Yuchen Shi
6962247146
MATH 467
November 7th, 2018

Project 1 Report

Introduction

In this project, I investigate the behavior of three algorithms when used for training of a neural network. The three algorithms are fixed step-size gradient method, steepest descent gradient method, and conjugate gradient method. Specifically, I am trying to minimize the following function:

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

where $N(x_k;w)$ is the neural network that I am training, x_k and y_k are the set of training data. Training the neural network basically means choosing the weight w. This function represents the error between true y_k value and the y_k value we get from the neural network. If the error is very small, it means the neural network is trained very well, and the choose of weight w is very good. As a result, I will implement the three algorithms to find a weight w that will try to minimize this function.

Topic

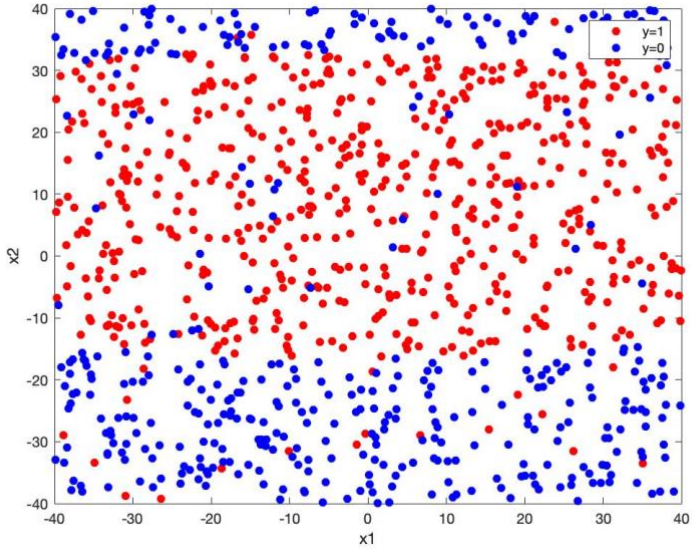
While experimenting the three algorithms, I will focus on investigating the following two topics:

1. Convergence of the algorithm for different selection of step-sizes for fixed step-size method.
2. Comparison of performance between fixed step-size gradient method and steepest descent of conjugate gradient methods.

Process

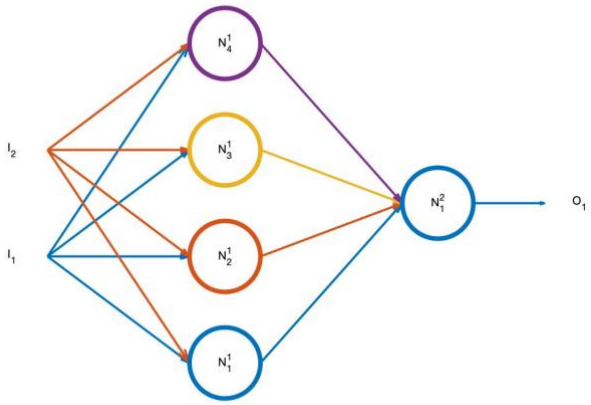
I choose the dimension of input vectors x_k to be 2, and I choose the number of data points to be 1000.

Firstly, I draw a scatterplot to see how the 1000 data points look like.



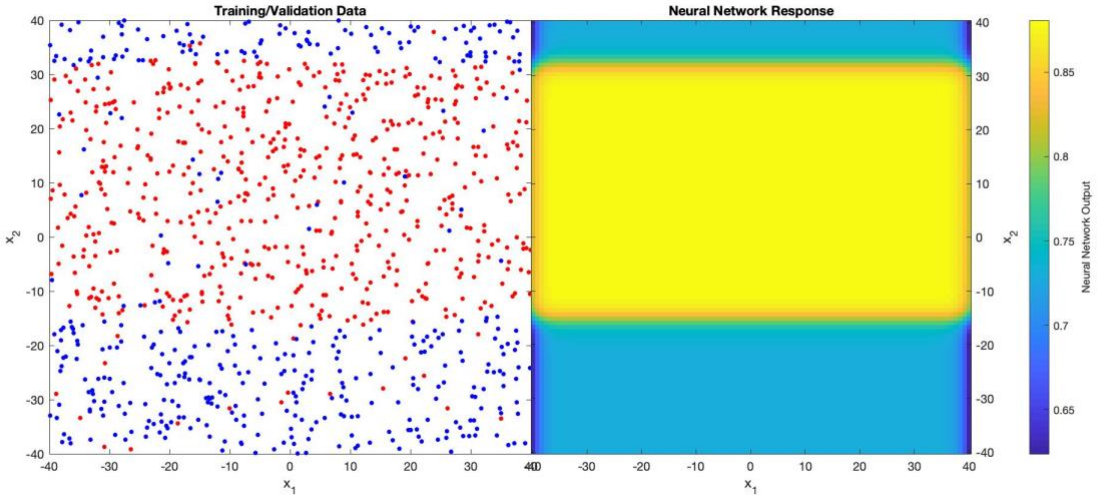
From this graph, I could see that 4 lines would probably separate the red points and blue points. The 4 lines are: $x_1 + 40 = 0$, $x_1 - 40 = 0$, $x_2 - 32 = 0$, $x_2 + 15 = 0$.

