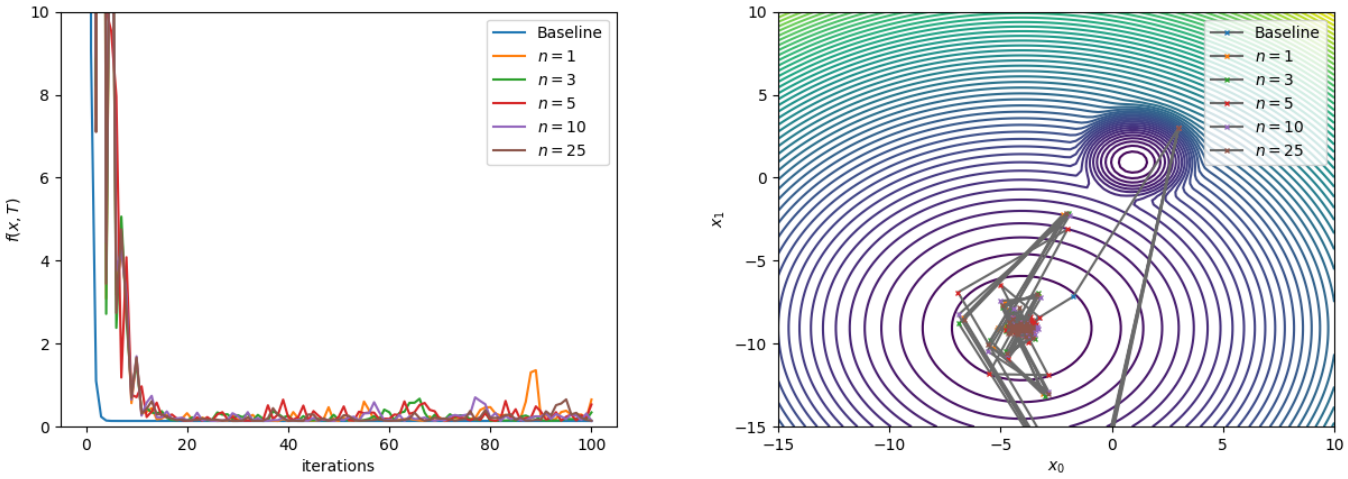

Figure 12: Change in x and $f(x, T)$ over iterations for different n and $\alpha = 10, \beta_1 = 0.9, \beta_2 = 0.999$ (SGD + Adam)

Like RMSProp, the step size does not stabilise due to the moving average of square gradient sums constantly changing during iterations.

Appendix A: Code

```
from copy import deepcopy
from enum import Enum
import matplotlib.pyplot as plt
import numpy as np
import sympy as sp

""" Assignment Functions """

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(
            18 * ((z[0] ** 2) + (z[1] ** 2)),
            ((z[0] + 10) ** 2) + ((z[1] + 5) ** 2)
        )
        count = count + 1
    return y / count

""" SGD Algorithm """

class StepSize(Enum):
    CONSTANT = 0
    POLYAK = 1
    RMSPROP = 2
    HEAVYBALL = 3
    ADAM = 4

class SGD:
    epsilon = 1e-8

    def __init__(self, f, df, x, algo, params, batch_size, training):
        # regular params
        self.f = f
        self.df = df
        self.x = deepcopy(x)
        self.n = len(x)
        self.params = params
        # minibatch params
        self.batch_size = batch_size
        self.T = training
        # store results for plotting, etc
        self.logs = {
            'x': [deepcopy(self.x)],
            'f': [self.f(self.x, self.T)],
            'step': []
        }
        # set up for specific step size function
        self.iter_function = self.__get_iter_function(algo)
        self.__init_function_vars(algo)

    def run_iter_minibatch(self):
        np.random.shuffle(self.T)
        N = len(self.T)
        for i in range(0, N, self.batch_size):
```

```
            if i + self.batch_size > N: continue
            sample = self.T[i:(i + self.batch_size)]
            self.iter_function(sample)
        self.logs['x'].append(deepcopy(self.x))
        self.logs['f'].append(self.f(self.x, self.T))

    def __get_iter_function(self, algo):
        if algo == StepSize.CONSTANT:
            return self.__iter_constant
        elif algo == StepSize.POLYAK:
            return self.__iter__polyak
        elif algo == StepSize.RMSPROP:
            return self.__iter__rmsprop
        elif algo == StepSize.HEAVYBALL:
            return self.__iter__heavy_ball
        else:
            return self.__iter__adam

    def __init_function_vars(self, algo):
        if algo == StepSize.RMSPROP:
            self.logs['step'] = [[self.params['alpha0']] * self.n]
            self.vars = {
                'sums': [0] * self.n,
                'alphas': [self.params['alpha0']] * self.n
            }
        elif algo == StepSize.HEAVYBALL:
            self.logs['step'] = [0]
            self.vars = {
                'z': 0
            }
        elif algo == StepSize.ADAM:
            self.logs['step'] = [[0] * self.n]
            self.vars = {
                'ms': [0] * self.n,
                'vs': [0] * self.n,
                'step': [0] * self.n,
                't': 0
            }

    def __iter_constant(self, sample):
        alpha = self.params['alpha']
        for i in range(self.n):
            self.x[i] -= alpha * self.__calc_approx_derivative(i, sample)
        self.logs['step'].append(alpha)

    def __iter__polyak(self, sample):
        step = self.f(self.x, sample) / (
