Payal Mohapatra
NetID : PMS4829
Student ID : 3334086 EE-435 – Homework 1

All the functions are that are called in the code snippet of each solution are defined in the appendix. Alternately, the entire code suite can be found here - **https://github.com/payalmohapatra/Deep-Learning-EE-435/tree/main/HW_1**

3.5 Try out gradient descent Run gradient descent to minimize the function 3.8 Exercises 73 g(w) = 1 50 ± w 4 + w 2 + 10w ² (3.44) with an initial point w 0 = 2 and 1000 iterations. Make three separate runs using each of the steplength values α = 1, α = 10 -1 , and α = 10 -2 .

Compute the derivative of this function by hand, and implement it (as well as the function itself) in Python using NumPy.

Plot the resulting cost function history plot of each run in a single figure to compare their performance. Which steplength value works best for this particular function and initial point?

Solution :

Derivative of the function:

3.5

$$g(w) = \frac{1}{50}\left(w^4 + w^2 + 10w\right)$$

$$g'(w) = \frac{1}{50}\left(4w^3 + 2w + 10\right)$$

Plot of the derivative of the function shown in Figure 1:

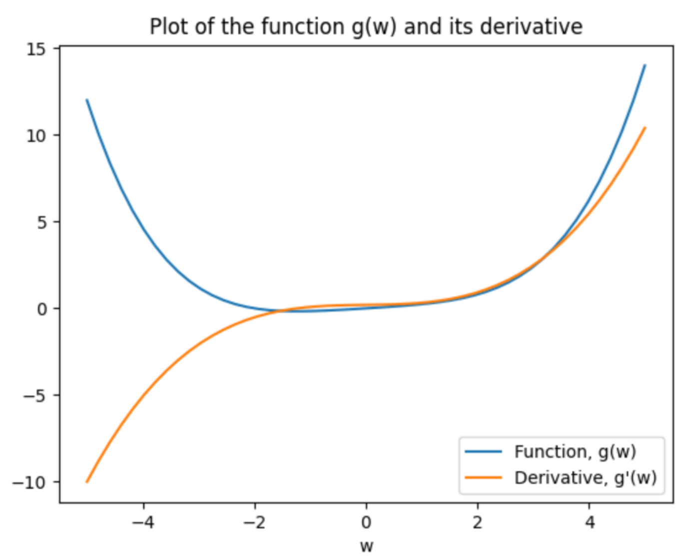

Figure 1 Plot of the function g(w) and its derivative for Q.3.5.

The below graph in Figure 2 shows the cost history using different learning rates (labeled).

- The cost function using 10^-2 learning rate is very slow and 1000 iterations are not enough to minimize it.
- The minimization using learning rates 1 and 10^-1 start to coincide after approximately 200 iterations. The final value of weights obtained using learning rates 1 and 10^-1 are the same after 1000 iteration. (.i.e. -1.2347, which are the roots of g'(w)).
- Although 10^-1 and 1 learning rates have similar results, performance wise alpha=1 is better since it converges faster in this case.