Thus, I determine that my neural network only need two layers: the first layer would have 4 neurons, and the second layer would have 1 neuron.



There would be a total of 17 weights. I choose my initial weight based on the 4 lines so that it is close to the optimal weight that I am trying to find to minimize the function. The initial weight is a 17 by 1 vector: [1;-1;0;0;0;0;-1;1;40;40;32;15;1;1;1;1;-2].

I also draw the response of the neural network based on this initial weight.



We could see that the shape of the response is already pretty good, but the color is not good enough. The initial network could almost determine the red points to have value of 1, but could only determine the blue points to have value of 0.75; we want it to be sure that the blue points will have value of 0. As a result, we start the three algorithms to train this neural network.

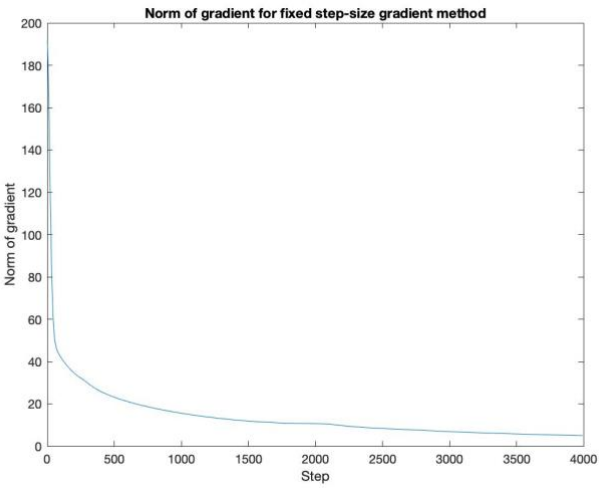
Firstly, I implement a **fixed step-size gradient method**. The algorithm is as follows:

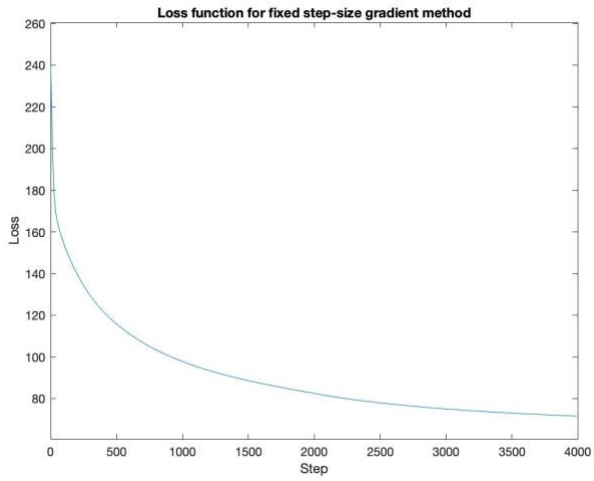
Fixed step-size gradient method algorithm:
1. Start with the initial weight w_0
2. For each step k, $w_{k+1} = w_k - \alpha g_k$, where