            sum(self.__calc_approx_derivative(i, sample) ** 2
            for i in range(self.n)) + self.epsilon
        )
        for i in range(self.n):
            self.x[i] -= step * self.__calc_approx_derivative(i, sample)
        self.logs['step'].append(step)

    def __iter__rmsprop(self, sample):
        alpha0 = self.params['alpha0']
        beta = self.params['beta']
        alphas = self.vars['alphas']
        sums = self.vars['sums']
        for i in range(self.n):
            self.x[i] -= alphas[i] * self.__calc_approx_derivative(i, sample)
            sums[i] = (beta * sums[i]) + ((1 - beta) * (
                self.__calc_approx_derivative(i, sample) ** 2
```

```
            ))
            alphas[i] = alpha0 / ((sums[i] ** 0.5) + self.epsilon)
        self.logs['step'].append(deepcopy(alphas))

    def __iter__heavy_ball(self, sample):
        alpha = self.params['alpha']
        beta = self.params['beta']
        z = self.vars['z']
        z = (beta * z) + (alpha * self.f(self.x, sample) / (sum(
            self.__calc_approx_derivative(i, sample) ** 2
            for i in range(self.n)
        ) + self.epsilon))
        for i in range(self.n):
            self.x[i] -= z * self.__calc_approx_derivative(i, sample)
        self.vars['z'] = z
        self.logs['step'].append(z)

    def __iter__adam(self, sample):
        alpha = self.params['alpha']
        beta1 = self.params['beta1']
        beta2 = self.params['beta2']
        ms = self.vars['ms']
        vs = self.vars['vs']
        step = self.vars['step']
        t = self.vars['t']
        t += 1
        for i in range(self.n):
            ms[i] = (beta1 * ms[i]) + ((1 - beta1) *
                self.__calc_approx_derivative(i, sample))
            vs[i] = (beta2 * vs[i]) + ((1 - beta2) *
                (self.__calc_approx_derivative(i, sample) ** 2))
            m_hat = ms[i] / (1 - (beta1 ** t))
            v_hat = vs[i] / (1 - (beta2 ** t))
            step[i] = alpha * (m_hat / ((v_hat ** 0.5) + self.epsilon))
            self.x[i] -= step[i]
        self.vars['t'] = t
        self.logs['step'].append(deepcopy(step))

    def __calc_approx_derivative(self, i, sample):
        return sum(
            self.df[i](*self.x, *sample[j])
            for j in range(self.batch_size)
        ) / self.batch_size

""" Question (a) """

def a_ii():
    T = generate_trainingdata()
    X = np.linspace(-15, 10, 100)
    Y = np.linspace(-15, 10, 100)
    Z = []
    for x in X:
        z = []
        for y in Y: z.append(f([x, y], T))
        Z.append(z)
    Z = np.array(Z)
    X, Y = np.meshgrid(X, Y)
    _, (ax1, ax2) = plt.subplots(1, 2, subplot_kw=dict(projection='3d'))
    ax1.contour3D(X, Y, Z, 60)
    ax1.set_xlabel('$x_0$')
    ax1.set_ylabel('$x_1$')
    ax1.set_zlabel('$f(x, T)$')
    ax1.set_title('Contour plot of $f(x, T)$')
```

```
    ax2.plot_wireframe(X, Y, Z, rstride=5, cstride=5)
    ax2.set_xlabel('$x_0$')
    ax2.set_ylabel('$x_1$')
    ax2.set_zlabel('$f(x, T)$')
    ax2.set_title('Wireframe plot of $f(x, T)$')
    plt.show()

def a_iii():
    x0, x1, w0, w1 = sp.symbols('x0 x1 w0 w1', real=True)
    f = sp.Min(18*(((x0-w0-1)**2)+((x1-w1-1)**2)),(((x0-w0-1)+10)**2)+(((x1-w1-1)+5)**2))
    df0 = sp.diff(f, x0)
