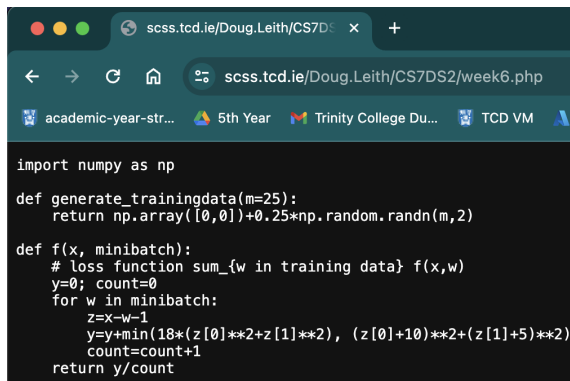


## Loss function

The loss function I will use for this assignment are:



```
import numpy as np

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
```

Figure 1: Loss function obtained

```
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
```

Table 1: Loss function in code

## Question (a)

### Part (i)

I implemented the mini-batch stochastic gradient descent algorithm with the ability to use a constant step size, Polyak, RMSProp, Heavy Ball or Adam steps inside of `stochastic_gradient_descent.py`. The file contains a class with parameters for the function itself being minimized, the derivative functions,  $x$ 's starting value, the choice of step size, the hyperparameters (inside a Python Dict), the batch size and training data itself. An iteration of minibatch stochastic gradient descent is implemented through this function:

```
def minibatch_iteration(self):
    np.random.shuffle(self.training_data)
    for starting_index_of_minibatch in range(0, len(self.training_data),
    self.batch_size):
        if starting_index_of_minibatch + self.batch_size >
        len(self.training_data):
            continue
        training_data_sample =
        self.training_data[starting_index_of_minibatch:
        (starting_index_of_minibatch + self.batch_size)]
        self.iteration_function(training_data_sample)
        self.logs['function_value'].append(self.function_being_minimized(
        self.x_starting_value, self.training_data))
        self.logs['x_value'].append(deepcopy(self.x_starting_value))
```

This function shuffles the training data, iterates over samples of `batch_size` length from the training data, and calls the `iteration_function` for the sample, while simultaneously not sampling from outside the training data array. Different choices for step size results in different operation - for example, using a constant step size results in `iteration_function` calling these:

```
def compute_approximate_derivative(self, i, sample):
    return sum(
```

```

        self.derivative_functions[i](*self.x_starting_value,
        *sample[batch_size])
    for batch_size in range(self.batch_size) / self.batch_size

def constant_step_size(self, sample):
    a = self.hyperparameters_dict['alpha']
    self.logs['step'].append(a)
    for i in range(self.number_of_parameters):
        self.x_starting_value[i] -= a *
        self.compute_approximate_derivative(i, sample)

```

Which shows that there are only minor differences between this and the previous assignment, such as `array_of_lambda_functions` instead being a function call to computing the approximate derivative using the mini-batch sample.

## Part (ii)

I plotted a wireframe and contour plot of the function  $f(x, T)$  for  $N = T$  using the range  $-15 \leq x_1 \leq 10$  and  $-15 \leq x_0 \leq 10$ :

```

# Ranges -15 <= x_0 <= 10 and -15 <= x_1 <= 10
X = np.linspace(-15, 10, 100)
Y = np.linspace(-15, 10, 100)
Z = []
T = generate_trainingdata()
for x_val in X:
    z = []
    for y_val in Y:
        z.append(f([x_val, y_val], T))
    Z.append(z)
_, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5),
subplot_kw=dict(projection='3d'), gridspec_kw={'wspace': 0.2})

```

I used this since it displays how  $x_1 > 5$  causes the function to be steep and  $x_1 < 1$  causes the function to be flat while simultaneously showing the minima of the function. Using these justified ranges, the full training data  $T$  was used for both plots to show a fair comparison:

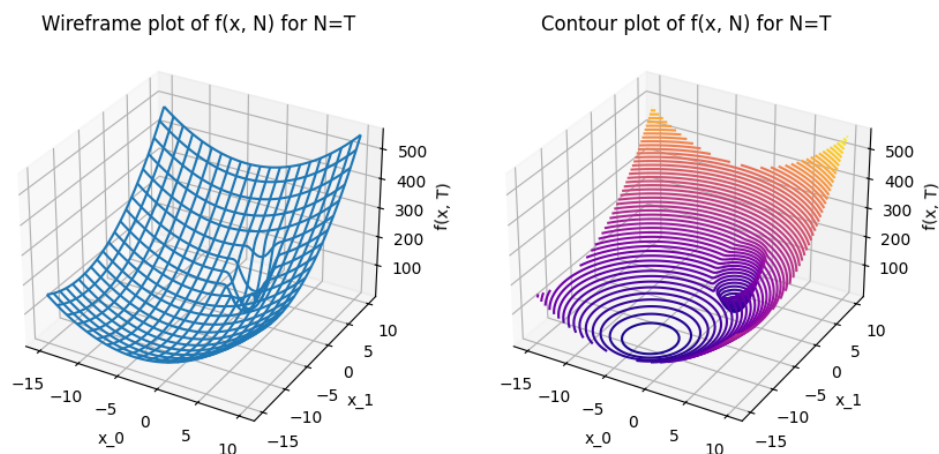


Figure 2: Wireframe and contour plots of  $f(x, T)$

### Part (iii)

Using SymPy, we can find the derivative of  $f$ . We need to find the derivative of this part where  $z_i = x_i - w_i - 1, w \in T: \min(18(z_0^2 + z_1^2), (z_0 + 10)^2 + (z_1 + 5)^2)$

Therefore, the SymPy function can use the symbols  $x_0, x_1, w_0, w_1$  so we can differentiate it using `diff()` with respect to  $x_0$  once and with respect to  $x_1$  once thereafter.

```
x0, x1, w0, w1 = sp.symbols('x0 x1 w0 w1', real=True)
function_to_differentiate = sp.Min(18*((x0-w0-1)**2)+((x1-w1-1)**2),
((x0-w0-1)+10)**2)+((x1-w1-1)+5)**2))
differentiate_with_respect_to_x0 = sp.diff(function_to_differentiate, x0)
differentiate_with_respect_to_x1 = sp.diff(function_to_differentiate, x1)
```

This results in:

**Differentiate with respect to  $x_0$ :**  $(-36*w_0 + 36*x_0 - 36)*Heaviside(-18*(-w_0 + x_0 - 1)**2 + (-w_0 + x_0 + 9)**2 - 18*(-w_1 + x_1 - 1)**2 + (-w_1 + x_1 + 4)**2) + (-2*w_0 + 2*x_0 + 18)*Heaviside(18*(-w_0 + x_0 - 1)**2 - (-w_0 + x_0 + 9)**2 + 18*(-w_1 + x_1 - 1)**2 - (-w_1 + x_1 + 4)**2)$

**Differentiate with respect to  $x_1$ :**  $(-36*w_1 + 36*x_1 - 36)*Heaviside(-18*(-w_0 + x_0 - 1)**2 + (-w_0 + x_0 + 9)**2 - 18*(-w_1 + x_1 - 1)**2 + (-w_1 + x_1 + 4)**2) + (-2*w_1 + 2*x_1 + 8)*Heaviside(18*(-w_0 + x_0 - 1)**2 - (-w_0 + x_0 + 9)**2 + 18*(-w_1 + x_1 - 1)**2 - (-w_1 + x_1 + 4)**2)$

## Question (b)

### Part (i)

I used the contents of `stochastic_gradient_descent.py` from part (a) (i) since we can use a batch size equal to the length of the training data using a constant step size to imitate classical gradient descent. For finding a step size, I ran 100 iterations using a range of  $\alpha$  values of  $[0.1, 0.01, 0.001, 0.0001]$  for each  $\alpha$  to produce this plot:

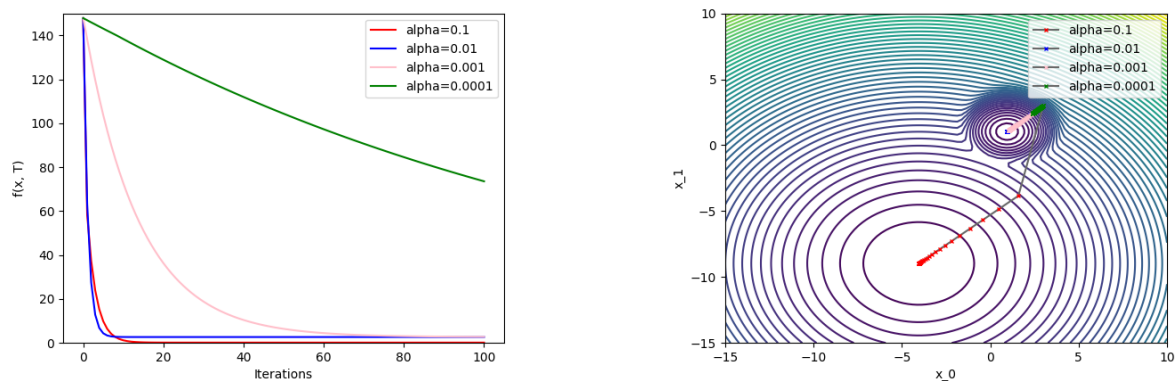


Figure 3: A normal plot and contour plot showing 100 iterations for visualizing the change in  $x$  and  $f(x, T)$  for different values of  $\alpha$  (Classical gradient descent with constant step size)

Using  $\alpha = 0.0001$  does not allow the algorithm to converge. Using  $\alpha = 0.001$  and  $\alpha = 0.01$  makes the algorithm not converge at the global minimum. However,  $\alpha = 0.1$  converges to a minimum that is not precisely zero but is the closest, while other values of  $\alpha$  do not allow the algorithm to converge on the global minimum because the training data has

noise added to it which causes them to converge to a minimum that is bigger than zero and not as close to the minimum (as shown in the contour for the function's noiseless minimum). Hence,  $\alpha = 0.1$  is the best option.

### Part (ii)

With  $\alpha = 0.1$ , I ran 5 attempts of stochastic gradient descent with constant step size, with each having 10 iterations with the same training data.

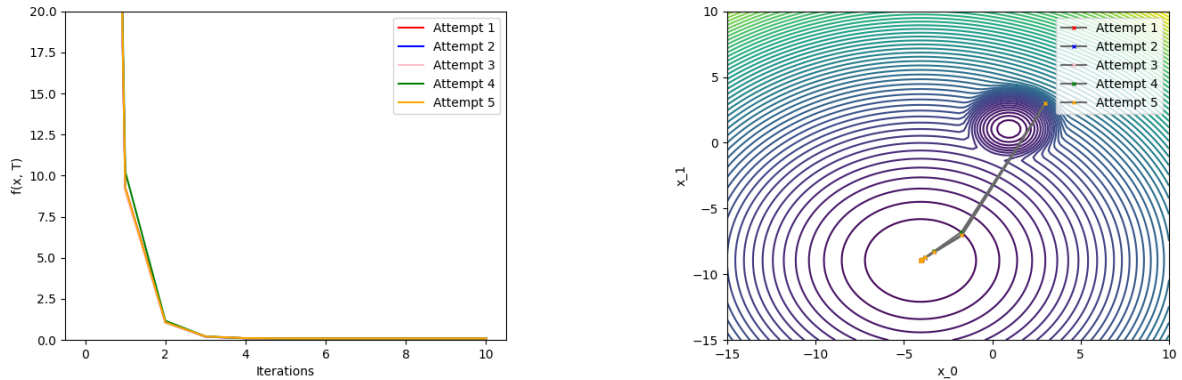


Figure 4: A normal plot and contour plot showing 5 attempts with 10 iterations each to visualize the change in  $x$  and  $f(x, T)$  using  $\alpha = 0.1$  (Stochastic gradient descent with constant step size)

The plot above shows that each attempt has little difference between them, meaning that the shuffling of training data from each iteration of stochastic gradient descent has minor effect on the result even though it adds randomness from shuffling. It took about six iterations for each attempt to converge to a minimum, compared to around ten iterations for the most optimal convergence in part (b) (i). Overall, every trial converged on the global minimum instead of a local minimum with the noisy training data, and is around twice as fast in terms of iterations compared to part (b) (i).

### Part (iii)

I applied a range of batch sizes  $[1, 3, 5, 10, 25]$  for multiple attempts with 25 iterations per attempt.

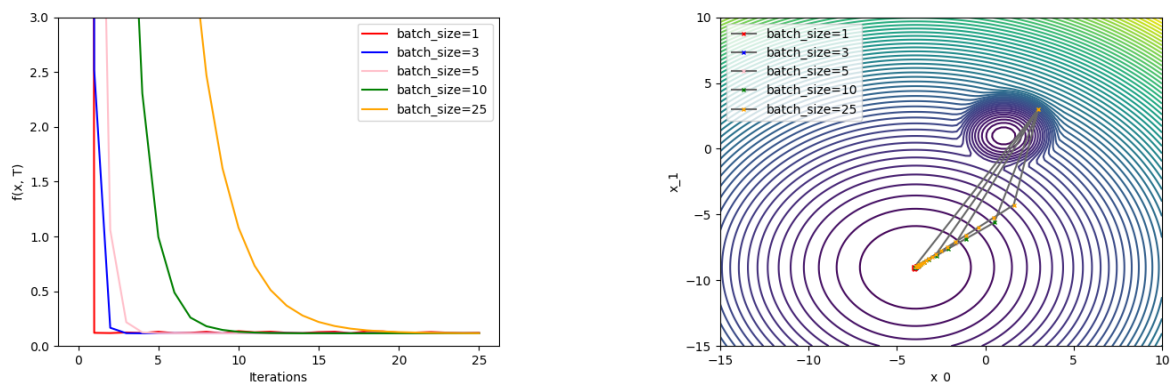


Figure 5: A normal plot and contour plot showing different batch sizes with 25 iterations per attempt to visualize the change in  $x$  and  $f(x, T)$  using  $\alpha = 0.1$  (Stochastic gradient descent with constant step size)

The plots show that all attempts with various batch sizes converge to a global minimum eventually, since early iterations expand  $x$  away from the local minimum that they were initially beside, because of  $\alpha$  being big enough. Increasing the batch size increases the number of iterations required for convergence, with the exception of a batch size of 1 (small values) which seems to not converge properly (with small fluctuations) due to big changes to  $x$  in each iteration from the **noise** in each separate sample. After a batch size of 3 and above, this issue begins to stop, and a batch size of 5 and above stops this issue completely.

### Part (iv)

I used a range of values for  $\alpha$  as  $[0.1, 0.01, 0.001, 0.0001]$  from part (b) (i), and ran each attempt for 25 iterations.

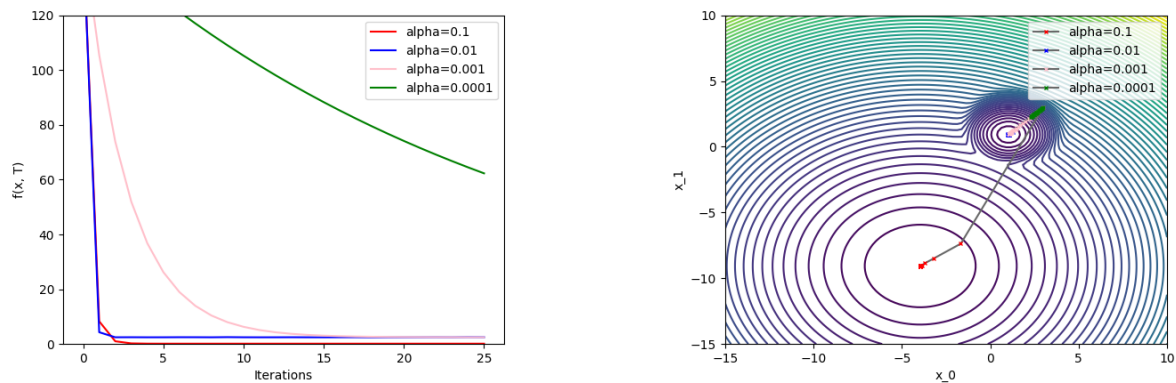


Figure 6: A normal plot and contour plot showing 25 iterations for each attempt to visualize the change in  $x$  and  $f(x, T)$  for different values of  $\alpha$  (Stochastic gradient descent with constant step size)