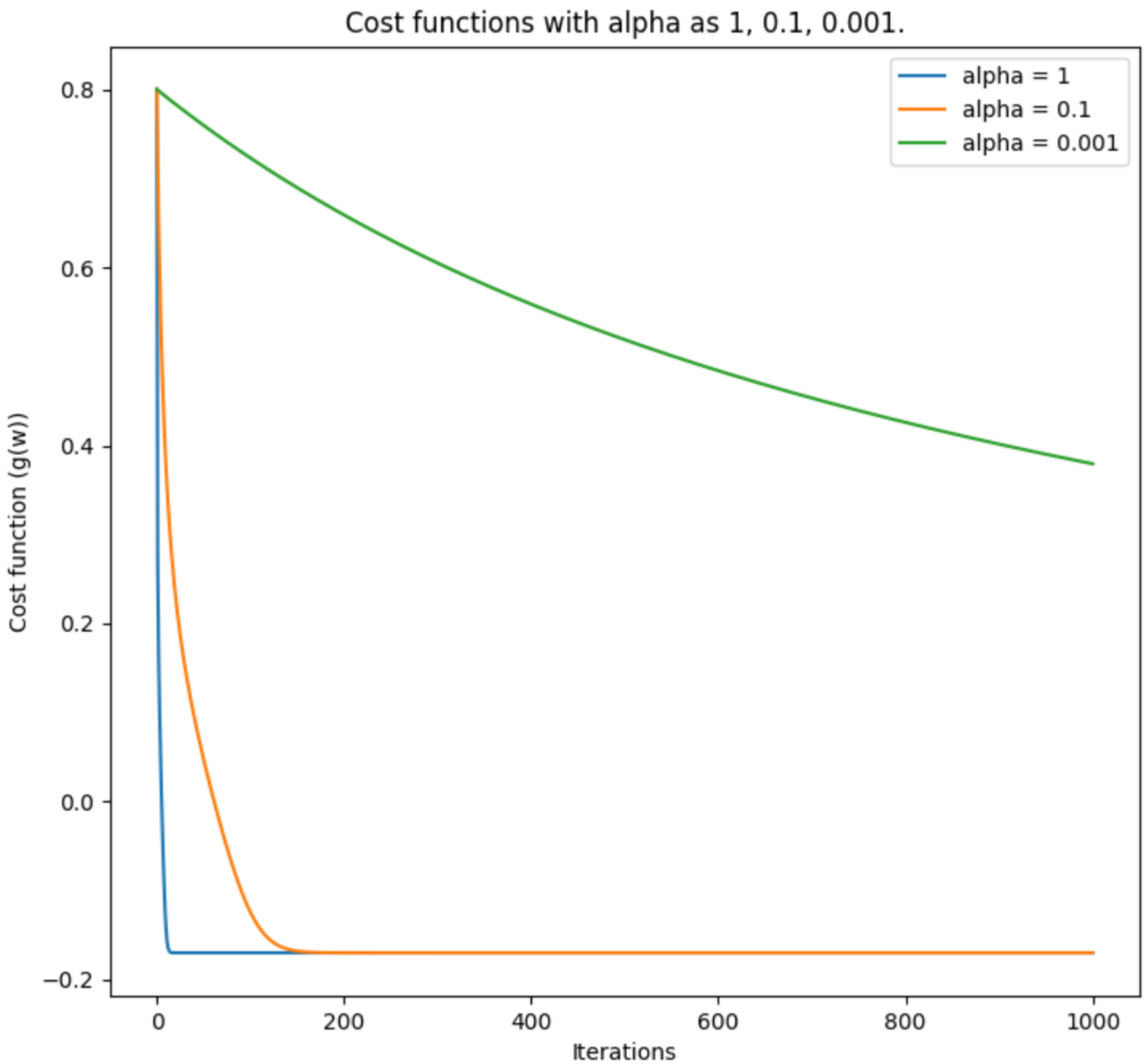

Figure 2 Cost functions with alpha as 1, 0.1, 0.001 for Q.3.5.

Code snippet for 3.5 implementation :

```
from numpy.core.fromnumeric import argmax
import autograd.numpy as np
import matplotlib.pyplot as plt
```

```
from scipy.stats import mode
from skimage import exposure
from sklearn.datasets import fetch_openml
from optimizers_only import gradient_descent

w = 2.0
## cost function
def model(w):
    g = (w**4 + w**2 + 10*w)*(1/50)
    return g

plt.figure(1)
plt.legend(["g(w)", "g'(w)"])

max_its = 1000
alpha_choice = 1
weight_history,cost_history = gradient_descent(model ,alpha_choice,max_its,w)
plt.plot(cost_history)

alpha_choice = 0.1
weight_history,cost_history = gradient_descent(model ,alpha_choice,max_its,w)
plt.plot(cost_history)

alpha_choice = 0.001
weight_history,cost_history = gradient_descent(model ,alpha_choice,max_its,w)
plt.plot(cost_history)

plt.xlabel('Iterations')
plt.ylabel('Cost function (g(w))')
plt.legend(["alpha = 1", "alpha = 0.1", "alpha = 0.001"])
plt.title('Cost functions with alpha as 1, 0.1, 0.001.')

### plotting g(w) and
f = lambda w: (w**4 + w**2 + 10.0*w)/50.0
deri = lambda w: (4*w**3 + 2*w + 10.0)/50.0
plt.figure(2)
xpts = np.linspace(-5, 5, 50)
plt.plot(xpts, f(xpts))
plt.plot(xpts, deri(xpts))
plt.legend(["Function, g(w)", "Derivative, g'(w)"])
plt.xlabel('w')
plt.title('Plot of the function g(w) and its derivative')
plt.show()
```

3.8 Tune fixed steplength for gradient descent. Take the cost function g (w) = wTw (3.45) where w is an N = 10 dimensional input vector, and g is convex with a single global minimum at w = 0N×1 . Code up gradient descent and run it for 100 steps using the initial point w0 = 10 · 1N×1 , with three steplength values: α1 = 0.001, α2 = 0.1, and α3 = 1. Produce a cost function history plot to compare the three runs and determine which performs best.

Solution:

The graph in **Error! Reference source not found.** below shows the cost history plots for different learning rates. The steplenght/learning rate 0.1 performs the best in this case.

