Name: Elena Pan
Date: Oct 27th, 2021

Project 2 Report

Goal: Solving non-linear least square problem

$$\min_{w \in \mathbb{R}^m} \sum_{k=1}^{n} |y_k - N(x_k; w)|^2$$

Algorithms: Steepest Descent, Fixed Step-size, and Conjugate Gradient Methods

1) Algorithm 1: Steepest Descent Method

Input:

[x,y]=getData(1000,2,2471729356)
[weight_m, cost_m, i] = Steepest_Descent(x, y)

Result:

stopping criteria = `((cost_m(length(cost_m)-1) - cost_m(length(cost_m))) > 10^(-3))`

Iteration number = 26

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
897.0358	46.3085	45.9657	45.7910	45.7621	45.7519	45.7455	45.7406
9	**10**	**11**	**12**	**13**	**14**	**15**	**16**
45.7364	45.7325	45.7291	45.7260	45.7233	45.7207	45.7184	45.7162
17	**18**	**19**	**20**	**21**	**22**	**23**	**24**
45.7143	45.7125	45.7109	45.7093	45.7079	45.7065	45.7053	45.7041
25	**26**	**27**					
45.7030	45.7019	45.7010					

Conclusion:

Starting point is 897.0358. The first iteration starts from #2, so there are 26 iterations in total. And the cost function is convergent to 45.7010 when it meets the stopping criteria.

2) Algorithm 2: Fixed Step-size Method

Input:

min_a = 0.01
[x,y]=getData(1000,2,2471729356)
[weight_m, cost_m, i] = Fixed_Step(x, y)

Result:

stopping criteria = `((cost_m(length(cost_m)-1) - cost_m(length(cost_m))) > 10^(-3))`

Iteration number = 96

1	**2**	**3**	**4**	**5**	**6**	**7**	**…**
897.0358	724.6799	46.5192	46.5009	46.4830	46.4657	46.4490	…
90	**91**	**92**	**93**	**94**	**95**	**96**	**97**
45.9415	45.9402	45.9389	45.9377	45.9366	45.9355	45.9344	45.9335

Question: Convergence of the algorithm for different selection of step-sizes for fixed step-size method

Step sizes = 20, 10, 1, 0.1, 0.01, 0.001, 0.0001

min_a	20	10	1	0.1	0.01	0.001	0.0001
Cost	897.0358	897.0358	897.0358	897.0358	897.0358	897.0358	897.0358
	48.0	48.0	48.0	47.9980	724.6799	888.0974	896.1983
	NaN	48.0	48.0	47.9980	…	876.0979	895.3363
						…	…
						46.1933	46.4283
					45.9335	46.1923	46.4273
Iterations	2	2	2	2	96	138	699
Convergent	No	Yes	Yes	Yes	Yes	Yes	Yes

Conclusion:

I tried 7 min_a represents for different selection of step sizes. When the step-size equals to 20, it is too large so that the cost function is not convergent after 2 iterations. For min_a = 10 and 1, although the cost function converges but it is not the ideal step size. Then, I tried min_a = 0.01, there're 96 iterations and the cost function is 45.9335, which is the lowest cost function. To see whether taking even smaller step sizes will generate a lower cost, I chose step size equals to 0.001 and 0.0001. The result turns out that their cost functions are greater than 45.9335. As the step sizes are relatively small, more iterations are needed in terms of taking longer runtime. As a result, step size equals to 0.01 is the optimal one and cost function is convergent at 45.9335.

3) Algorithm 3: Conjugate Gradient Method

Input:

[x,y]=getData(1000,2,2471729356)
[weight_m, cost_m, i] = Conjugate(x, y)

Result:

stopping criteria = `((cost_m(length(cost_m)-1) - cost_m(length(cost_m))) > 10^(-3))`

Iteration number = 2

Iterations	1	2	3
Cost	897.0358	46.3085	46.3085

Question: Compare Performance of Steepest Descent, Fixed Step-size, and Conjugate Gradient Methods

Method	Steepest Descent	Fixed Step-size	Conjugate Gradient
Iterations	26	96	2
Cost	45.7010	45.9335	46.3085

Conclusion:

Considering iteration numbers, Conjugate Gradient is the best algorithm as it only takes 2 iterations. When three algorithms start at the same point, at each iteration, Conjugate Gradient method will yield a lower cost than the Steepest Descent and Fixed Step-size does. Considering the non-linear least square function, Steepest Descent method yields a relatively smaller cost compared to the other two. Although fixed step-size method needs more iterations, its method is the simplest. Instead of finding min_a using fminsearch method, we give a fixed step size every time. However, this also gives the disadvantages as we don't know which fixed step size is the idea and may try several times. From this project, I have a deeper understand of these three algorithms, so much fun!