All trials converge towards a local minimum which is close to  $x$ 's initial value except  $\alpha = 0.1$  which converges to the global minimum. As mentioned in part (b) (iii), this phenomenon occurs because of smaller values of  $\alpha$  in the early iterations not expanding  $x$  away enough from the initial value. More noise that is introduced to the training data means that the algorithm is less likely to converge on the local minimum (limited) and instead more likely to converge on the global minimum (broad) since the global minimum is consistent and flat instead of the local minimum which is limited and steeper. However, the lack of enough noise in the training data causes only all attempts to converge to a local minimum except when  $\alpha = 0.1$ , which also takes the least amount of time for convergence like before. Smaller values of alpha take the most time to converge. Hence, the best value for  $\alpha$  is still  $\alpha = 0.1$  since it reaches convergence quickest to a global minimum (unlike  $\alpha = 0.01$  which is faster but only reaches a local minimum).

### Question (c)

#### Part (i)

I used mini-batch SGD to minimise the loss function with Polyak step size by running it for 100 iterations for various batch sizes  $[1, 3, 5, 10, 25]$  and using a mini-batch SGD with a mini-batch size of 5 and  $\alpha = 0.1$  from part (b) (ii) as a baseline for comparison. No hyperparameters were needed to be chosen.



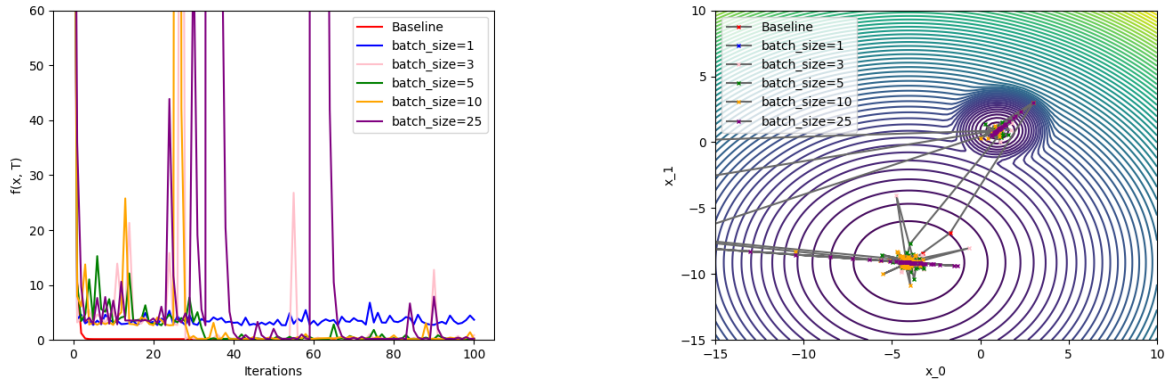


Figure 7: A normal plot and contour plot showing different batch sizes with 100 iterations per attempt to visualize the change in  $x$  and  $f(x, T)$  (Stochastic gradient descent with Polyak step size)

Interestingly, the plot shows that all attempts are worse than the baseline in red. A small batch size such as 1 converges to a local minimum but is not stable and fluctuates a lot (shown in blue). Other cases converge to a global minimum but are still not stable and fluctuate a lot in comparison to the baseline (in red). An observation I made was that all attempts (apart from the baseline and a batch size of 1) initially were going to converge to the local minimum but ended up aligning to the global minimum eventually to converge appropriately. Overall, using Polyak step size is not suitable since it leads to results that fluctuate compared to a baseline. It does not converge to a global minimum all the time, and even if it does, it seems to be unstable. This is because of the step sizes being too big in each iteration for flat parts of the function because of the squares of the approximate derivatives being smaller than the value of the function.

## Part (ii)

I used mini-batch SGD to minimise the loss function with RMSProp step size by running it for 100 iterations for various batch sizes [1, 3, 5, 10, 25] and using a mini-batch SGD with  $\alpha = 0.1$  from part (b) as a baseline for comparison. I used a range of hyperparameters for  $\alpha_0$  of [0.1, 0.01, 0.001] and  $\beta$  of [0.25, 0.9]:

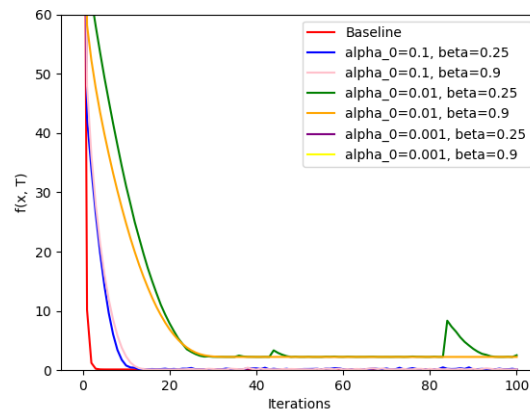


Figure 8: Change in  $x$  and  $f(x, T)$  using different values of  $\alpha_0$  and  $\beta$  for 100 iterations (Stochastic gradient descent with RMSProp step size)

It is evident from the plot that using  $\alpha_0 = 0.1$  is the best for converging on a global minimum with a stable result and little fluctuation using  $\beta = 0.9$ . Hence, using these values for various batch sizes, we get these plots:

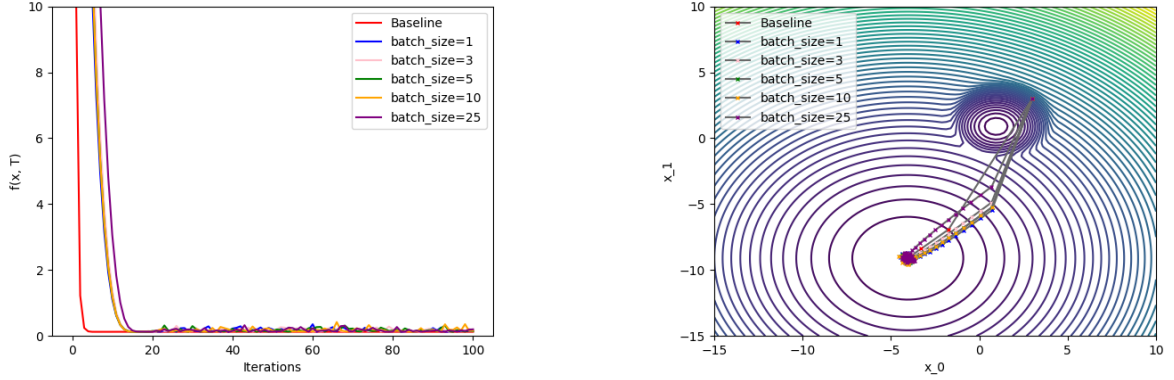


Figure 9: A normal plot and contour plot showing different batch sizes with 100 iterations per attempt to visualize the change in  $x$  and  $f(x, T)$  using  $\alpha_0 = 0.1$  and  $\beta = 0.9$  (Stochastic gradient descent with RMSProp step size)

From the plots, we observe that using RMSProp does not converge at a global minimum quicker than the baseline (in red), but the attempts are better than using the Polyak step size in part (c) (i). Flatter parts of the region do not result in increased step size (which Polyak had), with the step size not becoming stable as it depends on the moving average of square gradient sums which changes during each iteration constantly. Using RMSProp takes around 15 iterations to converge and the end result is not stable, while the baseline takes around 5 iterations (3 times less iterations required) with a stable result. The best options for limiting the fluctuations with RMSProp seems to be using a batch size of 3 or 5, which aligns with discoveries in part (b) (iii).