    df1 = sp.diff(f, x1)
    print(f)
    print(df0)
    print(df1)

""" Question (b) """

colors = [
    'tab:blue', 'tab:orange', 'tab:green', 'tab:red', 'tab:purple',
    'tab:brown', 'tab:pink', 'tab:gray', 'tab:olive', 'tab:cyan',
    'red', 'blue', 'lime'
]

def plot_contour(func, T, xs, fs, rngs=None, is_3d=True, legend=None):
    if rngs is None:
        X = np.linspace(-15, 10, 100)
        Y = np.linspace(-15, 10, 100)
    else:
        X = np.linspace(*rngs[0], 100)
        Y = np.linspace(*rngs[1], 100)
    Z = []
    for x in X:
        z = []
        for y in Y: z.append(func([x, y], T))
        Z.append(z)
    Z = np.array(Z)
    X, Y = np.meshgrid(X, Y)
    if is_3d:
        ax = plt.axes(projection='3d')
        ax.contour3D(X, Y, Z, 60)
        ax.set_xlabel('$x_0$')
        ax.set_ylabel('$x_1$')
        ax.set_zlabel('$f(x, T)$')
    else:
        plt.contour(X, Y, Z, 60)
        plt.xlabel('$x_0$')
        plt.ylabel('$x_1$')
    for i in range(len(xs)):
        x0 = [x[1] for x in xs[i]]
        x1 = [x[0] for x in xs[i]]
        if is_3d:
            ax.plot(x0, x1, fs[i], color='dimgrey',
                marker='x', markeredgecolor=colors[i], markersize=3)
        else:
            plt.plot(x0, x1, color='dimgrey',
                marker='x', markeredgecolor=colors[i], markersize=3)
            plt.xlim([-15, 10])
            plt.ylim([-15, 10])
    if legend is not None: plt.legend(legend)
    plt.show()

def b_i(func, df):
```

```
    T = generate_trainingdata()
    num_iters = 100
    ci = 0
    iters = list(range(num_iters + 1))
    alphas = [0.1, 0.01, 0.001, 0.0001]
    labels = [f'$\\alpha={alpha}$' for alpha in alphas]
    xs, fs = [], []
    for i, alpha in enumerate(alphas):
        sgd = SGD(func, df, [3, 3], StepSize.CONSTANT, {'alpha': alpha},
            batch_size=len(T), training=T)
        for _ in range(num_iters):
            sgd.run_iter_minibatch() # not actually mini-batch since batch_size=|T|
        plt.plot(iters, sgd.logs['f'], label=labels[i], color=colors[ci])
        ci += 1
        xs.append(deepcopy(sgd.logs['x']))
        fs.append(deepcopy(sgd.logs['f']))
    plt.ylim([0, 150])
    plt.xlabel('iterations')
    plt.ylabel('$f(x, T)$')
    plt.legend()
    plt.show()
    plot_contour(func, T, xs, fs, is_3d=False, legend=labels)

def b_ii(func, df):
    T = generate_trainingdata()
    num_trials = 5
    num_iters = 10
    ci = 0
    iters = list(range(num_iters + 1))
    alpha = 0.1
    labels = [f'trial ${i+1}$' for i in range(num_trials)]
    xs, fs = [], []
    for trial in range(num_trials):
        sgd = SGD(func, df, [3, 3], StepSize.CONSTANT, {'alpha': alpha}, 5, T)
        for _ in range(num_iters):
            sgd.run_iter_minibatch()
        plt.plot(iters, sgd.logs['f'], label=labels[trial], color=colors[ci])
        ci += 1
        xs.append(deepcopy(sgd.logs['x']))
        fs.append(deepcopy(sgd.logs['f']))
    plt.ylim([0, 20])
    plt.xlabel('iterations')
    plt.ylabel('$f(x, T)$')
    plt.legend()
    plt.show()
    plot_contour(func, T, xs, fs, is_3d=False, legend=labels)

def b_iii(func, df):
    T = generate_trainingdata()
    num_iters = 25
    ci = 0
    iters = list(range(num_iters + 1))
    alpha = 0.1