$$g_k = \nabla f(w) = -2\sum_{k=1}^{n}(y_k - N(x_k; w))\nabla_w N(x_k; w)$$

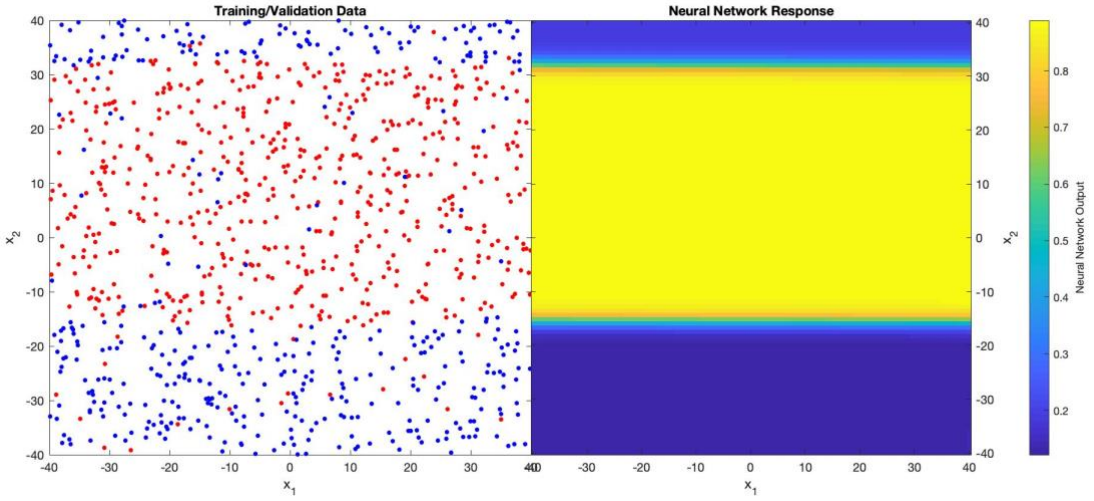
Stop when $\|g_k\| \leq 5$

The norm of the initial gradient is 191.61, so I set my stopping condition to be 5, which means I would have decreased the value of gradient by 1/40.
Here I show the result of using a fixed step size of 0.0001, but I also explored other different step sizes to experiment Topic 1, which I will talk more in the conclusion section. Using a fixed step size of 0.0001, the algorithm is definitely converging, the loss function which is just the function we are trying to minimize has decreased from 241.69 to 71.6 within 3992 steps.





I also draw the response of the trained neural network to compare with that of the untrained neural network. We could see that the color for the data above the line $x_2 - 32 = 0$ and below the line $x_2 + 15 = 0$ is darker, which means the network is more affirmative that those data points have value of 0, which is correct. As a result, this training is successful.



Then, I implement the **steepest descent gradient method**. The algorithm is as follows:

Steepest descent gradient method algorithm:
1. Start with the initial weight w_0
2. For each step k, $w_{k+1} = w_k - \alpha_k g_k$, where

$$g_k = \nabla f(w) = -2\sum_{k=1}^{n}(y_k - N(x_k; w))\nabla_w N(x_k; w)$$

Stop when $\|g_k\| \leq 5$

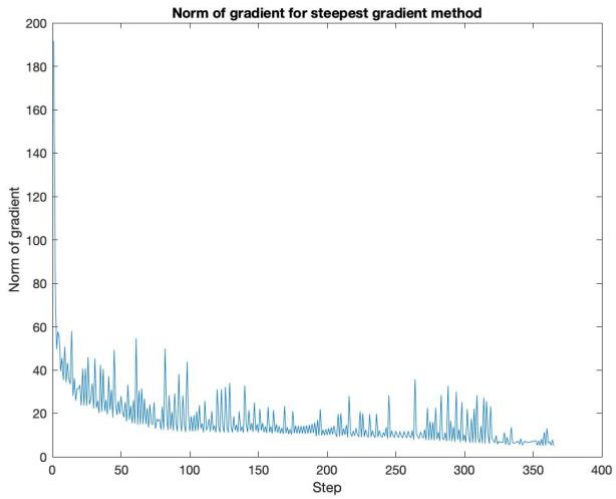
a_k is determined via line search. The line search algorithm is as follows:

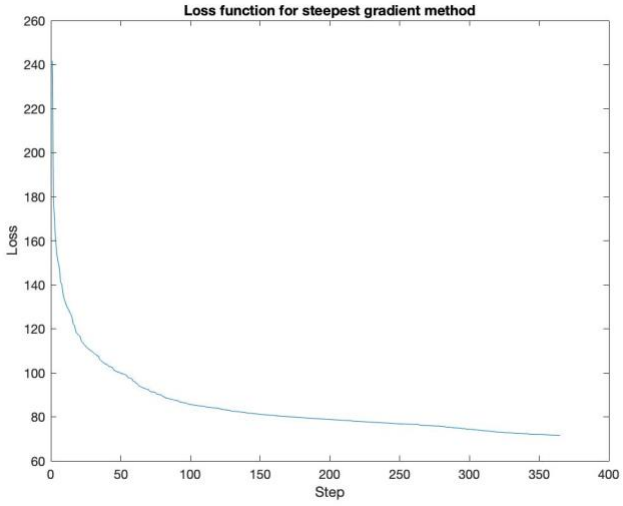
Line search:
1. Start with $\alpha = \alpha_0$
2. Test if $f(w + \alpha d) \leq f(w) + \beta\alpha\nabla f(w)^T d, \quad 0 < \beta < 1$
If yes, stop; Otherwise, updata $\alpha = t\alpha, \quad 0 < t < 1$