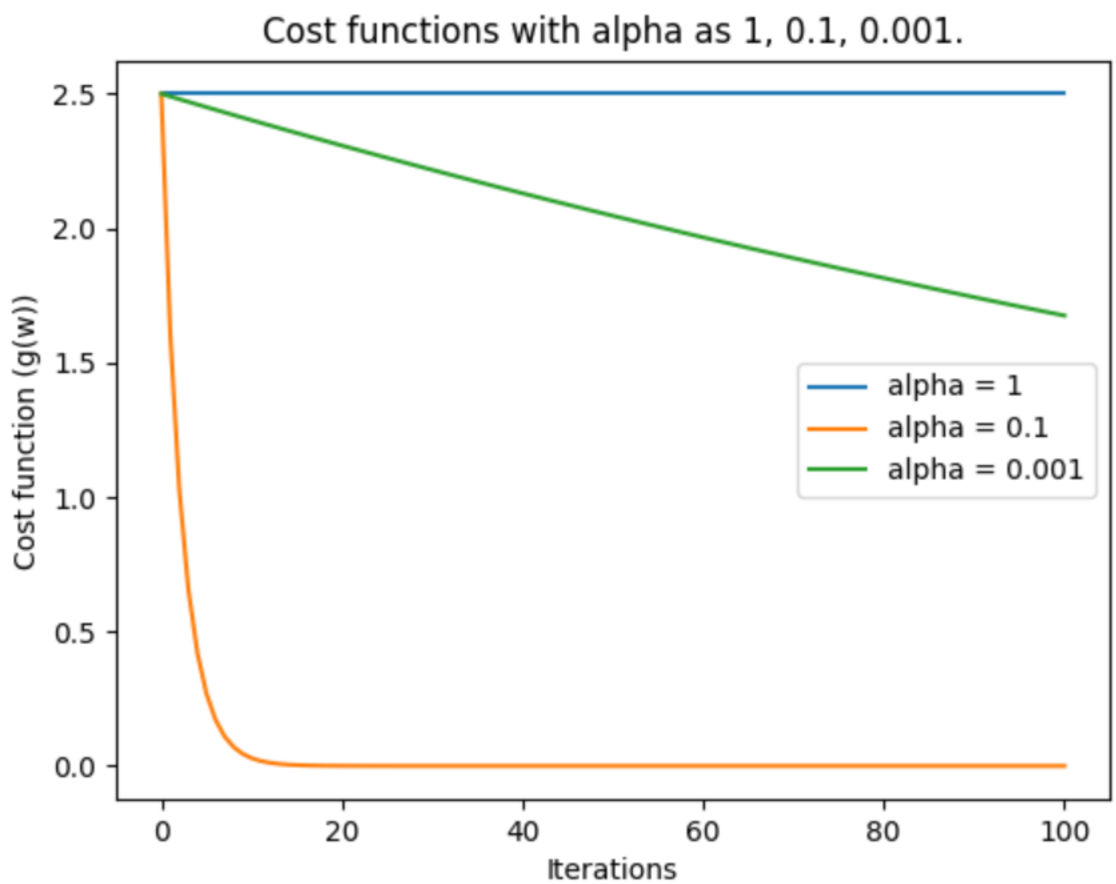

Figure 3 Cost functions with alpha as 1, 0.1, 0.001 for Q.3.8.

An additional observation for step-length 1, is that it does not change its value as per the plot. This is due to the fact that the cost function is quadratic and produces same results for both negative and positive values of the w. This is why we do not see any update in the cost function since the weights are toggling between $\pm$0.5 resulting in same cost function value.

Code snippet for 3.8 implementation:

```
from numpy.core.fromnumeric import argmax
import autograd.numpy as np
import matplotlib.pyplot as plt
from scipy.stats import mode
from skimage import exposure
from sklearn.datasets import fetch_openml
#from optimizers import gradient_descent
from optimizers_only import gradient_descent

w = np.zeros((10,1)) + 0.5
## cost function
def model(w):
    g = np.dot(w.T,w)
    return g

max_its = 1
alpha_choice = 1
weight_history_1,cost_history = gradient_descent(model ,alpha_choice,max_its,w)
plt.plot(cost_history)

alpha_choice = 0.1
weight_history_0_1,cost_history = gradient_descent(model ,alpha_choice,max_its,w)
plt.plot(cost_history)

alpha_choice = 0.001
```

```
weight_history_0_001,cost_history = gradient_descent(model ,alpha_choice,max_its,w)
plt.plot(cost_history)

plt.xlabel('Iterations')
plt.ylabel('Cost function (g(w))')
plt.legend(["alpha = 1", "alpha = 0.1", "alpha = 0.001"])
plt.title('Cost functions with alpha as 1, 0.1, 0.001.')
plt.show()

print(weight_history_0_001[-1])
print(weight_history_0_1[-1])
print(weight_history_1[-1])
```

3.9 Code up momentum-accelerated gradient descent Code up the momentum-accelerated gradient descent scheme described in Section A.2.2 and use it to repeat the experiments detailed in Example A.1 using a cost function history plot to come to the same conclusions drawn by studying the contour plots shown in Figure A.3.

Solution :