    batch_sizes = [1, 3, 5, 10, 25]
    labels = [f'$n={n}$' for n in batch_sizes]
    xs, fs = [], []
    for i, n in enumerate(batch_sizes):
        sgd = SGD(func, df, [3, 3], StepSize.CONSTANT, {'alpha': alpha}, n, T)
        for _ in range(num_iters):
            sgd.run_iter_minibatch()
        plt.plot(iters, sgd.logs['f'], label=labels[i], color=colors[ci])
        ci += 1
        xs.append(deepcopy(sgd.logs['x']))
```

```
        fs.append(deepcopy(sgd.logs['f']))
    plt.ylim([0, 3])
    plt.xlabel('iterations')
    plt.ylabel('$f(x, T)$')
    plt.legend()
    plt.show()
    plot_contour(func, T, xs, fs, is_3d=False, legend=labels)

def b_iv(func, df):
    T = generate_trainingdata()
    num_iters = 25
    ci = 0
    iters = list(range(num_iters + 1))
    alphas = [0.1, 0.01, 0.001, 0.0001]
    labels = [f'$\\alpha={alpha}$' for alpha in alphas]
    xs, fs = [], []
    for i, alpha in enumerate(alphas):
        sgd = SGD(func, df, [3, 3], StepSize.CONSTANT, {'alpha': alpha}, 5, T)
        for _ in range(num_iters):
            sgd.run_iter_minibatch()
        plt.plot(iters, sgd.logs['f'], label=labels[i], color=colors[ci])
        ci += 1
        xs.append(deepcopy(sgd.logs['x']))
        fs.append(deepcopy(sgd.logs['f']))
    plt.ylim([0, 120])
    plt.xlabel('iterations')
    plt.ylabel('$f(x, T)$')
    plt.legend()
    plt.show()
    plot_contour(func, T, xs, fs, is_3d=False, legend=labels)

""" Question (c) """

def c_i(func, df):
    T = generate_trainingdata()
    num_iters = 100
    iters = list(range(num_iters + 1))
    # results
    ci = 0
    xs, fs = [], []
    labels = ['Baseline']
    # baseline
    sgd_bl = SGD(func, df, [3, 3], StepSize.CONSTANT, {'alpha': 0.1}, 5, T)
    for _ in range(num_iters):
        sgd_bl.run_iter_minibatch()
    plt.plot(iters, sgd_bl.logs['f'], label=labels[0], color=colors[ci])
    ci += 1
    xs.append(deepcopy(sgd_bl.logs['x']))
    fs.append(deepcopy(sgd_bl.logs['f']))
    # polyak
    batch_sizes = [1, 3, 5, 10, 25]
    for n in batch_sizes:
        sgd = SGD(func, df, [3, 3], StepSize.POLYAK, {}, n, T)
        for _ in range(num_iters):
            sgd.run_iter_minibatch()
        labels.append(f'$n={n}$')
        plt.plot(iters, sgd.logs['f'], label=labels[-1], color=colors[ci])
        ci += 1
        xs.append(deepcopy(sgd.logs['x']))
        fs.append(deepcopy(sgd.logs['f']))
    # plotting
    plt.ylim([0, 60])
    plt.xlabel('iterations')
```

```
    plt.ylabel('$f(x, T)$')
    plt.legend()
    plt.show()
    # mini-batch variation
    plot_contour(func, T, xs, fs, is_3d=False, legend=labels)

def c_ii(func, df):
    T = generate_trainingdata()
    num_iters = 100
    iters = list(range(num_iters + 1))
    # results
    ci = 0
    xs, fs = [], []
    labels = ['Baseline']
    # baseline
    sgd_bl = SGD(func, df, [3, 3], StepSize.CONSTANT, {'alpha': 0.1}, 5, T)
    for _ in range(num_iters):
        sgd_bl.run_iter_minibatch()
    plt.plot(iters, sgd_bl.logs['f'], label=labels[0], color=colors[ci])
    ci += 1
    xs.append(deepcopy(sgd_bl.logs['x']))
    fs.append(deepcopy(sgd_bl.logs['f']))
    # parameter selection
    alpha0s = [0.1, 0.01, 0.001]
    betas = [0.25, 0.9]
    for alpha0 in alpha0s:
        for beta in betas:
            sgd = SGD(func, df, [3, 3], StepSize.RMSPROP,
                {'alpha0': alpha0, 'beta': beta}, 5, T)
            for _ in range(num_iters):
                sgd.run_iter_minibatch()
            labels.append(f'$\\alpha_0={alpha0},\\,\\beta={beta}$')
            plt.plot(iters, sgd.logs['f'], label=labels[-1], color=colors[ci])
            ci += 1
            xs.append(deepcopy(sgd.logs['x']))
            fs.append(deepcopy(sgd.logs['f']))
    # plotting
    plt.ylim([0, 60])
    plt.xlabel('iterations')
    plt.ylabel('$f(x, T)$')
    plt.legend()
    plt.show()
    # optimal parameters
    ci = 0
    xs, fs = [], []
    labels = ['Baseline']
    plt.plot(iters, sgd_bl.logs['f'], label=labels[0], color=colors[ci])
    ci += 1
    xs.append(deepcopy(sgd_bl.logs['x']))
    fs.append(deepcopy(sgd_bl.logs['f']))
    batch_sizes = [1, 3, 5, 10, 25]
    for n in batch_sizes:
        sgd = SGD(func, df, [3, 3], StepSize.RMSPROP, {'alpha0': 0.1, 'beta': 0.9}, 5, T)
        for _ in range(num_iters):
            sgd.run_iter_minibatch()
        labels.append(f'$n={n}$')
        plt.plot(iters, sgd.logs['f'], label=labels[-1], color=colors[ci])
        ci += 1
        xs.append(deepcopy(sgd.logs['x']))
        fs.append(deepcopy(sgd.logs['f']))
    # plotting
    plt.ylim([0, 10])
    plt.xlabel('iterations')
```

```
    plt.ylabel('$f(x, T)$')
    plt.legend()
    plt.show()
    plot_contour(func, T, xs, fs, is_3d=False, legend=labels)

def c_iii(func, df):
    T = generate_trainingdata()
    num_iters = 100
    iters = list(range(num_iters + 1))
    # results
    ci = 0
    xs, fs = [], []
    labels = ['Baseline']
    # baseline
    sgd_bl = SGD(func, df, [3, 3], StepSize.CONSTANT, {'alpha': 0.1}, 5, T)
    for _ in range(num_iters):
        sgd_bl.run_iter_minibatch()
    plt.plot(iters, sgd_bl.logs['f'], label=labels[0], color=colors[ci])
    ci += 1
    xs.append(deepcopy(sgd_bl.logs['x']))
    fs.append(deepcopy(sgd_bl.logs['f']))
    # parameter selection
    alphas = [0.01, 0.001]
    betas = [0.25, 0.5, 0.9]
    for alpha in alphas:
        for beta in betas:
            sgd = SGD(func, df, [3, 3], StepSize.HEAVYBALL,
                {'alpha': alpha, 'beta': beta}, 5, T)
            for _ in range(num_iters):
                sgd.run_iter_minibatch()
            labels.append(f'$\\alpha={alpha},\\,\\beta={beta}$')
            plt.plot(iters, sgd.logs['f'], label=labels[-1], color=colors[ci])
            ci += 1
            xs.append(deepcopy(sgd.logs['x']))
            fs.append(deepcopy(sgd.logs['f']))
    # plotting
    plt.ylim([0, 60])
    plt.xlabel('iterations')
    plt.ylabel('$f(x, T)$')
    plt.legend()
    plt.show()
    # optimal parameters
    ci = 0
    xs, fs = [], []
    labels = ['Baseline']
    plt.plot(iters, sgd_bl.logs['f'], label=labels[0], color=colors[ci])
    ci += 1
    xs.append(deepcopy(sgd_bl.logs['x']))
    fs.append(deepcopy(sgd_bl.logs['f']))
    batch_sizes = [1, 3, 5, 10, 25]
    for n in batch_sizes:
        sgd = SGD(func, df, [3, 3], StepSize.HEAVYBALL,
            {'alpha': 0.01, 'beta': 0.9}, 5, T)
        for _ in range(num_iters):
            sgd.run_iter_minibatch()
        labels.append(f'$n={n}$')
        plt.plot(iters, sgd.logs['f'], label=labels[-1], color=colors[ci])