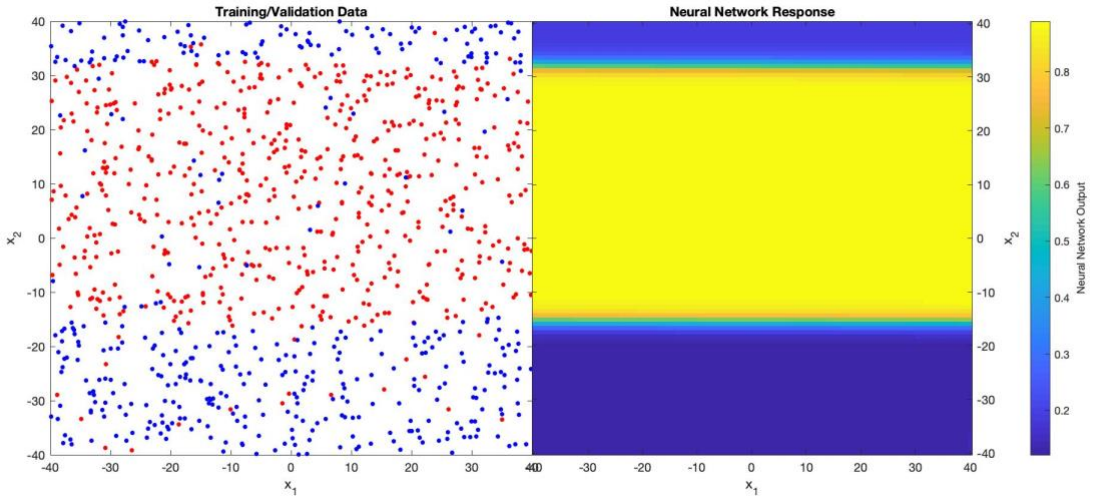
I choose α_0 to be 0.005, both t and β to be 0.5. And d is the negative gradient direction - g_k.

Again, the stopping condition is 5. I use a constant stopping condition for all three algorithms so that they are decreasing the gradient by the same amount. Therefore, I could compare their performances based on how much they could decrease the loss function.

This algorithm is also converging, and the loss function decreases from 241.69 to 71.53 within 364 steps.







The response of the trained neural network also looks very good. The neural network could effectively identify the points where y=0 (the blue points) and the points where y=1 (the red point).

Finally, I implement the **conjugate gradient method**. The algorithm is as follows:

Conjugate gradient method algorithm:

1. Start with the initial weight w_0
2. The initial conjugate direction $v_0 = g_0 = \nabla f(w_0)$
3. For each step k, $w_{k+1} = w_k - \alpha_k v_k$
4. $v_{k+1} = g_{k+1} - \beta_k v_k$, where $\beta_k = \dfrac{{g_{k+1}}^T g_{k+1}}{g_k^T g_k}$