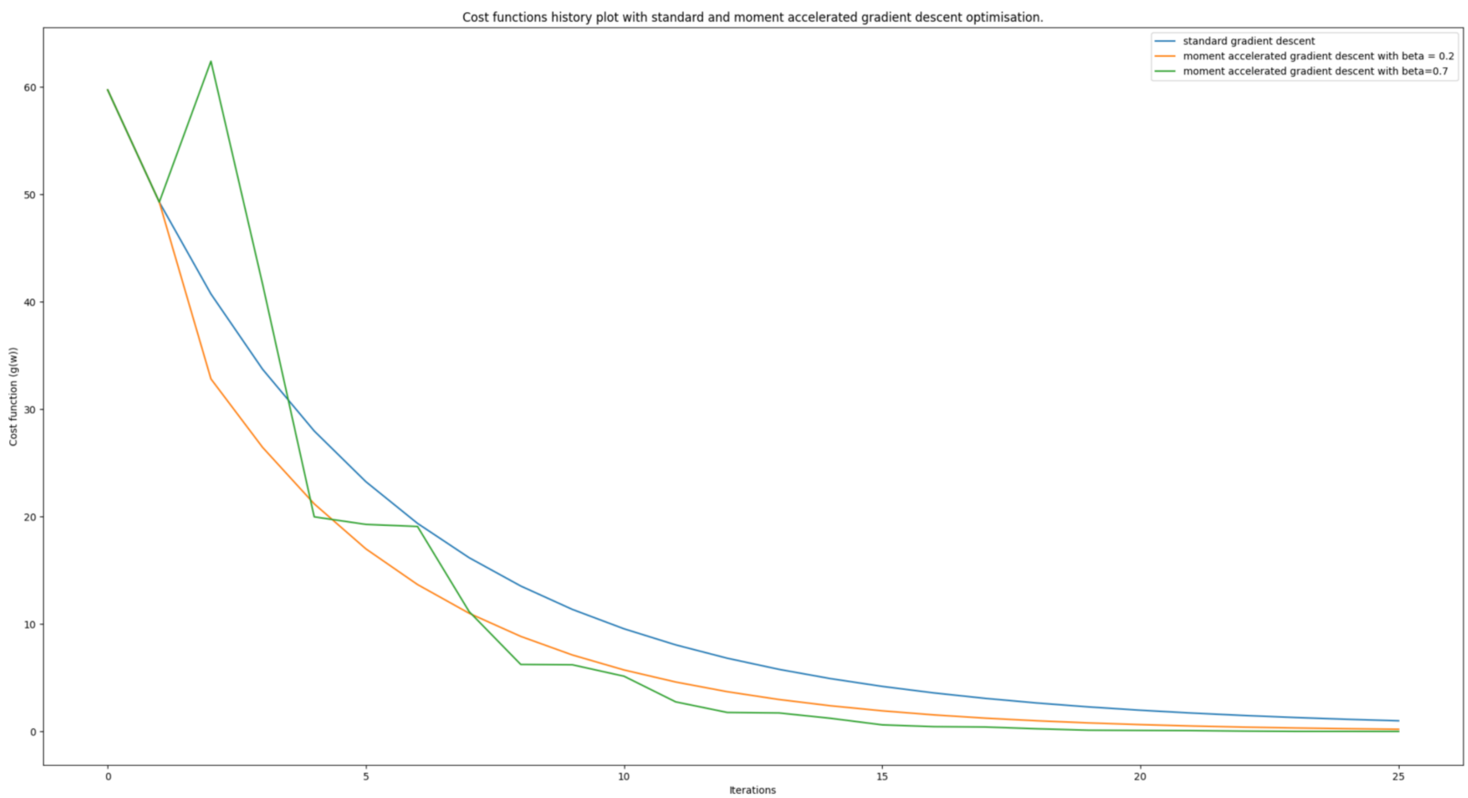

Figure 4 Cost functions history plot with standard and moment accelerated gradient descent optimization for Q.3.9.

Observation :

- The moment accelerated gradient descent indeed helps in arriving at the minima hence, optimizing the cost function faster. Seen from the green curve getting minimized the fastest.
- Additionally, we observe oscillations in the initial iteration for the moment accelerated optimizations with beta = 0.7. Below is the update rule we use, and we can see for beta > 0.5 it is more influenced my the moving average of

previous values. For the first few iterations, not enough sample points are resent to make a reasonable representation of the past points. Hence, we see the sharp jumps for beta=0.7 for initial iterations.

```
d_eval = beta*(d_eval_iter[-1]) + (1-beta)*(-grad_eval_iter[-1])
```

Code snippet for 3.9 implementation :

```
from numpy.core.fromnumeric import argmax
import autograd.numpy as np
import matplotlib.pyplot as plt
from scipy.stats import mode
#from optimizers import gradient_descent
from optimizers_only import gradient_descent
from optimizers_only import gradient_descent_momentum

w = np.array([[10.0], [1.0]])
C = np.array([[0.5, 0],[0, 9.75]])

## cost function
def model(w):
    g = np.dot(np.dot(w.T, C), w)
    return g
# g = lambda w: (w*w*w*w + w*w + 10.0*w)/50.0

plt.figure(1)
plt.legend(["g(w)", "g'(w)"])

max_its =25
alpha_choice = 0.1
weight_history,cost_history = gradient_descent(model ,alpha_choice,max_its,w)
plt.plot(cost_history)



## momentum accelerated
beta = 0.2
weight_history,cost_history = gradient_descent_momentum(model ,alpha_choice,max_its,w, beta)
plt.plot(cost_history)

beta = 0.7
weight_history,cost_history = gradient_descent_momentum(model ,alpha_choice,max_its,w, beta)
plt.plot(cost_history)

# alpha_choice = 0.1
# weight_history,cost_history = gradient_descent(model ,alpha_choice,max_its,w)
# plt.plot(cost_history)

# alpha_choice = 0.001
# weight_history,cost_history = gradient_descent(model ,alpha_choice,max_its,w)
# plt.plot(cost_history)
plt.xlabel('Iterations')
plt.ylabel('Cost function (g(w))')
plt.legend(["standard gradient descent","moment accelerated gradient descent with beta = 0.2","moment accelerated gradient descent with beta=0.7"])
plt.title('Cost functions history plot with standard and moment accelerated gradient descent optimisation.')
plt.show()
```

3.10 Slow-crawling behavior of gradient descent In this exercise you will compare the standard and fully normalized gradient descent schemes in minimizing the function g(w1, w2) = tanh(4 w1 + 4 w2) + max(1, 0.4 w 2 1) + 1. (3.46)

Using the initialization w0 = [2 2] T make a run of 1000 steps of the standard and fully normalized gradient descent schemes, using a steplength value of α = 10 -1 in both instances. Use a cost function history plot to compare the two runs, noting the progress made with each approach.