### Part (iii)

I used mini-batch SGD to minimise the loss function with Heavyball step size by running it for 100 iterations for various batch sizes [1, 3, 5, 10, 25] and using a mini-batch SGD with  $\alpha = 0.1$  from part (b) as a baseline for comparison. I used a range of hyperparameters for  $\alpha$  of [0.01, 0.001] and  $\beta$  of [0.25, 0.5, 0.9]:

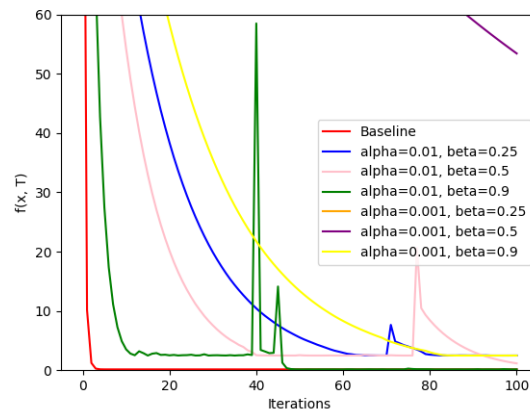


Figure 10: Change in  $x$  and  $f(x, T)$  for different values of  $\alpha$  and  $\beta$  for 100 iterations (Stochastic gradient descent with Heavyball step size)

Using  $\alpha = 0.01$  results in convergence at the global minimum, with  $\beta = 0.9$  converging the quickest (in green). Hence, we can use these hyperparameters to vary the batch size and make observations.

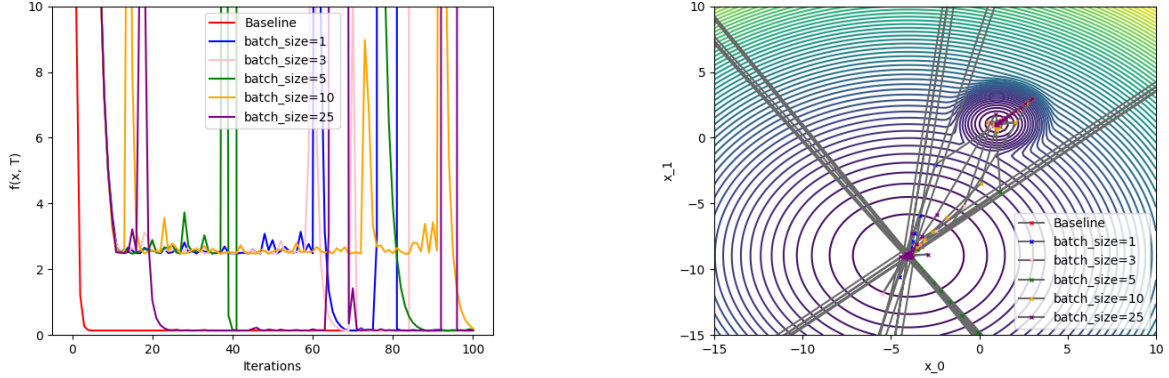


Figure 11: A normal plot and contour plot showing different batch sizes with 100 iterations per attempt to visualize the change in  $x$  and  $f(x, T)$  using  $\alpha_0 = 0.01$  and  $\beta = 0.9$  (Stochastic gradient descent with Heavyball step size)

From the plot it is evident that the attempts converge to a local minimum at the start before aiming for the global minimum after some time. They are not stable as they diverge from the global minimum after some iterations and subsequently come back to the global minimum. Bigger values for batch size such as 10 and 25 eventually converge to a global minimum in a relatively stable manner compared to the rest, but it is not clear if even this result is reliable or not. However, it is clear that the baseline is much better than the rest of the attempts. The fluctuations are because of the step size being too big in flat regions of the function (like Polyak) and adding  $z\beta$  during each iteration multiple times causing large fluctuations (even though this is reduced a bit by  $\alpha$ ).

#### Part (iv)

I used mini-batch SGD to minimise the loss function with Adam step size by running it for 100 iterations for various batch sizes [1, 3, 5, 10, 25] and with part (b) as a baseline. I used a range of hyperparameters for  $\alpha$  of [10, 1, 0.1] and  $\beta_1$  of [0.25, 0.9] where  $\beta_2$  was a constant of 0.999 (to ensure that enough history of squared gradients are accumulated).

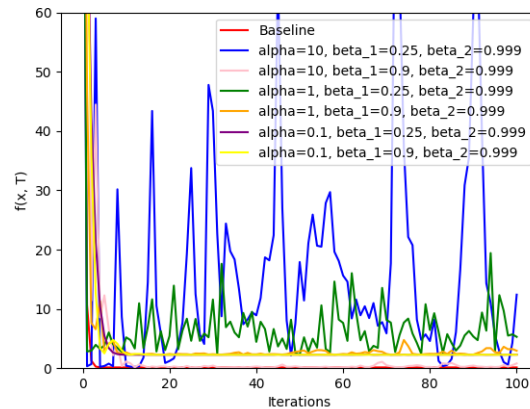


Figure 12: Change in  $x$  and  $f(x, T)$  for different values of  $\alpha$  and  $\beta_1$  ( $\beta_2 = 0.999$ ) for 100 iterations (Stochastic gradient descent with Adam step size)



Using  $\alpha = 10$  and  $\beta_1 = 0.9$  is the most optimal since it was the only attempt (in pink) that converged on the global minimum with relatively stable results across iterations. Using smaller values for  $\alpha$  results in not converging on the global minimum, while smaller values of  $\beta_1$  cause unstable results (i.e. the blue line having massive divergences).

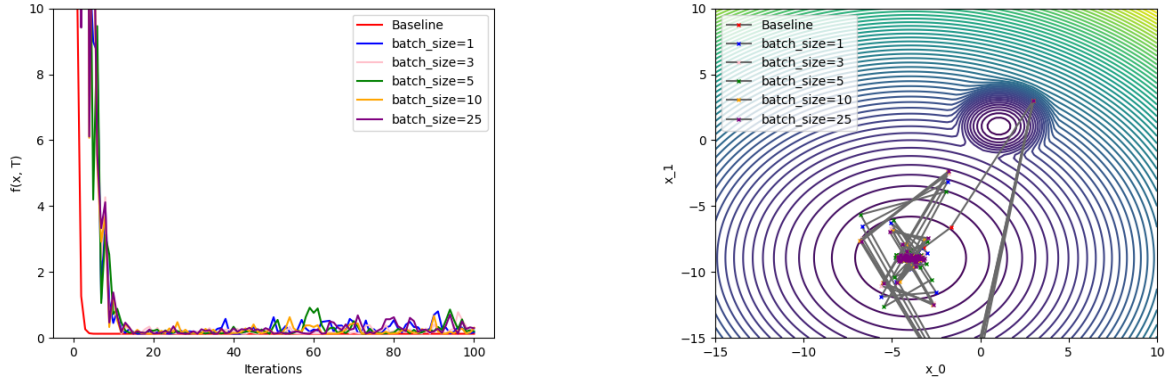


Figure 13: A normal plot and contour plot showing different batch sizes for 100 iterations to visualize the change in  $x$  and  $f(x, T)$  using  $\alpha = 10$ ,  $\beta_1 = 0.9$  and  $\beta_2 = 0.999$  (Stochastic gradient descent with Adam step size)

It is evident from the plot that no attempt leads to stable results, but they all (except baseline) converge to the global minimum eventually around the same number of iterations ( $\approx 15$ ). Since all attempts are very similar, it is unknown which batch size is the best, however it is evident that the baseline takes less iterations to converge on the global minimum. A batch size of 5 (in green) seems to have distinct divergences at around 60 and 90 iterations (alongside their constant smaller fluctuations, similar to the rest) which may suggest it is the worst but once again, they are all very similar. The step size doesn't become stable since the moving average of square gradient sums in each iteration is changing continuously (like in RMSProp).