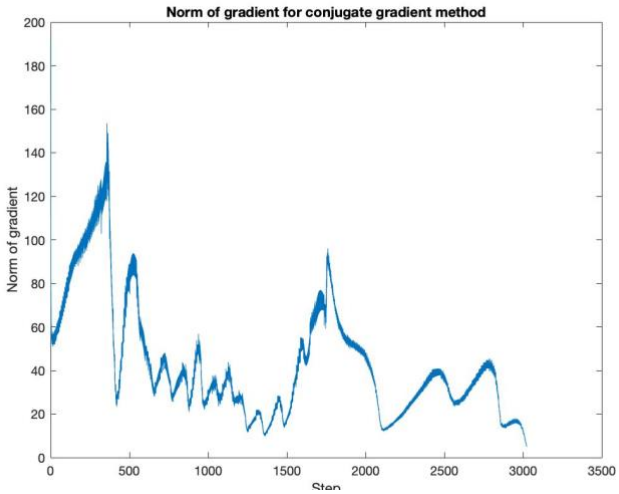
Stop when $\|g_k\| \leq 5$

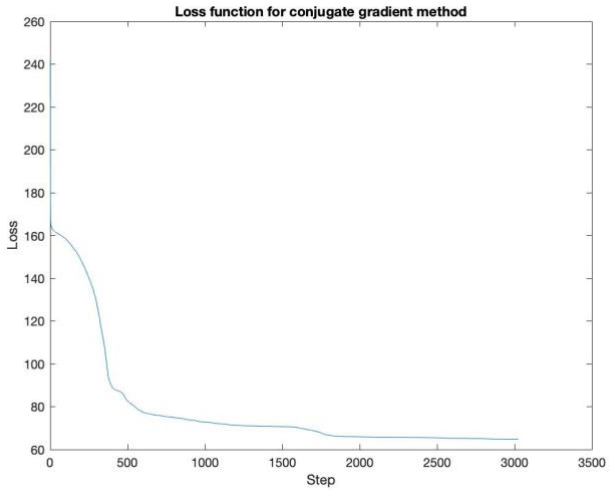
Again, a_k is determined via line search. The line search algorithm is the same as before. α_0 is chosen to be 0.005, both t and β is 0.5, and this time, d is the negative conjugate direction $-v_k$.

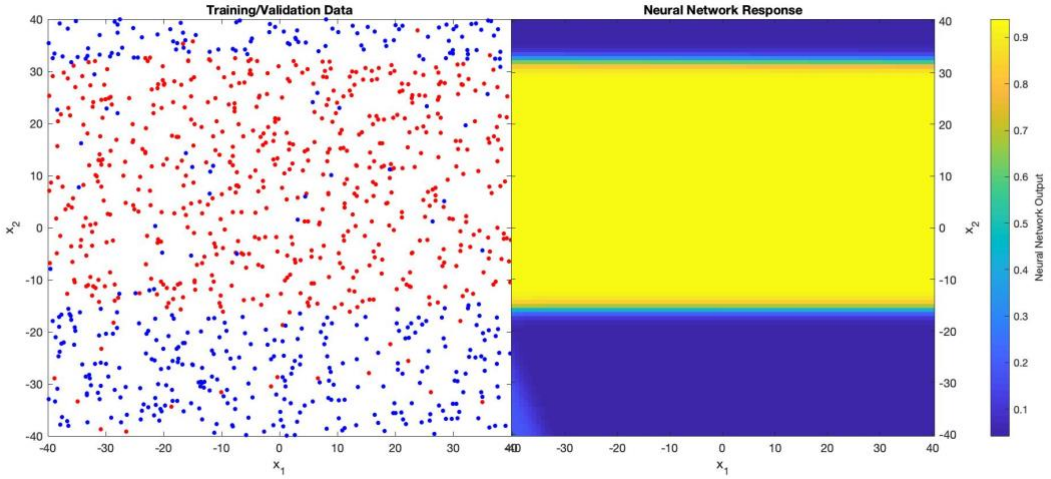
Line search:
1. Start with $\alpha = \alpha_0$
2. Test if $f(w + \alpha d) \leq f(w) + \beta\alpha\nabla f(w)^T d, \quad 0 < \beta < 1$
If yes, stop; Otherwise, updata $\alpha = t\alpha, \quad 0 < t < 1$

My stopping condition is still 5, which means I would have decreased the value of gradient by 1/40.

This algorithm is still converging. Although the norm of gradient is oscillating a lot, generally, it is still decreasing, and if I actually apply a smaller stopping condition like 1 or 0.5, the algorithm could still reach it. Most importantly, the loss function is keep decreasing, which means this algorithm is minimizing this function of weight successfully. If the stopping condition is 5 (same level as in the fixed step-size and steepest descent gradient method), the loss function decreases from 241.69 to 64.8 within 3023 steps.