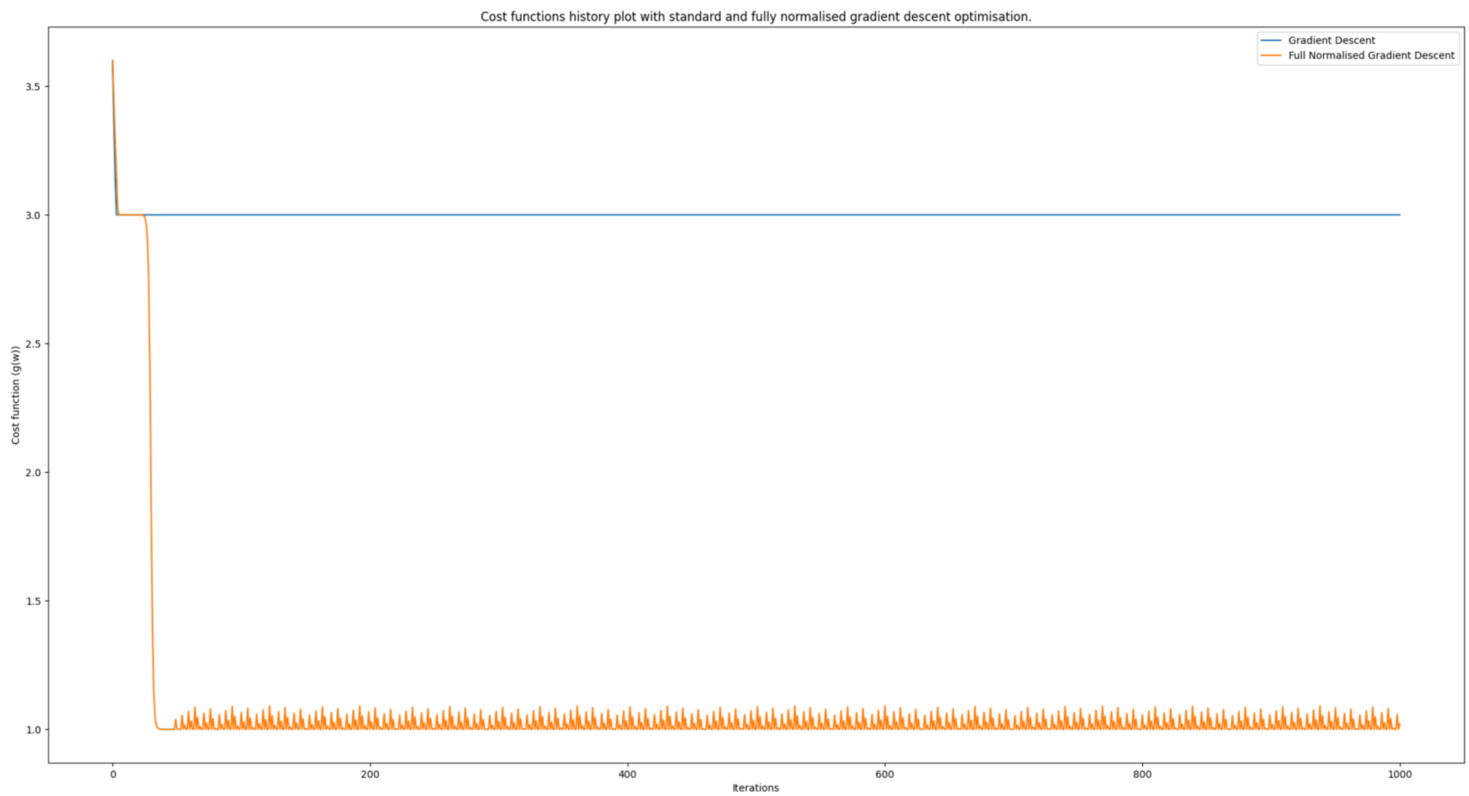

Figure 5 Cost functions history plot with standard and fully normalised gradient descent optimisation for Q.3.10.

Figure 5 shows the plots for the cost functions optimized using a standard and fully normalized gradient descent. We observe that the standard gradient descent is unable to minimize cost function as effectively possibly due to close proximity to a stationary point which diminishes the gradient hence, no more updates to our weight matrix. This is overcome by using the normalized gradient descent.

Code snippet for 3.10 :

```
from numpy.core.fromnumeric import argmax
import autograd.numpy as np
import matplotlib.pyplot as plt
from scipy.stats import mode
#from optimizers import gradient_descent
from optimizers_only import gradient_descent
from optimizers_only import gradient_descent_full_norm

w = np.array([[2.0],[2.0]])

## cost function
# g(w1, w2) = tanh(4 w1 + 4 w2) + max(1, 0.4 w21) + 1.
def model(w): # w is a (1,2) vector
    g = np.tanh(4.0*w[0] + 4.0*w[1]) + max(1.0, 0.4*w[0]*w[0]) + 1.0
    return g


max_its =1000
alpha_choice = 0.1
weight_history,cost_history = gradient_descent(model ,alpha_choice,max_its,w)
plt.plot(cost_history)


## full normalised
weight_history,cost_history = gradient_descent_full_norm(model ,alpha_choice,max_its,w)
plt.plot(cost_history)
```

```
plt.xlabel('Iterations')
plt.ylabel('Cost function (g(w))')
plt.legend(["Gradient Descent","Full Normalised Gradient Descent"])
plt.title('Cost functions history plot with standard and fully normalised gradient descent optimisation.')
plt.show()
```

3.11 Comparing normalized gradient descent schemes Code up the full and component-wise normalized gradient descent schemes and repeat the experiment described in Example A.4 using a cost function history plot to come to the same conclusions drawn by studying the plots shown in Figure A.6.

