Ishaan Chawla

Prof. Chunming Wang

Math 467

16 December 2022

Project 2

Intro

Here is my code for the basic implementation of my 2 networks.

```
function fixedStepSize2

[network]=createNetwork(2,[4,4,2]);
VisualizeNN(network);
%
% Initialize network using randomly generated weights.
%
[Weight]=getNNWeight(network);
Weight=0.01*randn(size(Weight));
[network]=setNNWeight(network,Weight); %network with optimal weight
%
% Test the output of the network
%

xVal=randn(2,100);
[yVal,yintVal]=networkFProp(xVal,network); %training data

alpha = .1;

[network2]=createNetwork(2,[4,4,2]); %network to train

[Weight2]=getNNWeight(network2);
Weight2=0.01*randn(size(Weight2));
[network2]=setNNWeight(network2,Weight2);

[yVal2,yintVal2]=networkFProp(xVal,network2);
[yGrad,yGrad_Struct]=networkBProp(network2,yintVal2);
```

```
Wgrad = zeros(size(Weight));

for k=1:1000

    for i=1:100 %evaluate weight gradient
        Wgradupdate = 2*(yVal2(:,i) - yVal(:,i))' * yGrad(:,:,i);
        Wgrad = Wgrad + Wgradupdate'; %update Wgrad
    end

    Weight2 = (Weight2 - alpha*Wgrad); %employ fixed step size method

    [network2]=setNNWeight(network2,Weight2); %set new weight

    [yVal2,yintVal2]=networkFProp(xVal,network2); %re-evaluate to get output
    [yGrad,yGrad_Struct]=networkBProp(network2,yintVal2); %new output gradient

end

Wgrad;
Weight;
Weight2;
abs(Weight-Weight2)
yVal;
yVal2;
abs(yVal-yVal2)


end
```

I used randomly generated weights for the both networks, but I treat the output data of the first as my training data. I then try to alter the second network's weights to match the first data output. The nested for loops accomplish this through the formula for the weight gradient I calculated by hand. I got $2*(N(x;w) - y)^T * dN/dw$, where dy/dw is the gradient of the output with respect to the weights. This is ygrad(: , : , i) for each data point. The output is a 1x42 row vector, i.e. the gradient of the squared sum with respect to the weights. Taking its transpose yields the weight gradient vector for a specific sample, and the sum over all sample differences gives us the weight gradient for the function.

Fixed Step Size

I subtracted the weight gradient from the original weight with a step-size of alpha = .1. This takes us closer to minimizing the differences between the two networks' outputs. My output is shown here:

```
 1.0e-03 *

Columns 1 through 9

  0.3847    0.3847    0.3847    0.3847    0.3847    0.3847    0.3847    0.3847    0.3847
  0.9752    0.9752    0.9751    0.9750    0.9751    0.9752    0.9753    0.9753    0.9752

Columns 10 through 18

  0.3847    0.3847    0.3847    0.3847    0.3847    0.3847    0.3847    0.3847    0.3847
  0.9753    0.9751    0.9752    0.9752    0.9751    0.9752    0.9751    0.9753    0.9752

Columns 19 through 27

  0.3847    0.3847    0.3847    0.3847    0.3847    0.3847    0.3847    0.3847    0.3847
  0.9751    0.9751    0.9751    0.9752    0.9753    0.9752    0.9752    0.9751    0.9752
```

Clearly after 1000 iterations the outputs of the two networks have come quite close together.

After 2000 iterations, even more so:

```
ans =

   1.0e-03 *

  Columns 1 through 9

    0.3282    0.3282    0.3283    0.3282    0.3281    0.3282    0.3282    0.3281    0.3281
    0.2423    0.2423    0.2425    0.2425    0.2424    0.2424    0.2424    0.2426    0.2424

  Columns 10 through 18

    0.3280    0.3282    0.3282    0.3281    0.3282    0.3282    0.3281    0.3281    0.3282
    0.2427    0.2423    0.2425    0.2424    0.2424    0.2422    0.2425    0.2424    0.2423
```

For a larger step size of 1, I get less precise data, as one would expect due to overshooting. 1000 iterations:

```
ans =

  Columns 1 through 9

    0.0018    0.0018    0.0018    0.0018    0.0018    0.0018    0.0018    0.0018    0.0018
    0.0015    0.0015    0.0015    0.0015    0.0015    0.0015    0.0015    0.0015    0.0015

  Columns 10 through 18

    0.0018    0.0018    0.0018    0.0018    0.0018    0.0018    0.0018    0.0018    0.0018
    0.0015    0.0015    0.0015    0.0015    0.0015    0.0015    0.0015    0.0015    0.0015
```