        ci += 1
        xs.append(deepcopy(sgd.logs['x']))
        fs.append(deepcopy(sgd.logs['f']))
    # plotting
    plt.ylim([0, 10])
    plt.xlabel('iterations')
```

```
    plt.ylabel('$f(x, T)$')
    plt.legend()
    plt.show()
    plot_contour(func, T, xs, fs, is_3d=False, legend=labels)

def c_iv(func, df):
    T = generate_trainingdata()
    num_iters = 100
    iters = list(range(num_iters + 1))
    # results
    ci = 0
    xs, fs = [], []
    labels = ['Baseline']
    # baseline
    sgd_bl = SGD(func, df, [3, 3], StepSize.CONSTANT, {'alpha': 0.1}, 5, T)
    for _ in range(num_iters):
        sgd_bl.run_iter_minibatch()
    plt.plot(iters, sgd_bl.logs['f'], label=labels[0], color=colors[ci])
    ci += 1
    xs.append(deepcopy(sgd_bl.logs['x']))
    fs.append(deepcopy(sgd_bl.logs['f']))
    # parameter selection
    alphas = [10, 1, 0.1]
    beta1s = [0.25, 0.9]
    beta2s = [0.999]
    for alpha in alphas:
        for beta1 in beta1s:
            for beta2 in beta2s:
                sgd = SGD(func, df, [3, 3], StepSize.ADAM,
                    {'alpha': alpha, 'beta1': beta1, 'beta2': beta2}, 5, T)
                for _ in range(num_iters):
                    sgd.run_iter_minibatch()
                labels.append(
                    f'$\\alpha={alpha},\\,\\beta_1={beta1},\\,\\beta_2={beta2}$'
                )
                plt.plot(iters, sgd.logs['f'], label=labels[-1], color=colors[ci])
                ci += 1
                xs.append(deepcopy(sgd.logs['x']))
                fs.append(deepcopy(sgd.logs['f']))
    # plotting
    plt.ylim([0, 60])
    plt.xlabel('iterations')
    plt.ylabel('$f(x, T)$')
    plt.legend()
    plt.show()
    # optimal parameters
    ci = 0
    xs, fs = [], []
    labels = ['Baseline']
    plt.plot(iters, sgd_bl.logs['f'], label=labels[0], color=colors[ci])
    ci += 1
    xs.append(deepcopy(sgd_bl.logs['x']))
    fs.append(deepcopy(sgd_bl.logs['f']))
    batch_sizes = [1, 3, 5, 10, 25]
    for n in batch_sizes:
        sgd = SGD(func, df, [3, 3], StepSize.ADAM,
            {'alpha': 10, 'beta1': 0.9, 'beta2': 0.999}, 5, T)
        for _ in range(num_iters):
            sgd.run_iter_minibatch()
        labels.append(f'$n={n}$')
        plt.plot(iters, sgd.logs['f'], label=labels[-1], color=colors[ci])
        ci += 1
        xs.append(deepcopy(sgd.logs['x']))
```

```
            fs.append(deepcopy(sgd.logs['f']))
        # plotting
        plt.ylim([0, 10])
        plt.xlabel('iterations')
        plt.ylabel('$f(x, T)$')
        plt.legend()
        plt.show()
        plot_contour(func, T, xs, fs, is_3d=False, legend=labels)

""" Calling Functions """

a_ii()
a_iii()
df0 = lambda x0, x1, w0, w1:\
    (-36*w0+36*x0-36)*np.heaviside(-18*(-w0+x0-1)**2+(-w0+x0+9)**2-18*(-w1+x1-1)**2+(-w1+x
    (-2*w0+2*x0+18)*np.heaviside(18*(-w0+x0-1)**2-(-w0+x0+9)**2+18*(-w1+x1-1)**2-(-w1+x1+4
df1 = lambda x0, x1, w0, w1:\
    (-36*w1+36*x1-36)*np.heaviside(-18*(-w0+x0-1)**2+(-w0+x0+9)**2-18*(-w1+x1-1)**2+(-w1+x
    (-2*w1+2*x1+8)*np.heaviside(18*(-w0+x0-1)**2-(-w0+x0+9)**2+18*(-w1+x1-1)**2-(-w1+x1+4)
b_i(f, [df0, df1])
b_ii(f, [df0, df1])
b_iii(f, [df0, df1])
b_iv(f, [df0, df1])
c_i(f, [df0, df1])
c_ii(f, [df0, df1])
c_iii(f, [df0, df1])
c_iv(f, [df0, df1])
```