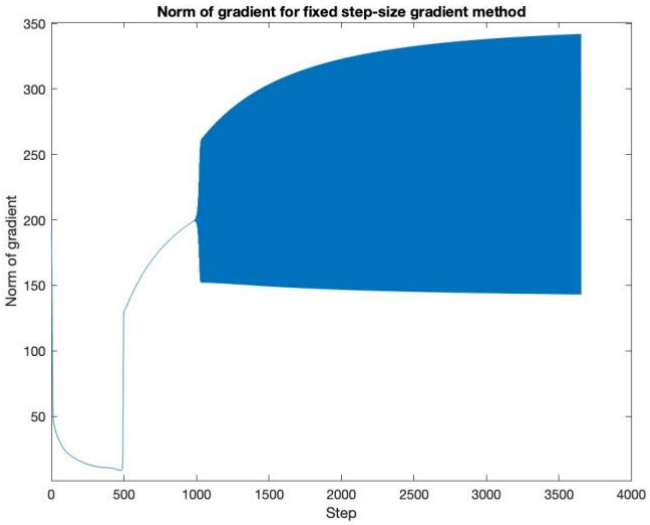


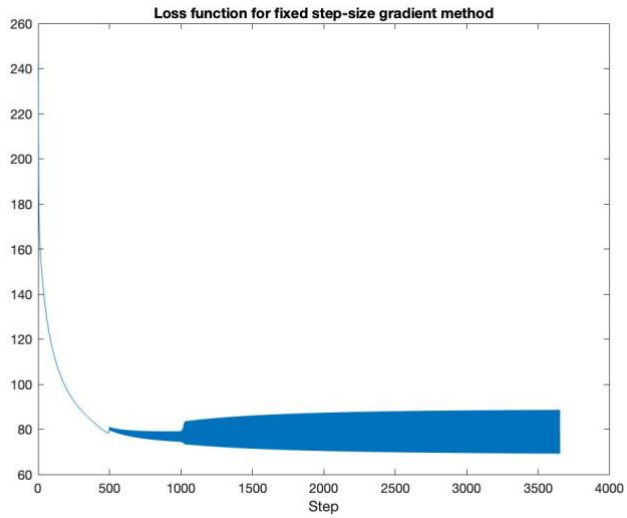


Again, the response of the neural network after the training looks pretty good. The neural network could identify almost all the blue points and red points correctly.

Conclusion

Regarding the first topic, the different selection of step sizes for the fixed step-size algorithm has a great impact on the convergence of this algorithm. If I use a larger step size such as 0.0005, the algorithm would not perform well:

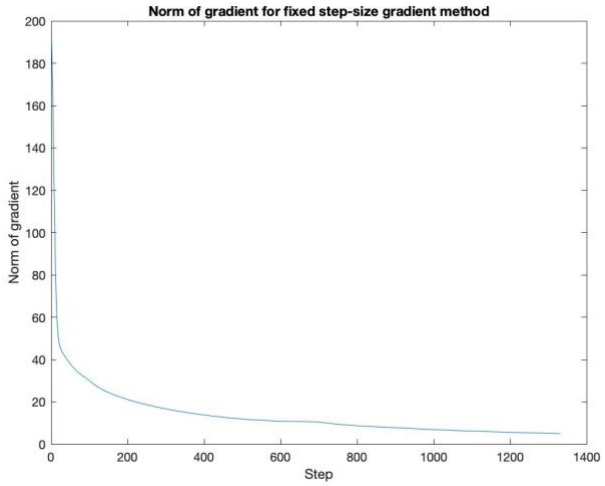


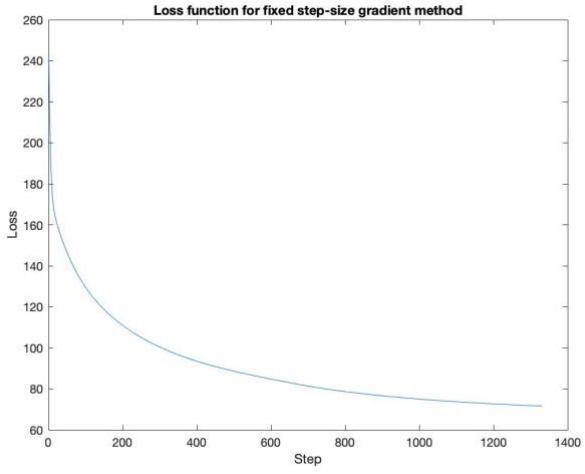


This is definitely not converging, the norm of gradient is oscillating around a value that is even higher than the original norm of gradient. The loss function is also not keep decreasing. As a result, this step size is not working.

Because it is fixed, if the step size is large, it is very likely that the weight we are iterating jumps through the minimum, so basically it is jumping between the left and right of the minimum but never reaching a point close to it, which is the reason the graph is oscillating, and the algorithm never reach the stopping condition.