And for a smaller step size of .01, while I can get better data, I need more iterations to move closer to the minimum. This is computationally exhausting for the computer but perhaps worth it. After 5000 iterations the error is ~½ that of 2000 iterations of a .1 step size. That's 2.5 times the work for 2 times better results.

```
ans =

   1.0e-03 *

  Columns 1 through 9

    0.2885    0.2887    0.2889    0.2887    0.2887    0.2888    0.2886    0.2885    0.2889
    0.0690    0.0690    0.0691    0.0692    0.0691    0.0690    0.0690    0.0690    0.0692

  Columns 10 through 18

    0.2887    0.2887    0.2888    0.2887    0.2886    0.2887    0.2887    0.2887    0.2887
    0.0691    0.0690    0.0691    0.0690    0.0690    0.0689    0.0690    0.0691    0.0692
```

alpha = .01 seems to be a sweet spot. Trying .001 took too long to make any progress. After 5000 iterations, my data was still not very good compared to .01 step size.

```
ans =

  Columns 1 through 9

    0.0007    0.0007    0.0007    0.0007    0.0007    0.0007    0.0007    0.0007    0.0007
    0.0070    0.0070    0.0070    0.0070    0.0070    0.0070    0.0070    0.0070    0.0070

  Columns 10 through 18

    0.0007    0.0007    0.0007    0.0007    0.0007    0.0007    0.0007    0.0007    0.0007
    0.0070    0.0070    0.0070    0.0070    0.0070    0.0070    0.0070    0.0070    0.0070
```

What's interesting is my weights fluctuate and can be quite different, but still the output continues to get closer and closer to the training data. This is likely an example of the ease of finding local minimums but difficulty in obtaining the global minimum. Regardless, the algorithm converges as expected.

Steepest Descent

For this one I eliminated alpha and did a line search in increments of .01 directly using the Weight gradient, W grad. I simply kept updating until the newest iteration went above the most recent one, meaning I passed a minimizer. Here is my code:

```
Wlast = Weight2; %employ steepest descent method
Wnew = Weight2;
while Wnew -Wlast < 0
    Wlast = Wnew;
    Wnew = Wlast - .01*Wgrad;
end
Weight2 = Wlast;
```

I got good results faster than the fixed step size method, getting decent results with only ~50 iterations. However, the accuracy seems to almost plateau. It does get better, but slowly. I assume this is because the search gets trapped in areas that are not the global minimizer and so any

further progress is likely minimal. The fixed step size is still advantageous in that its fixed step can allow some probability for it to leave a local minimum and discover a new, more optimal one. However, for fast and decent results, this algorithm is much less computationally intensive.

Stochastic Gradient Descent

I implemented this making the necessary alterations to the Steepest Descent method. Rather than use the full calculated gradient for the weight, I randomly chose a sample from 1 to 100 and used that as my gradient for the whole function. This is far less computationally intensive but less likely to be accurate. I kept the line search method rather than fixed step size to offset the randomness that comes with this method. The results are okay, not amazing. With 1000 iterations I still have error on an order of 10^(-2), the worst among all methods.

```
ans =

  Columns 1 through 9

    0.0030    0.0030    0.0030    0.0030    0.0030    0.0030    0.0030    0.0030    0.0030
    0.0141    0.0141    0.0141    0.0141    0.0141    0.0141    0.0141    0.0141    0.0141

  Columns 10 through 18

    0.0030    0.0030    0.0030    0.0030    0.0030    0.0030    0.0030    0.0030    0.0030
    0.0141    0.0141    0.0141    0.0141    0.0141    0.0141    0.0141    0.0141    0.0141

  Columns 19 through 27
```

This method should be reserved for tasks that are far out of reach for more thorough algorithms due to computational complexity.

Conclusion

The fixed step size method is good for both accuracy and keeping some probability of discovering new local minimums. Further, small step sizes can guarantee higher degrees of accuracy. For the particular problem, alpha of .01 was very good.

The steepest descent method is also very good for quickly narrowing down on a local minimum. However, it may be useful to identify starting points likely to be near global

minimums if possible. Similar to Newton's method, this method can rabbit hole into less optimal locations quickly with no chance of escaping.

The stochastic gradient descent method has merit for problems of large complexity that cannot be dealt with using more exhaustive algorithms like fixed step or steepest descent. It is definitely useful for problems involving a very large number of variables because it significantly cuts computational time. Further, there are likely ways to improve upon its implementation, such as sampling a few gradient vectors and choosing a minimizer among them. As such, it is useful, but is not needed for the problem at hand.