## 1 Appendix

### 1.1 Loss function

```
import numpy as np

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
```

## 1.2 Contour plot utility function

```
import numpy as np
from matplotlib import pyplot as plt

colors = [
    'red', 'blue', 'pink', 'green', 'orange',
    'purple', 'yellow', 'cyan', 'grey', 'olive',
]

def generate_contour_plot(filepath, func, T, x_values, function_values,
    range_for_plot=None, three_dimensional=True, legend=None):
    Z = []
    if range_for_plot is not None:
        X = np.linspace(*range_for_plot[0], 100)
        Y = np.linspace(*range_for_plot[1], 100)
    else:
        X = np.linspace(-15, 10, 100)
        Y = np.linspace(-15, 10, 100)
    for x in X:
        z = []
        for y in Y:
            z.append(func([x, y], T))
        Z.append(z)
    X, Y = np.meshgrid(X, Y)
    Z = np.array(Z)
    if not three_dimensional:
        plt.contour(X, Y, Z, 60)
        plt.xlabel('x_0')
        plt.ylabel('x_1')
        for i in range(len(x_values)):
            x1 = [x[0] for x in x_values[i]]
            x0 = [x[1] for x in x_values[i]]
            plt.plot(x0, x1, markersize=3, marker='x', color='dimgrey',
                markeredgecolor=colors[i])
            plt.xlim([-15, 10])
            plt.ylim([-15, 10])
    else:
        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)')
        for i in range(len(x_values)):
            x1 = [x[0] for x in x_values[i]]
            x0 = [x[1] for x in x_values[i]]
            ax.plot(x0, x1, function_values[i], markersize=3, marker='x',
                color='dimgrey', markeredgecolor=colors[i])
    if legend is not None:
        plt.legend(legend)
    plt.savefig(filepath)
    plt.show()
```

### 1.3 Mini-batch stochastic gradient descent

```
import math
from copy import deepcopy
import numpy as np

class StochasticGradientDescent:
    # Constant values for stochastic gradient descent
    epsilon = 1e-8
    STEP_CONSTANT = 0
    STEP_POLYAK = 1
    STEP_RMSPROP = 2
    STEP_HEAVYBALL = 3
    STEP_ADAM = 4

    def __init__(self, function_being_minimized, derivative_functions,
                  x_starting_value, step_size_choice,
                  hyperparameters_dict, batch_size, training_data):
        # Parameters for stochastic gradient descent
        self.function_being_minimized = function_being_minimized
        self.derivative_functions = derivative_functions
        self.number_of_parameters = len(x_starting_value)
        self.x_starting_value = deepcopy(x_starting_value)
        self.hyperparameters_dict = hyperparameters_dict
        # Parameters for minibatch stochastic gradient descent
        self.batch_size = batch_size
        self.training_data = training_data
        # Data for plotting
        self.logs = {
            'x_value': [deepcopy(self.x_starting_value)],
            'function_value':
                [self.function_being_minimized(self.x_starting_value,
                                                self.training_data)],
            'step': []
        }
        # Step size function
        self.iteration_function = self.iteration_function(step_size_choice)
        self.vars = None
        self.initialize_variables_for_function(step_size_choice)

    def compute_approximate_derivative(self, i, sample):
        return sum(
            self.derivative_functions[i](*self.x_starting_value,
                                         *sample[batch_size])
            for batch_size in range(self.batch_size)) / self.batch_size

    def iteration_function(self, step_size_choice):
        return {
            self.STEP_POLYAK: self.polyak_step_size,
            self.STEP_RMSPROP: self.rmsprop_step_size,
            self.STEP_HEAVYBALL: self.heavy_ball_step_size,
            self.STEP_ADAM: self.adam_step_size,
            self.STEP_CONSTANT: self.constant_step_size,
        }.get(step_size_choice, None)
```

```
def initialize_variables_for_function(self, algo):
    if algo == self.STEP_ADAM:
        self.logs['step'] = [[0] * self.number_of_parameters]
        self.vars = {
            'ms': [0] * self.number_of_parameters,
            'vs': [0] * self.number_of_parameters,
            'step': [0] * self.number_of_parameters,
            't': 0
        }
    elif algo == self.STEP_HEAVYBALL:
        self.logs['step'] = [0]
        self.vars = {
            'z': 0
        }
    elif algo == self.STEP_RMSPROP:
        self.logs['step'] = [[self.hyperparameters_dict['alpha0']] *
                               self.number_of_parameters]
        self.vars = {
            'sums': [0] * self.number_of_parameters,
            'alphas': [self.hyperparameters_dict['alpha0']] *
                       self.number_of_parameters
        }

def minibatch_iteration(self):
    np.random.shuffle(self.training_data)
    for starting_index_of_minibatch in range(0,
        len(self.training_data), self.batch_size):
        if starting_index_of_minibatch + self.batch_size >
            len(self.training_data):
            continue
        training_data_sample =
            self.training_data[starting_index_of_minibatch:
                (starting_index_of_minibatch + self.batch_size)]
        self.iteration_function(training_data_sample)
        self.logs['function_value'].append(self.function_being_minimized(
            self.x_starting_value, self.training_data))
        self.logs['x_value'].append(deepcopy(self.x_starting_value))

def polyak_step_size(self, sample):
    sum_of_squares_of_gradients = 0
    for iteration in range(self.number_of_parameters):
        sum_of_squares_of_gradients +=
            self.compute_approximate_derivative(iteration, sample) ** 2
        step_size = (self.function_being_minimized(self.x_starting_value,
            sample) / (sum_of_squares_of_gradients + self.epsilon))
    for iteration in range(self.number_of_parameters):
        self.x_starting_value[iteration] -= step_size *
            self.compute_approximate_derivative(iteration, sample)
        self.logs['step'].append(step_size)

def rmsprop_step_size(self, sample):
    a_0 = self.hyperparameters_dict['alpha0']
    b = self.hyperparameters_dict['beta']
    sums = self.vars['sums']
```

```

        alphas = self.vars['alphas']
        for iteration in range(self.number_of_parameters):
            self.x_starting_value[iteration] -= (alphas[iteration] *
            self.compute_approximate_derivative(iteration, sample))
            sums[iteration] = (b * sums[iteration]) + ((1 - b) *
            (self.compute_approximate_derivative(iteration, sample) ** 2))
            alphas[iteration] = a_0 / ((math.sqrt(sums[iteration])) +
            self.epsilon)
        self.logs['step'].append(deepcopy(alphas))

    def heavy_ball_step_size(self, sample):
        a = self.hyperparameters_dict['alpha']
        b = self.hyperparameters_dict['beta']
        historical_sum_of_square_gradients = self.vars['z']
        # Calculate sum of the squares of the gradients (partial
derivatives)
        sum_of_squares_of_gradients = 0
        for iteration in range(self.number_of_parameters):
            sum_of_squares_of_gradients +=
            self.compute_approximate_derivative(iteration, sample) ** 2
        # Calculate the step size
        historical_sum_of_square_gradients = (b *
        historical_sum_of_square_gradients) + (
            a * self.function_being_minimized(self.x_starting_value,
            sample) /
            (sum_of_squares_of_gradients + self.epsilon))

        for i in range(self.number_of_parameters):
            self.x_starting_value[i] -=
            (historical_sum_of_square_gradients *
            self.compute_approximate_derivative(i, sample))
        self.vars['z'] = historical_sum_of_square_gradients
        self.logs['step'].append(historical_sum_of_square_gradients)

    def adam_step_size(self, sample):
        a = self.hyperparameters_dict['alpha']
        b1 = self.hyperparameters_dict['beta1']
        b2 = self.hyperparameters_dict['beta2']
        ms = self.vars['ms']
        vs = self.vars['vs']
        step = self.vars['step']
        iteration_count = self.vars['t']
        iteration_count += 1
        for iteration in range(self.number_of_parameters):
            ms[iteration] = (b1 * ms[iteration]) + ((1 - b1) *
            self.compute_approximate_derivative(iteration, sample))
            vs[iteration] = (b2 * vs[iteration]) + ((1 - b2) *
            (self.compute_approximate_derivative(iteration, sample) ** 2))
            m_hat = ms[iteration] / (1 - (b1 ** iteration_count))
            v_hat = vs[iteration] / (1 - (b2 ** iteration_count))
            step[iteration] = a * (m_hat / ((v_hat ** 0.5) + self.epsilon))
            self.x_starting_value[iteration] -= step[iteration]
        self.vars['t'] = iteration_count
        self.logs['step'].append(deepcopy(step))