Smaller step size would make the algorithm converging. For my data set, if I use a fixed step size of 0.0003, the algorithm would converge (see the graphs below), but if I use 0.0004 step size, the graphs of norm of gradient and loss function would look similar to those in the case of 0.0005 step size (see the graphs above), which is not converging.





To sum up, the fixed step-size algorithm needs a step size small enough to converge. The smaller the step size is, the longer it takes to converge, because the weight is updating very slowly. However, if we use a large step size, the algorithm may not even converge.

Regarding the second topic, if I use the same stopping condition for the three algorithms, the fixed step-size gradient method converges in 3992 iterations, the steepest descent gradient method converges in 364 iterations, and the conjugate gradient method converges in 3023 iterations. The fixed step-size algorithm is the slowest; while the steepest descent gradient method is the fastest. The loss function decreases from 241.69 to 71.6 for fixed step-size gradient method, it decreases from 241.69 to 71.53 for steepest descent gradient method, and it decreases from 241.69 to 64.8 for conjugate gradient method. As a result, the conjugate gradient method has the best performance, it results in the lowest value of the function that we are minimizing. The performance of fixed step-size and steepest descent gradient method is similar: they all decrease the function to around 71.5, but it takes much smaller steps for the steepest descent gradient method to reach it.

Appendix

Code for *fixed step-size gradient method*:

```
% Fixed-step gradient descent
[x,y]=getData(1000,2,69622471);
x1=x(1,:);
x2=x(2,:);
ind = find(y==1);
plot(x(1,ind),x(2,ind),'o','MarkerFaceColor','r','MarkerEdgeColor','r','Mar
kerSize',5);
hold on
ind = find(y==0);
plot(x(1,ind),x(2,ind),'o','MarkerFaceColor','b','MarkerEdgeColor','b','Mar
kerSize',5);
legend('y=1','y=0');
xlabel('x1');
ylabel('x2');

[network]=createNetwork(2,[4,1]);
VisualizeNN(network)
[Weight]=getNNWeight(network);
weight=[1;-1;0;0;0;0;-1;1;40;40;32;15;1;1;1;1;-2];
[network]=setNNWeight(network,weight);
CompareData2NNResponse_2D(x,y,network)

[yVal,yintVal]=networkFProp(x,network);
[yGrad,yGrad_Struct]=networkBProp(network,yintVal);

yGrad=reshape(yGrad,[17,1000]);
gradient=-2*yGrad*(y-yVal)';
gradVal=norm(gradient,2);
index=1;
gradArray(index)=gradVal;
loss(index)=(y-yVal)*(y-yVal)';
while gradVal>5
     weight=weight-0.0005*gradient;
     [network]=setNNWeight(network,weight);
     [yVal,yintVal]=networkFProp(x,network);
     [yGrad,yGrad_Struct]=networkBProp(network,yintVal);
     yGrad=reshape(yGrad,[17,1000]);
     gradient=-2*yGrad*(y-yVal)';
     gradVal=norm(gradient,2);
     index=index+1;
     gradArray(index)=gradVal;
     loss(index)=(y-yVal)*(y-yVal)';
end

CompareData2NNResponse_2D(x,y,network)
```