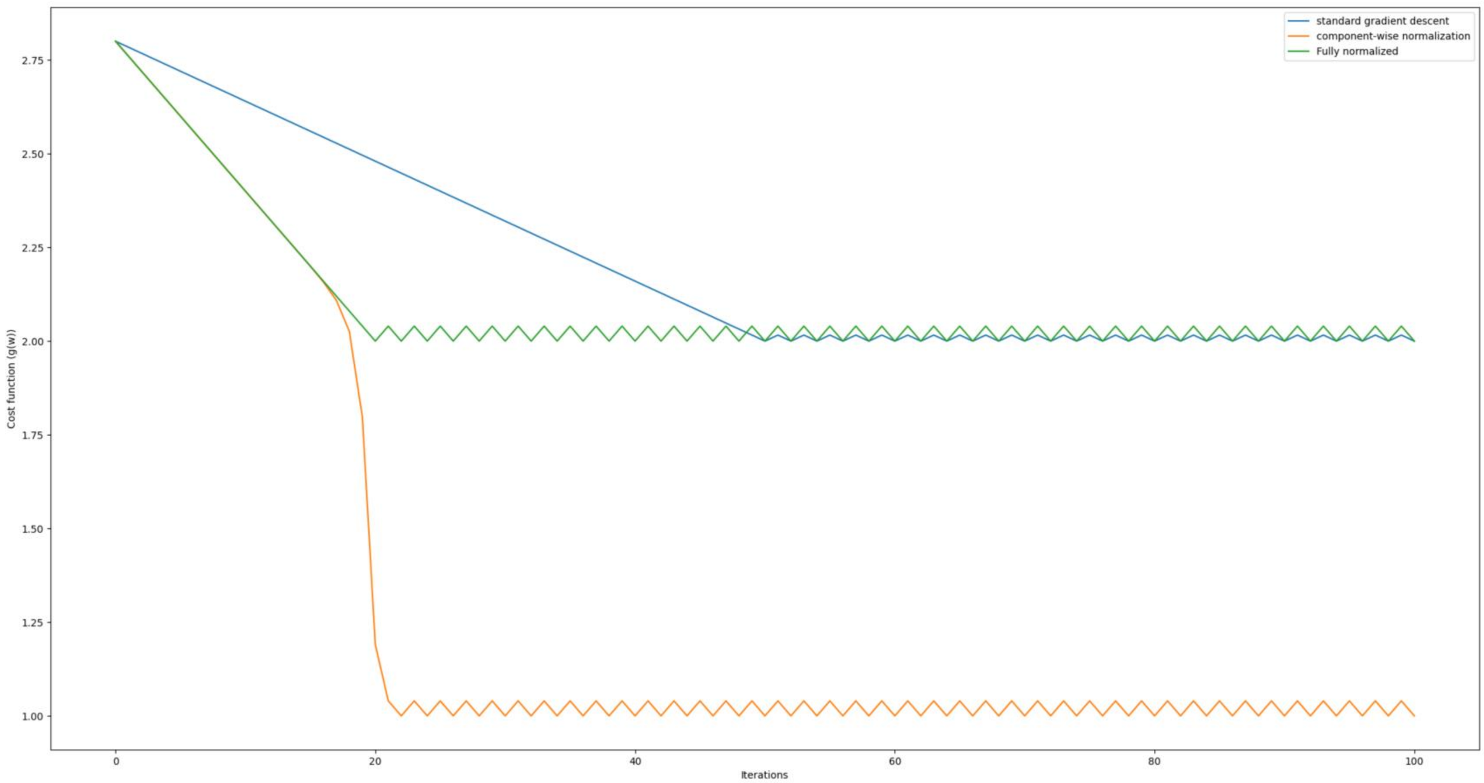

Figure 6 Comparison of cost function history using elementwise and full normalised optimisation methods for Q.3.11. Additonally plotted the cost history of standard gradient descent for comparison as well.

As shown in Figure 6, the component wise normalization outperforms the other optimizers in this case. Although, the standard gradient descent optimizes the cost to a similar extent as fully normalised gradient descent here, we can see that the fully normalized optimization achieves so faster.

Code Snippet for 3.11 :

```
from numpy.core.fromnumeric import argmax
#import mnist
import autograd.numpy as np
import matplotlib.pyplot as plt
from scipy.stats import mode
#from optimizers import gradient_descent
from optimizers_only import gradient_descent
from optimizers_only import gradient_descent_momentum
from optimizers_only import gradient_descent_component
from optimizers_only import gradient_descent_full_norm

w = np.array([[2.0], [2.0]])
```

```
C = np.array([[0.5, 0],[0, 9.75]])

## cost function
def model(w):
    g = max(0, np.tanh(4*w[0] + 4*w[1])) + np.abs(0.4*w[0]) + 1
    return g

plt.figure(1)
plt.legend(["g(w)", "g'(w)"])

max_its =100
alpha_choice = 0.1
weight_history,cost_history = gradient_descent(model ,alpha_choice,max_its,w)
plt.plot(cost_history)

weight_history,cost_history = gradient_descent_component(model ,alpha_choice,max_its,w)
plt.plot(cost_history)

weight_history,cost_history = gradient_descent_full_norm(model ,alpha_choice,max_its,w)
plt.plot(cost_history)

plt.xlabel('Iterations')
plt.ylabel('Cost function (g(w))')
plt.legend(["standard gradient descent","component-wise normalization","Fully normalized"])
plt.show()
```