```



```

def constant_step_size(self, sample):
    a = self.hyperparameters_dict['alpha']
    self.logs['step'].append(a)
    for i in range(self.number_of_parameters):
        self.x_starting_value[i] -= a *
        self.compute_approximate_derivative(i, sample)

```

## 1.4 Question (a)

```

import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
from loss_function import generate_trainingdata, f

def part_a_ii():
    # Ranges -15 <= x_0 <= 10 and -15 <= x_1 <= 10
    X = np.linspace(-15, 10, 100)
    Y = np.linspace(-15, 10, 100)
    Z = []
    T = generate_trainingdata()
    for x_val in X:
        z = []
        for y_val in Y:
            z.append(f([x_val, y_val], T))
        Z.append(z)
    _, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5),
                                subplot_kw=dict(projection='3d'), gridspec_kw={'wspace': 0.2})
    X, Y = np.meshgrid(X, Y)
    Z = np.array(Z)
    ax1.set_title('Wireframe plot of f(x, N) for N=T')
    ax1.plot_wireframe(X, Y, Z, rstride=5, cstride=5)
    ax1.set_zlabel('f(x, T)')
    ax1.set_ylabel('x_1')
    ax1.set_xlabel('x_0')
    ax2.set_title('Contour plot of f(x, N) for N=T')
    ax2.contour3D(X, Y, Z, 60, cmap='plasma')
    ax2.set_zlabel('f(x, T)')
    ax2.set_ylabel('x_1')
    ax2.set_xlabel('x_0')
    plt.savefig("plots/part_a/ii.png")
    plt.show()

def part_a_iii():
    x0, x1, w0, w1 = sp.symbols('x0 x1 w0 w1', real=True)
    function_to_differentiate = sp.Min(18*(((x0-w0-1)**2)+((x1-w1-1)**2)),
    (((x0-w0-1)+10)**2)+(((x1-w1-1)+5)**2))
    differentiate_with_respect_to_x0 = sp.diff(function_to_differentiate,
    x0)
    differentiate_with_respect_to_x1 = sp.diff(function_to_differentiate,
    x1)
    return f, differentiate_with_respect_to_x0,
    differentiate_with_respect_to_x1

```

```

if __name__ == '__main__':
    part_a_ii()
    function_to_differentiate, differentiate_with_respect_to_x0,
    differentiate_with_respect_to_x1 = part_a_iii()
    print(differentiate_with_respect_to_x0)
    print(differentiate_with_respect_to_x1)

```

## 1.5 Question (b)

```

from copy import deepcopy
import numpy as np
from matplotlib import pyplot as plt
from loss_function import generate_trainingdata, f
from plot_utility import generate_contour_plot, colors
from stochastic_gradient_descent import StochasticGradientDescent

def part_b_i(function, derivative_functions):
    alphas = [0.1, 0.01, 0.001, 0.0001]
    x_values = []
    function_values = []
    iterations = 100
    iterations_as_list = list(range(iterations + 1))
    T = generate_trainingdata()
    for iteration, alpha in enumerate(alphas):
        stochastic_gradient_descent = StochasticGradientDescent(function,
            derivative_functions, [3, 3],
            StochasticGradientDescent.STEP_CONSTANT, {'alpha': alpha}, len(T),
            T)
        # We are using a batch size of T, so this is not a mini-batch
            iteration in reality
        for _ in range(iterations):
            stochastic_gradient_descent.minibatch_iteration()
            plt.plot(iterations_as_list,
                stochastic_gradient_descent.logs['function_value'],
                color=colors[iteration],
                label=f'alpha={alpha}')
            function_values.append(deepcopy(stochastic_gradient_descent.logs[
                'function_value']))
            x_values.append(deepcopy(stochastic_gradient_descent.logs[
                'x_value']))
        plt.ylabel('f(x, T)')
        plt.ylim([0, 150])
        plt.xlabel('Iterations')
        plt.legend()
        plt.savefig("plots/part_b/i-plot.png")
        plt.show()
        generate_contour_plot("plots/part_b/i-contour.png", function, T,
            x_values, function_values, three_dimensional=False,
            legend=[f'alpha={a}' for a in alphas])

```

```
def part_b_ii(function, derivative_functions):
    alpha = 0.1
    x_values = []
    function_values = []
    attempts = 5
    iterations = 10
    iterations_as_list = list(range(iterations + 1))
    T = generate_trainingdata()
    for attempt in range(attempts):
        stochastic_gradient_descent = StochasticGradientDescent(function,
            derivative_functions, [3, 3],
            StochasticGradientDescent.STEP_CONSTANT, {'alpha': alpha}, 5, T)
        for _ in range(iterations):
            stochastic_gradient_descent.minibatch_iteration()
        plt.plot(iterations_as_list,
            stochastic_gradient_descent.logs['function-value'],
            color=colors[attempt],
            label=f'Attempt {attempt+1}')
        function_values.append(deepcopy(stochastic_gradient_descent.logs[
            'function-value']))
        x_values.append(deepcopy(stochastic_gradient_descent.logs[
            'x-value']))
    plt.ylabel('f(x, T)')
    plt.ylim([0, 20])
    plt.xlabel('Iterations')
    plt.legend()
    plt.savefig("plots/part_b/ii-plot.png")
    plt.show()
    generate_contour_plot("plots/part_b/ii-contour.png", function, T,
        x_values, function_values, three_dimensional=False, legend=[f'Attempt
        {attempt+1}' for attempt in range(attempts)])

def part_b_iii(function, derivative_functions):
    batch_sizes = [1, 3, 5, 10, 25]
    alpha = 0.1
    x_values = []
    function_values = []
    iterations = 25
    iterations_as_list = list(range(iterations + 1))
    T = generate_trainingdata()
    for batch_iteration, batch_size in enumerate(batch_sizes):
        stochastic_gradient_descent = StochasticGradientDescent(function,
            derivative_functions, [3, 3],
            StochasticGradientDescent.STEP_CONSTANT, {'alpha': alpha},
            batch_size, T)
        for _ in range(iterations):
            stochastic_gradient_descent.minibatch_iteration()
        plt.plot(iterations_as_list,
            stochastic_gradient_descent.logs['function-value'],
            color=colors[batch_iteration],
            label=f'batch_size={batch_size}')
        function_values.append(deepcopy(stochastic_gradient_descent.logs[
            'function-value']))
```

```

        x_values.append(deepcopy(stochastic_gradient_descent.logs[ '
        x_value' ]))
plt.ylabel('f(x, T)')
plt.ylim([0, 3])
plt.xlabel('Iterations')
plt.legend()
plt.savefig("plots/part_b/iii-plot.png")
plt.show()
generate_contour_plot("plots/part_b/iii-contour.png", function, T,
x_values, function_values, three_dimensional=False,
legend=[f'batch_size={batch_size}' for batch_size in batch_sizes])

def part_b_iv(function, derivative_functions):
    alphas = [0.1, 0.01, 0.001, 0.0001]
    x_values = []
    function_values = []
    iterations = 25
    iterations_as_list = list(range(iterations + 1))
    T = generate_trainingdata()
    for alpha_iteration, alpha in enumerate(alphas):
        stochastic_gradient_descent = StochasticGradientDescent(function,
        derivative_functions, [3, 3],
        StochasticGradientDescent.STEP_CONSTANT, {'alpha': alpha}, 5, T)
        for _ in range(iterations):
            stochastic_gradient_descent.minibatch_iteration()
            plt.plot(iterations_as_list,
            stochastic_gradient_descent.logs['function_value'],
            color=colors[alpha_iteration], label=f'alpha={alpha}')
            function_values.append(deepcopy(stochastic_gradient_descent.logs[ '
            function_value' ]))
            x_values.append(deepcopy(stochastic_gradient_descent.logs[ '
            x_value' ]))
        plt.ylabel('f(x, T)')
        plt.ylim([0, 120])
        plt.xlabel('Iterations')
        plt.legend()
        plt.savefig("plots/part_b/iv-plot.png")
        plt.show()
        generate_contour_plot("plots/part_b/iv-contour.png", function, T,
        x_values, function_values,
        three_dimensional=False,
        legend=[f'alpha={alpha}' for alpha in alphas])

if __name__ == '__main__':
    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 + x1 + 4) ** 2, 0) + \
        (-2 * w0 + 2 * x0 + 18) * np.heaviside(
            18 * (-w0 + x0 - 1) ** 2 - (-w0 + x0 + 9) ** 2 + 18 * (-w1 +
            x1 - 1) ** 2 - (-w1 + x1 + 4) ** 2, 0)
    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 + x1 + 4) ** 2, 0) + \
(-2 * w1 + 2 * x1 + 8) * np.heaviside(
18 * (-w0 + x0 - 1) ** 2 - (-w0 + x0 + 9) ** 2 + 18 * (-w1 +
x1 - 1) ** 2 - (-w1 + x1 + 4) ** 2, 0)
part_b_i(f, [df0, df1])
part_b_ii(f, [df0, df1])
part_b_iii(f, [df0, df1])
part_b_iv(f, [df0, df1])