Appendix A Steepest Descent Method

```
function [weight_m, cost_m, i] = Steepest_Descent(x, y)

% create the network
[network]=createNetwork(2,[3,3,1]);

% use networkFProp to generate yVal and calculate gradient
[yVal,yintVal]=networkFProp(x,network);
yGrad=networkBProp(network,yintVal);
yGrad = squeeze(yGrad);
gradient = -2.*yGrad*(y-yVal)';

% generate weight
weight = getNNWeight(network);

% weight = weight - a*gradient;
% network = setNNWeight(network,weight);
% yVal=networkFProp(x,network);

% cost function
cost = sum((y - yVal).^2);

% create weight and cost matrix
weight_m = [weight];
cost_m = [cost];
i = 0;

while (i < 1) || ((cost_m(length(cost_m)-1) - cost_m(length(cost_m))) > 10^(-
3))

    % use fminsearch to find min_a
    func = @(a) sum((y - networkFProp(x,setNNWeight(network,(weight -
a.*gradient)))).^2);
    min_a = fminsearch(func, 0.01);
    disp(min_a)

    % update weight
    weight = weight - min_a.*gradient;
    weight_m = [weight_m, weight];

    % update network
    network = setNNWeight(network, weight);
    [yVal,yintVal] = networkFProp(x,network);

    % calculate cost function with updated yVal
    cost = sum((y - yVal).^2);
    cost_m = [cost_m, cost];

    % update gradient
    yGrad=networkBProp(network,yintVal);
    yGrad = squeeze(yGrad);
    gradient = -2.*yGrad*(y-yVal)';

    i = i + 1;

end

end
```

Appendix B Fixed Step-size Method

```
function [weight_m, cost_m, i] = Fixed_Step(x, y)

% create network
[network]=createNetwork(2,[3,3,1]);

% use networkbProp to generate yGrad and calculate gradient vector
[yVal,yintVal]=networkFProp(x,network);
yGrad=networkBProp(network,yintVal);
yGrad = squeeze(yGrad);
gradient = -2.*yGrad*(y-yVal)';

% generate weight
weight = getNNWeight(network);

% weight = weight - a*gradient;
% network = setNNWeight(network,weight);
% yVal=networkFProp(x,network);

% cost function
cost = sum((y - yVal).^2);

% create weight and cost matrix
weight_m = [weight];
cost_m = [cost];
i = 0;

% set fixed step-size
min_a = 0.01;

while (i < 1) || ((cost_m(length(cost_m)-1) - cost_m(length(cost_m))) > 10^(-
3))

    % update weight
    weight = weight - min_a.*gradient;
    weight_m = [weight_m, weight];

    % update network
    network = setNNWeight(network, weight);
    [yVal,yintVal] = networkFProp(x,network);
    cost = sum((y - yVal).^2);
    cost_m = [cost_m, cost];

    % update gradient
    yGrad=networkBProp(network,yintVal);
    yGrad = squeeze(yGrad);
    gradient = -2.*yGrad*(y-yVal)';

    i = i + 1;

end

end
```

Appendix C Conjugate Gradient Method

```
function [weight_m, cost_m, i] = Conjugate(x, y)

% create network
[network]=createNetwork(2,[3,3,1]);

% use networkBProp to generate yGrad and calculate gradient
[yVal,yintVal]=networkFProp(x,network);
yGrad=networkBProp(network,yintVal);
yGrad = squeeze(yGrad);
gradient = -2.*yGrad*(y-yVal)';

% d0 = g0
direction = gradient;

% generate weight
weight = getNNWeight(network);

% weight = weight - a*gradient;
% network = setNNWeight(network,weight);
% yVal=networkFProp(x,network);

% cost function
cost = sum((y - yVal).^2);

% create weight and cost matrix
weight_m = [weight];
cost_m = [cost];
i = 0;

while (i < 1) || ((cost_m(length(cost_m)-1) - cost_m(length(cost_m))) > 10^(-
3))

    % func: gradient --> direction
    func = @(a) sum((y - networkFProp(x,setNNWeight(network,(weight - a.*
direction))))).^2);
    min_a = fminsearch(func, 0.01);

    % update weight
    weight = weight - min_a.* direction;
    weight_m = [weight_m, weight];

    % update network
    network = setNNWeight(network, weight);
    [yVal,yintVal] = networkFProp(x,network);
    yGrad=networkBProp(network,yintVal);
    yGrad = squeeze(yGrad);
    gradient = -2.*yGrad*(y-yVal)';

    % generate gradient_k which is g(k+1)
    gradient_k = -2.*yGrad*(y-yVal)';

    cost = sum((y - yVal).^2);
    cost_m = [cost_m, cost];

    % Use Fletcher-Reeves Formula to generate beta
    beta = (gradient_k'*gradient_k)/(gradient'*gradient);

    % d(k+1)
```

```
    direction_k = gradient_k - beta * direction;
    i = i + 1;
end
end
```