Code for *steepest descent gradient method*:

```
% Steepest gradient descent
a=0.5;
b=0.5;
t0=0.005; % Default step size of 0.005
```

```
[x,y]=getData(1000,2,69622471);
x1=x(1,:);
x2=x(2,:);
ind = find(y==1);
plot(x(1,ind),x(2,ind),'o','MarkerFaceColor','r','MarkerEdgeColor','r','Mar
kerSize',5);
hold on
ind = find(y==0);
plot(x(1,ind),x(2,ind),'o','MarkerFaceColor','b','MarkerEdgeColor','b','Mar
kerSize',5);
legend('y=1','y=0');
xlabel('x1');
ylabel('x2');

[network]=createNetwork(2,[4,1]);
VisualizeNN(network)
weight=[1;-1;0;0;0;0;-1;1;40;40;32;15;1;1;1;1;-2];
[network]=setNNWeight(network,weight);
CompareData2NNResponse_2D(x,y,network)

[yVal,yintVal]=networkFProp(x,network);
[yGrad,yGrad_Struct]=networkBProp(network,yintVal);
yGrad=reshape(yGrad,[17,1000]);
gradient=-2*yGrad*(y-yVal)';
gradVal=norm(gradient,2);
index=1;
gradArray(index)=gradVal;
loss(index)=(y-yVal)*(y-yVal)';
while gradVal>5
    % line search to determine step size
    gradNorm=gradient'*gradient;
    updatedWeight=weight-t0*gradient;
    [updatedNetwork]=setNNWeight(network,updatedWeight);
    [yValUpdated,yintValUpdated]=networkFProp(x,updatedNetwork);
    t=t0;
    while (y-yValUpdated)*(y-yValUpdated)' > (y-yVal)*(y-yVal)'-
a*t*gradNorm
        t=t*b;
        updatedWeight=weight-t*gradient;
        [updatedNetwork]=setNNWeight(network,updatedWeight);
        [yValUpdated,yintValUpdated]=networkFProp(x,updatedNetwork);
    end
    % Update weight based on the step size we determined from the line
search
    weight=weight-t*gradient;
    tArray(index)=t;
    [network]=setNNWeight(network,weight);
    [yVal,yintVal]=networkFProp(x,network);
    [yGrad,yGrad_Struct]=networkBProp(network,yintVal);
    yGrad=reshape(yGrad,[17,1000]);
    gradient=-2*yGrad*(y-yVal)';
    gradVal=norm(gradient,2);
    index=index+1;
    gradArray(index)=gradVal;
    loss(index)=(y-yVal)*(y-yVal)';
end

CompareData2NNResponse_2D(x,y,network)
```

Code for *conjugate gradient method*:

```
% Conjugate gradient method
a=0.5;
b=0.5;
t0=0.005; % Default step size of 0.005

[x,y]=getData(1000,2,69622471);
x1=x(1,:);
x2=x(2,:);
ind = find(y==1);
plot(x(1,ind),x(2,ind),'o','MarkerFaceColor','r','MarkerEdgeColor','r','Mar
kerSize',5);
hold on
ind = find(y==0);
plot(x(1,ind),x(2,ind),'o','MarkerFaceColor','b','MarkerEdgeColor','b','Mar
kerSize',5);
legend('y=1','y=0');
xlabel('x1');
ylabel('x2');

[network]=createNetwork(2,[4,1]);
VisualizeNN(network)
weight=[1;-1;0;0;0;0;-1;1;40;40;32;15;1;1;1;1;-2];
[network]=setNNWeight(network,weight);
CompareData2NNResponse_2D(x,y,network)

[yVal,yintVal]=networkFProp(x,network);
[yGrad,yGrad_Struct]=networkBProp(network,yintVal);
yGrad=reshape(yGrad,[17,1000]);
gradient=-2*yGrad*(y-yVal)';
gradVal=norm(gradient);
conjugate=gradient;
index=1;
gradArray(index)=gradVal;
loss(index)=(y-yVal)*(y-yVal)';
while gradVal>5
    % line search to determine step size
    gradNorm=gradient'*conjugate;
    updatedWeight=weight-t0*conjugate;
    [updatedNetwork]=setNNWeight(network,updatedWeight);
    [yValUpdated,yintValUpdated]=networkFProp(x,updatedNetwork);
    t=t0;
    while (y-yValUpdated)*(y-yValUpdated)' > (y-yVal)*(y-yVal)'-
a*t*gradNorm
        t=t*b;
        updatedWeight=weight-t*conjugate;
        [updatedNetwork]=setNNWeight(network,updatedWeight);
        [yValUpdated,yintValUpdated]=networkFProp(x,updatedNetwork);
    end
    % Update weight based on the step size we determined from the line
search
    weight=weight-t*conjugate;
    tArray(index)=t;
    [network]=setNNWeight(network,weight);
    [yVal,yintVal]=networkFProp(x,network);
    [yGrad,yGrad_Struct]=networkBProp(network,yintVal);
    yGrad=reshape(yGrad,[17,1000]);
```

```
    oldGradient=gradient;
    oldConjugate=conjugate;
    gradient=-2*yGrad*(y-yVal)';
    conjugate=gradient-
((gradient'*gradient)/(oldGradient'*oldGradient))*oldConjugate;
    gradVal=norm(gradient,2);
    index=index+1;
    gradArray(index)=gradVal;
    loss(index)=(y-yVal)*(y-yVal)';
end

CompareData2NNResponse_2D(x,y,network)
```