```

## 1.6 Question (c)

```

from copy import deepcopy
import numpy as np
from matplotlib import pyplot as plt
from loss_function import generate_trainingdata, f
from plot_utility import colors, generate_contour_plot
from stochastic_gradient_descent import StochasticGradientDescent

def part_c_i(function, derivative_functions):
    index_of_color = 0
    x_values = []
    function_values = []
    iterations = 100
    iterations_as_list = list(range(iterations + 1))
    T = generate_trainingdata()
    # Stochastic gradient descent as a baseline for first iteration
    labels = ['Baseline']
    stochastic_gradient_descent_baseline =
    StochasticGradientDescent(function, derivative_functions, [3, 3],
    StochasticGradientDescent.STEP_CONSTANT, {'alpha': 0.1}, 5, T)
    for _ in range(iterations):
        stochastic_gradient_descent_baseline.minibatch_iteration()
    plt.plot(iterations_as_list,
    stochastic_gradient_descent_baseline.logs['function-value'],
    label=labels[0],
            color=colors[index_of_color])
    function_values.append(deepcopy(stochastic_gradient_descent_baseline
    .logs['function-value']))
    x_values.append(deepcopy(stochastic_gradient_descent_baseline.logs['
    x-value']))
    index_of_color += 1
    # Stochastic gradient descent using polyak step size for remaining
    iterations
    batch_sizes = [1, 3, 5, 10, 25]
    for batch_size in batch_sizes:
        stochastic_gradient_descent = StochasticGradientDescent(function,
        derivative_functions, [3, 3],
        StochasticGradientDescent.STEP_POLYAK, {}, batch_size, T)
        for _ in range(iterations):
            stochastic_gradient_descent.minibatch_iteration()
        labels.append(f'batch_size={batch_size}')

```



```

plt.plot(iterations_as_list ,
stochastic_gradient_descent.logs['function_value'],
label=labels[-1],
        color=colors[index_of_color])
function_values.append(deepcopy(stochastic_gradient_descent
.logs['function_value']))
x_values.append(deepcopy(stochastic_gradient_descent.logs['
x_value']))
index_of_color += 1
plt.ylabel('f(x, T)')
plt.ylim([0, 60])
plt.xlabel('Iterations')
plt.legend()
plt.savefig("plots/part_c/i-plot.png")
plt.show()
generate_contour_plot("plots/part_c/i-contour.png", function, T,
x_values, function_values, three_dimensional=False,
                    legend=labels)

def part_c_ii(function, derivative_functions):
    index_of_color = 0
    x_values = []
    function_values = []
    iterations = 100
    iterations_as_list = list(range(iterations + 1))
    T = generate_trainingdata()
    # Stochastic gradient descent as a baseline for first iteration
    labels = ['Baseline']
    stochastic_gradient_descent_baseline =
    StochasticGradientDescent(function, derivative_functions, [3, 3],
    StochasticGradientDescent.STEP_CONSTANT, {'alpha': 0.1}, 5, T)
    for _ in range(iterations):
        stochastic_gradient_descent_baseline.minibatch_iteration()
    plt.plot(iterations_as_list ,
stochastic_gradient_descent_baseline.logs['function_value'],
label=labels[0],
        color=colors[index_of_color])
    function_values.append(deepcopy(stochastic_gradient_descent_baseline
.logs['function_value']))
    x_values.append(deepcopy(stochastic_gradient_descent_baseline.logs['
x_value']))
    index_of_color += 1

    # Stochastic gradient descent using RMSProp step size for remaining
    iterations to find best hyperparameters
    list_of_alpha_0 = [0.1, 0.01, 0.001]
    list_of_beta = [0.25, 0.9]
    for a0 in list_of_alpha_0:
        for b in list_of_beta:
            stochastic_gradient_descent =
            StochasticGradientDescent(function, derivative_functions, [3,
            3], StochasticGradientDescent.STEP_RMSPROP, {'alpha0': a0,
            'beta': b}, 5, T)
            for _ in range(iterations):

```

```

        stochastic_gradient_descent.minibatch_iteration()
    labels.append(f'alpha_0={a0}, beta={b}')
    plt.plot(iterations_as_list,
             stochastic_gradient_descent.logs['function_value'],
             label=labels[-1],
             color=colors[index_of_color])
    function_values.append(deepcopy(stochastic_gradient_descent
                                   .logs['function_value']))
    x_values.append(deepcopy(stochastic_gradient_descent.logs['
    x_value']))
    index_of_color += 1
plt.ylabel('f(x, T)')
plt.ylim([0, 60])
plt.xlabel('Iterations')
plt.legend()
plt.savefig("plots/part_c/ii-plot.png")
plt.show()

# Stochastic gradient descent using RMSProp step size using the best
hyperparameters to explore various batch sizes
index_of_color = 0
x_values = []
function_values = []
labels = ['Baseline']
plt.plot(iterations_as_list,
         stochastic_gradient_descent_baseline.logs['function_value'],
         label=labels[0], color=colors[index_of_color])
function_values.append(deepcopy(stochastic_gradient_descent_baseline
                                .logs['function_value']))
x_values.append(deepcopy(stochastic_gradient_descent_baseline.logs['
x_value']))
index_of_color += 1
batch_sizes = [1, 3, 5, 10, 25]
for batch_size in batch_sizes:
    sgd = StochasticGradientDescent(function, derivative_functions,
    [3, 3], StochasticGradientDescent.STEP_RMSPROP, {'alpha0': 0.1,
    'beta': 0.9}, 5, T)
    for _ in range(iterations):
        sgd.minibatch_iteration()
        labels.append(f'batch_size={batch_size}')
        plt.plot(iterations_as_list, sgd.logs['function_value'],
                 label=labels[-1], color=colors[index_of_color])
        index_of_color += 1
        x_values.append(deepcopy(sgd.logs['x_value']))
        function_values.append(deepcopy(sgd.logs['function_value']))
plt.ylabel('f(x, T)')
plt.ylim([0, 10])
plt.xlabel('Iterations')
plt.legend()
plt.savefig("plots/part_c/ii-plot-2.png")
plt.show()
generate_contour_plot("plots/part_c/ii-contour.png", function, T,
x_values, function_values,
                    three_dimensional=False, legend=labels)