Appendix

```
import autograd.numpy as np
from autograd import value_and_grad
from autograd import hessian
from autograd.misc.flatten import flatten_func

# random search function
def random_search(g,alpha_choice,max_its,w,num_samples):
    # run random search
    weight_history = []         # container for weight history
    cost_history = []           # container for corresponding cost function history
    alpha = 0
    for k in range(1,max_its+1):
        # check if diminishing steplength rule used
        if alpha_choice == 'diminishing':
            alpha = 1/float(k)
        else:
            alpha = alpha_choice

        # record weights and cost evaluation
        weight_history.append(w)
        cost_history.append(g(w))

        # construct set of random unit directions
        directions = np.random.randn(num_samples,np.size(w))
        norms = np.sqrt(np.sum(directions*directions,axis = 1))[:,np.newaxis]
        directions = directions/norms

        ### pick best descent direction
        # compute all new candidate points
        w_candidates = w + alpha*directions

        # evaluate all candidates
        evals = np.array([g(w_val) for w_val in w_candidates])

        # if we find a real descent direction take the step in its direction
        ind = np.argmin(evals)
        if g(w_candidates[ind]) < g(w):
            # pluck out best descent direction
            d = directions[ind,:]

            # take step
            w = w + alpha*d

    # record weights and cost evaluation
    weight_history.append(w)
    cost_history.append(g(w))
    return weight_history,cost_history

# zero order coordinate search
def coordinate_search(g,alpha_choice,max_its,w):
```

```
    # construct set of all coordinate directions
    directions_plus = np.eye(np.size(w),np.size(w))
    directions_minus = - np.eye(np.size(w),np.size(w))
    directions = np.concatenate((directions_plus,directions_minus),axis=0)

    # run coordinate search
    weight_history = []         # container for weight history
    cost_history = []           # container for corresponding cost function history
    alpha = 0
    for k in range(1,max_its+1):
        # check if diminishing steplength rule used
        if alpha_choice == 'diminishing':
            alpha = 1/float(k)
        else:
            alpha = alpha_choice

        # record weights and cost evaluation
        weight_history.append(w)
        cost_history.append(g(w))

        ### pick best descent direction
        # compute all new candidate points
        w_candidates = w + alpha*directions

        # evaluate all candidates
        evals = np.array([g(w_val) for w_val in w_candidates])

        # if we find a real descent direction take the step in its direction
        ind = np.argmin(evals)
        if g(w_candidates[ind]) < g(w):
            # pluck out best descent direction
            d = directions[ind,:]

            # take step
            w = w + alpha*d

    # record weights and cost evaluation
    weight_history.append(w)
    cost_history.append(g(w))
    return weight_history,cost_history

# zero order coordinate search
def coordinate_descent_zero_order(g,alpha_choice,max_its,w):
    # run coordinate search
    N = np.size(w)
    weight_history = []         # container for weight history
    cost_history = []           # container for corresponding cost function history
    alpha = 0
    for k in range(1,max_its+1):
        # check if diminishing steplength rule used
        if alpha_choice == 'diminishing':
            alpha = 1/float(k)
        else:
            alpha = alpha_choice

        # random shuffle of coordinates
        c = np.random.permutation(N)

        # forming the dirction matrix out of the loop
        DIRECTION = np.eye(N)
        cost = g(w)

        # loop over each coordinate direction
        for n in range(N):
            #direction = np.zeros((N,1))
            #direction[c[n]] = 1
            direction = DIRECTION[:,[c[n]]]

            # record weights and cost evaluation
            weight_history.append(w)
            cost_history.append(cost)

            # evaluate all candidates
            evals =  [g(w + alpha*direction)]
            evals.append(g(w - alpha*direction))
            evals = np.array(evals)

            # if we find a real descent direction take the step in its direction
            ind = np.argmin(evals)
            if evals[ind] < cost_history[-1]:
                # take step
                w = w + ((-1)**(ind))*alpha*direction
```

```
                cost = evals[ind]

    # record weights and cost evaluation
    weight_history.append(w)
    cost_history.append(g(w))
    return weight_history,cost_history

# gradient descent function - inputs: g (input function), alpha (steplength parameter), max_its (maximum number of iterations), w (initialization)
def gradient_descent(g,alpha_choice,max_its,w):
    # flatten the input function to more easily deal with costs that have layers of parameters
    g_flat, unflatten, w = flatten_func(g, w) # note here the output 'w' is also flattened

    # compute the gradient function of our input function - note this is a function too
    # that - when evaluated - returns both the gradient and function evaluations (remember
    # as discussed in Chapter 3 we always ge the function evaluation 'for free' when we use
    # an Automatic Differntiator to evaluate the gradient)
    gradient = value_and_grad(g_flat)

    # run the gradient descent loop
    weight_history = []      # container for weight history
    cost_history = []        # container for corresponding cost function history
    alpha = 0
    for k in range(1,max_its+1):
        # check if diminishing steplength rule used
        if alpha_choice == 'diminishing':
            alpha = 1/float(k)
        else:
            alpha = alpha_choice

        # evaluate the gradient, store current (unflattened) weights and cost function value
        cost_eval,grad_eval = gradient(w)
        weight_history.append(unflatten(w))
        cost_history.append(cost_eval)

        # take gradient descent step
        w = w - alpha*grad_eval

    # collect final weights
    weight_history.append(unflatten(w))
    # compute final cost function value via g itself (since we aren't computing
    # the gradient at the final step we don't get the final cost function value
    # via the Automatic Differentiatoor)
    cost_history.append(g_flat(w))
    return weight_history,cost_history

# newtons method function - inputs: g (input function), max_its (maximum number of iterations), w (initialization)
def newtons_method(g,max_its,w,**kwargs):
    # flatten input funciton, in case it takes in matrices of weights
    flat_g, unflatten, w = flatten_func(g, w)

    # compute the gradient / hessian functions of our input function -
    # note these are themselves functions.  In particular the gradient -
    # - when evaluated - returns both the gradient and function evaluations (remember
    # as discussed in Chapter 3 we always ge the function evaluation 'for free' when we use
    # an Automatic Differntiator to evaluate the gradient)
