Robustness of Newton Methods and Running Time Analysis

Last updated: June 26, 2021

Newton versus SG: Presentation I

- It is better to give figures showing time versus accuracy
- Some give a table listing

  accuracy and time
- But a problem is when to terminate the optimization procedure
- In fact this is an important issue in deep learning training
- By a figure we can more clearly see the trend

Newton versus SG: Performance I

- Most of you found that SG diverges if the learning rate is too large
- This is right
- Selecting the initial learning rate is a painful issue in using SG
- Therefore, Newton seems to be more robust
- However, for VGG11, its performance is slightly worse than SG
- Our experiences so far are that the best accuracy by Newton is sometimes not as good as SG when the number of layers is large

Newton versus SG: Performance II

- Lots of research still need to be done

Newton Running Time Analysis I

- For this part we would like to check if you understand some contents of our lectures
- We are running the same algorithm for both settings. Only implementation on Gauss-Newton matrix-vector products are different
- Thus # iterations and # CGs should be almost the same
- We ran 5 iterations on department workstation linux14 for two settings. Each way is run by 6 times. And we selected the one with shortest running time.

Newton Running Time Analysis II

- Here shows our profiling results (including log files) of two ways.
- Because function and gradient evaluations are the same, all we need to newly analyze is the CG time
- Thus checking total time isn't very useful
- Now let's focus on CG
- We can check the details in CG for two ways (not storing Jacobian and storing Jacobian). In both cases, the total # CGs within 5 iterations are

  72

Newton Running Time Analysis III

- The average # CGs per iteration is

$$14.4$$

- Theoretical ratio in a single layer per iteration

$$\frac{5\#\ \text{CG}}{n_{L+1} + 1 + 2\#\ \text{CG}}$$

- In our case, this ratio is

$$\frac{5 \times\ 14.4}{10 + 1 + 2 \times 14.4} = 1.80$$

- For timing, we got

Newton Running Time Analysis IV

- Jacobian not stored:
  - 975.1s for products, i.e., R_JTBJv()
- Jacobian stored:
  - 219.8s for construction, i.e., Jacobian()
  - 431.9s for products, i.e., JTBJv()
- The ratio is

$$\frac{975.1}{219.8 + 431.9} = 1.49$$

- Some may compare this ratio with 1.80 for checking the practical running time and theoretical complexity

Newton Running Time Analysis V

- This may not be appropriate as we have learned in proj 3 and proj 4, MATLAB has efficient matrix-matrix product while some other functions may not be well-optimized.
- For these two implementations, they may have different non-optimized operations
- So let's check the matrix-matrix product to verify the theoretical ratio.
- For simplicity, let's focus on CG steps only. Thus for the approach of storing Jacobian, we ignore the initial construction cost

Newton Running Time Analysis VI

- Thus the theoretical ratio is

$$\frac{5 \times \# \text{ CG}}{2 \times \# \text{ CG}} = 2.5$$

- If Jacobian is not stored, in R_JTBJv(), firstly we check the matrix-matrix product in Jv().
- We can observe that line 17 takes 46.5s:

```
net.Z{m+1} = max(model.weight{m}*net.phiZ{m}
+ model.bias{m}, 0);
```

and line 20 takes 102.9s:

Newton Running Time Analysis VII

```
R_Z = model.weight{m}*R_Z
+ v_(:, 1:end-1)*net.phiZ{m} + v_(:, end);
```

- For JTv(), this function is also called by lossgrad_subset().
- We see JTv() in R_JTBJv() took 400.7s.
- This divided by the total time of JTv() (i.e., 1045.9s) gives a ratio

$$\frac{400.7}{1045.9} \approx 0.38$$

Newton Running Time Analysis VIII

- We can use this ratio to estimate time of operations in JTv() that are related to R_JTBJv().
- With this ratio, we can estimate the time for matrix-matrix products in JTv(). Line 21 costs $199.3 * 0.38 = 75.7$s:

  ```
  JTv_{m} = [v*net.phiZ{m}' sum(v, 2)];
  ```

  while line 24 costs $96.5 * 0.38 = 36.6$s:

  ```
  v = model.weight{m}' * v;
  ```

- So the matrix-matrix product in R_JTBJv() cost $46.5 + 102.9 + 75.7 + 36.6 = 261.8$s.

Newton Running Time Analysis IX

- If Jacobian is stored, in JTBJv(), line 78 takes 37.3s:

  ```
  p = p(:, 1:end-1)*net.phiZ{m} + p(:, end);
  ```

  while line 108 takes 84.3s:

  ```
  u_m = [u_m*net.phiZ{m}' sum(u_m, 2)];
  ```

  Their sum is $37.3 + 84.3 = 121.6$s.
- Therefore, the practical ratio of two ways involving matrix-matrix product is:

$$\frac{261.8}{121.6} \approx 2.15$$

Newton Running Time Analysis X

- In this way, it seems that the practical ratio is roughly consistent with theoretical ratio.
- We can see that other operations, due to inefficient implementations, may take more time than matrix-matrix products.
- For example, in JTBJv(), you can see the most time-consuming part is line 79:

```
p = sum(reshape(net.dzdS{m}, d*ab, nL,
[]) .* reshape(p, d*ab, 1, []),1);
```

  let's check the following our course slides

From Course Slides I

- To get

$$\begin{bmatrix} \frac{\partial \boldsymbol{z}^{L+1,1}}{\partial \text{vec}(S^{m,1})^T} \boldsymbol{p}^{m,1} \\ \vdots \\ \frac{\partial \boldsymbol{z}^{L+1,l}}{\partial \text{vec}(S^{m,l})^T} \boldsymbol{p}^{m,l} \end{bmatrix},$$

  we need l matrix-vector products
- There is no good way to transform it to matrix-matrix operations

From Course Slides II

- At this moment we calculate

$$J^{m,i}\boldsymbol{v}^m = \frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(S^{m,i})^T}\boldsymbol{p}^{m,i}, \quad i = 1,\ldots,l. \tag{1}$$

  by summing up all rows of the following matrix

$$\left[\frac{\partial z_1^{L+1,i}}{\partial \mathrm{vec}(S^{m,i})} \cdots \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial \mathrm{vec}(S^{m,i})}\right]_{d^{m+1}a_{\mathrm{conv}}^m b_{\mathrm{conv}}^m \times n_{L+1}} \odot$$

$$\left[\boldsymbol{p}^{m,i} \cdots \boldsymbol{p}^{m,i}\right]_{d^{m+1}a_{\mathrm{conv}}^m b_{\mathrm{conv}}^m \times n_{L+1}}.$$

  and extend this to cover all instances together