```

```
def part_c_iii(function, derivative_functions):
    index_of_color = 0
    x_values = []
    function_values = []
    iterations = 100
    iterations_as_list = list(range(iterations + 1))
    T = generate_trainingdata()
    # Stochastic gradient descent as a baseline for first iteration
    labels = ['Baseline']
    stochastic_gradient_descent_baseline =
    StochasticGradientDescent(function, derivative_functions, [3, 3],
    StochasticGradientDescent.STEP_CONSTANT, {'alpha': 0.1}, 5, T)
    for _ in range(iterations):
        stochastic_gradient_descent_baseline.minibatch_iteration()
    plt.plot(iterations_as_list,
    stochastic_gradient_descent_baseline.logs['function-value'],
    label=labels[0],
        color=colors[index_of_color])
    function_values.append(deepcopy(stochastic_gradient_descent_baseline
    .logs['function-value']))
    x_values.append(deepcopy(stochastic_gradient_descent_baseline.logs['
    x-value']))
    index_of_color += 1

    # Stochastic gradient descent using Heavyball step size to find best
    hyperparameters
    alphas = [0.01, 0.001]
    betas = [0.25, 0.5, 0.9]
    for alpha in alphas:
        for beta in betas:
            stochastic_gradient_descent =
            StochasticGradientDescent(function, derivative_functions, [3,
            3], StochasticGradientDescent.STEP_HEAVYBALL, {'alpha': alpha,
            'beta': beta}, 5, T)
            for _ in range(iterations):
                stochastic_gradient_descent.minibatch_iteration()
            labels.append(f'alpha={alpha}, beta={beta}')
            plt.plot(iterations_as_list,
            stochastic_gradient_descent.logs['function-value'],
            label=labels[-1],
                color=colors[index_of_color])
            function_values.append(deepcopy(stochastic_gradient_descent
            .logs['function-value']))
            x_values.append(deepcopy(stochastic_gradient_descent.logs['
            x-value']))
            index_of_color += 1

    plt.ylim([0, 60])
    plt.ylabel('f(x, T)')
    plt.xlabel('Iterations')
    plt.legend()
    plt.savefig("plots/part_c/iii-plot.png")
    plt.show()
```

```

# Stochastic gradient descent using Heavyball step size with the best hyperparameters to explore various batch sizes
index_of_color = 0
x_values = []
function_values = []
labels = ['Baseline']
plt.plot(iterations_as_list,
stochastic_gradient_descent_baseline.logs['function_value'],
label=labels[0],
        color=colors[index_of_color])
index_of_color += 1
x_values.append(deepcopy(stochastic_gradient_descent_baseline.logs['x_value']))
function_values.append(deepcopy(stochastic_gradient_descent_baseline.logs['function_value']))
batch_sizes = [1, 3, 5, 10, 25]
for batch_size in batch_sizes:
    stochastic_gradient_descent = StochasticGradientDescent(function,
derivative_functions, [3, 3],
StochasticGradientDescent.STEP_HEAVYBALL, {'alpha': 0.01, 'beta':
0.9}, 5, T)
    for _ in range(iterations):
        stochastic_gradient_descent.minibatch_iteration()
        labels.append(f'batch_size={batch_size}')
        plt.plot(iterations_as_list,
stochastic_gradient_descent.logs['function_value'],
label=labels[-1],
                color=colors[index_of_color])
        function_values.append(deepcopy(stochastic_gradient_descent.logs['function_value']))
        x_values.append(deepcopy(stochastic_gradient_descent.logs['x_value']))
        index_of_color += 1
plt.ylabel('f(x, T)')
plt.ylim([0, 10])
plt.xlabel('Iterations')
plt.legend()
plt.show()
plt.savefig("plots/part_c/iii-plot-2.png")
generate_contour_plot("plots/part_c/iii-contour.png", function, T,
x_values, function_values,
                        three_dimensional=False, legend=labels)

def part_c_iv(function, derivative_functions):
    index_of_color = 0
    x_values = []
    function_values = []
    iterations = 100
    iterations_as_list = list(range(iterations + 1))
    T = generate_trainingdata()
    # Stochastic gradient descent as a baseline for first iteration
    labels = ['Baseline']

```

```
stochastic_gradient_descent_baseline =
StochasticGradientDescent(function, derivative_functions, [3, 3],
StochasticGradientDescent.STEP_CONSTANT, {'alpha': 0.1}, 5, T)
for _ in range(iterations):
    stochastic_gradient_descent_baseline.minibatch_iteration()
plt.plot(iterations_as_list,
stochastic_gradient_descent_baseline.logs['function_value'],
label=labels[0],
        color=colors[index_of_color])
function_values.append(deepcopy(stochastic_gradient_descent_baseline
.logs['function_value']))
x_values.append(deepcopy(stochastic_gradient_descent_baseline.logs['
x_value']))
index_of_color += 1

# Stochastic gradient descent using Adam step size to find best
hyperparameters
list_of_alphas = [10, 1, 0.1]
list_of_beta_1 = [0.25, 0.9]
list_of_beta_2 = [0.999]
for a in list_of_alphas:
    for b1 in list_of_beta_1:
        for b2 in list_of_beta_2:
            stochastic_gradient_descent =
StochasticGradientDescent(function, derivative_functions,
[3, 3], StochasticGradientDescent.STEP_ADAM, {'alpha': a,
'beta1': b1, 'beta2': b2}, 5, T)
            for _ in range(iterations):
                stochastic_gradient_descent.minibatch_iteration()
            labels.append(
                f'alpha={a}, beta_1={b1}, beta_2={b2}'
            )
            plt.plot(iterations_as_list,
stochastic_gradient_descent.logs['function_value'],
label=labels[-1],
                color=colors[index_of_color])
            function_values.append(deepcopy(stochastic_gradient_descent
.logs['function_value']))
            x_values.append(deepcopy(stochastic_gradient_descent.logs['
x_value']))
            index_of_color += 1
plt.ylabel('f(x, T)')
plt.ylim([0, 60])
plt.xlabel('Iterations')
plt.legend()
plt.savefig("plots/part_c/iv-plot.png")
plt.show()

# Stochastic gradient descent using Adam step size with the best
hyperparameters to explore various batch sizes
batch_sizes = [1, 3, 5, 10, 25]
index_of_color = 0
x_values = []
function_values = []
labels = ['Baseline']
```



```

plt.plot(iterations_as_list ,
stochastic_gradient_descent_baseline.logs[ 'function_value' ],
label=labels[0],
        color=colors[index_of_color])
function_values.append(deepcopy(stochastic_gradient_descent_baseline
.logs[ 'function_value' ]))
x_values.append(deepcopy(stochastic_gradient_descent_baseline
.logs[ 'x_value' ]))
index_of_color += 1
for batch_size in batch_sizes:
    stochastic_gradient_descent = StochasticGradientDescent(function ,
    derivative_functions , [3, 3], StochasticGradientDescent.STEP_ADAM,
    { 'alpha': 10, 'beta1': 0.9, 'beta2': 0.999}, 5, T)
    for _ in range(iterations):
        stochastic_gradient_descent.minibatch_iteration()
    labels.append(f'batch_size={batch_size}')
    plt.plot(iterations_as_list ,
    stochastic_gradient_descent.logs[ 'function_value' ],
    label=labels[-1],
            color=colors[index_of_color])
    function_values.append(deepcopy(stochastic_gradient_descent.logs[ '
function_value' ]))
    x_values.append(deepcopy(stochastic_gradient_descent.logs[ '
x_value' ]))
    index_of_color += 1
plt.ylim([0, 10])
plt.ylabel('f(x, T)')
plt.xlabel('Iterations')
plt.legend()
plt.savefig("plots/part_c/iv-plot-2.png")
plt.show()
generate_contour_plot("plots/part_c/iv-contour.png", function , T,
x_values , function_values ,
                    three_dimensional=False , legend=labels)

if __name__ == '__main__':
    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 + x1 + 4) ** 2, 0) + \
        (-2 * w0 + 2 * x0 + 18) * np.heaviside(
            18 * (-w0 + x0 - 1) ** 2 - (-w0 + x0 + 9) ** 2 + 18 * (-w1 +
            x1 - 1) ** 2 - (-w1 + x1 + 4) ** 2, 0)
    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 + x1 + 4) ** 2, 0) + \
        (-2 * w1 + 2 * x1 + 8) * np.heaviside(
            18 * (-w0 + x0 - 1) ** 2 - (-w0 + x0 + 9) ** 2 + 18 * (-w1 +
            x1 - 1) ** 2 - (-w1 + x1 + 4) ** 2, 0)
    part_c_i(f, [df0, df1])
    part_c_ii(f,[df0, df1])
    part_c_iii(f, [df0, df1])
    part_c_iv(f, [df0, df1])

```