    gradient = value_and_grad(flat_g)
    hess = hessian(flat_g)

    # set numericxal stability parameter / regularization parameter
    epsilon = 10**(-7)
    if 'epsilon' in kwargs:
        epsilon = kwargs['epsilon']

    # run the newtons method loop
    weight_history = []      # container for weight history
    cost_history = []        # container for corresponding cost function history
    for k in range(max_its):
        # evaluate the gradient, store current weights and cost function value
        cost_eval,grad_eval = gradient(w)
        weight_history.append(unflatten(w))
        cost_history.append(cost_eval)

        # evaluate the hessian
        hess_eval = hess(w)

        # reshape for numpy linalg functionality
        hess_eval.shape = (int((np.size(hess_eval))**(0.5)),int((np.size(hess_eval))**(0.5)))

        # solve second order system system for weight update
        #w = w - np.dot(np.linalg.pinv(hess_eval + epsilon*np.eye(np.size(w))),grad_eval)

        # solve second order system system for weight update
```

```
        A = hess_eval + epsilon*np.eye(np.size(w))
        b = grad_eval
        w = np.linalg.lstsq(A,np.dot(A,w) - b)[0]

    # collect final weights
    weight_history.append(unflatten(w))
    # compute final cost function value via g itself (since we aren't computing
    # the gradient at the final step we don't get the final cost function value
    # via the Automatic Differentiatoor)
    cost_history.append(flat_g(w))

    return weight_history,cost_history

def gradient_descent_momentum(g,alpha_choice,max_its,w, beta):
    # flatten the input function to more easily deal with costs that have layers of parameters
    g_flat, unflatten, w = flatten_func(g, w) # note here the output 'w' is also flattened

    # compute the gradient function of our input function - note this is a function too
    # that - when evaluated - returns both the gradient and function evaluations (remember
    # as discussed in Chapter 3 we always ge the function evaluation 'for free' when we use
    # an Automatic Differntiator to evaluate the gradient)
    gradient = value_and_grad(g_flat)

    # run the gradient descent loop
    weight_history = []      # container for weight history
    cost_history = []        # container for corresponding cost function history
    d_eval = []
    grad_eval_iter = []
    d_eval_iter = []
    alpha = 0
    hp = np.zeros([np.size(w),1])
    hp = np.reshape(hp,(2,))
    for k in range(1,max_its+1):
        # check if diminishing steplength rule used
        if alpha_choice == 'diminishing':
            alpha = 1/float(k)
        else:
            alpha = alpha_choice

        # evaluate the gradient, store current (unflattened) weights and cost function value
        cost_eval,grad_eval = gradient(w)
        grad_eval_iter.append(grad_eval)
        weight_history.append(unflatten(w))
        cost_history.append(cost_eval)

        # take gradient descent stepwith momentum acceleration
        if k < 2 :
            d_eval = -grad_eval_iter[-1]

        else :
            d_eval = beta*(d_eval_iter[-1]) + (1-beta)*(-grad_eval_iter[-1])

        w = w + alpha*d_eval
        d_eval_iter.append(d_eval)
    # collect final weights
    weight_history.append(unflatten(w))
    # compute final cost function value via g itself (since we aren't computing
    # the gradient at the final step we don't get the final cost function value
    # via the Automatic Differentiator)
    cost_history.append(g_flat(w))
    return weight_history,cost_history


def gradient_descent_full_norm(g,alpha_choice,max_its,w):
    # flatten the input function to more easily deal with costs that have layers of parameters
    g_flat, unflatten, w = flatten_func(g, w) # note here the output 'w' is also flattened

    # compute the gradient function of our input function - note this is a function too
    # that - when evaluated - returns both the gradient and function evaluations (remember
    # as discussed in Chapter 3 we always ge the function evaluation 'for free' when we use
    # an Automatic Differntiator to evaluate the gradient)
    gradient = value_and_grad(g_flat)

    # run the gradient descent loop
    weight_history = []      # container for weight history
    cost_history = []        # container for corresponding cost function history
    alpha = 0
    for k in range(1,max_its+1):
        # check if diminishing steplength rule used
        if alpha_choice == 'diminishing':
            alpha = 1/float(k)
        else:
```

```
            alpha = alpha_choice

        # evaluate the gradient, store current (unflattened) weights and cost function value
        cost_eval,grad_eval = gradient(w)
        weight_history.append(unflatten(w))
        cost_history.append(cost_eval)

        # take gradient descent step
        w = w - alpha*(grad_eval/np.linalg.norm(grad_eval))

    # collect final weights
    weight_history.append(unflatten(w))
    # compute final cost function value via g itself (since we aren't computing
    # the gradient at the final step we don't get the final cost function value
    # via the Automatic Differentiatoor)
    cost_history.append(g_flat(w))
    return weight_history,cost_history

def gradient_descent_component(g,alpha_choice,max_its,w):
    # flatten the input function to more easily deal with costs that have layers of parameters
    g_flat, unflatten, w = flatten_func(g, w) # note here the output 'w' is also flattened

    # compute the gradient function of our input function - note this is a function too
    # that - when evaluated - returns both the gradient and function evaluations (remember
    # as discussed in Chapter 3 we always ge the function evaluation 'for free' when we use
    # an Automatic Differntiator to evaluate the gradient)
    gradient = value_and_grad(g_flat)

    # run the gradient descent loop
    weight_history = []      # container for weight history
    cost_history = []        # container for corresponding cost function history
    alpha = 0
    for k in range(1,max_its+1):
        # check if diminishing steplength rule used
        if alpha_choice == 'diminishing':
            alpha = 1/float(k)
        else:
            alpha = alpha_choice

        # evaluate the gradient, store current (unflattened) weights and cost function value
        cost_eval,grad_eval = gradient(w)
        weight_history.append(unflatten(w))
        cost_history.append(cost_eval)

        # take gradient descent step with grad component wise normalisation
        # print('-------------------')
        # print('grad_eval is:', grad_eval)
        # grad_abs = abs(grad_eval)
        # grad_abs = grad_eval/grad_abs
        # print('grad abs is:', grad_abs)
        grad_eval = grad_eval/(abs(grad_eval)+1e-13)
        w = w - alpha*grad_eval

    # collect final weights
    weight_history.append(unflatten(w))
    # compute final cost function value via g itself (since we aren't computing
    # the gradient at the final step we don't get the final cost function value
    # via the Automatic Differentiatoor)
    cost_history.append(g_flat(w))
    return weight_history,cost_history
```