# Newton Methods for Neural Networks: Algorithm

Chih-Jen Lin
National Taiwan University

Last updated: May 25, 2020

# Outline



# Outline



# Hessian-free Newton Method I

- Recall that at each Newton iteration we must solve a linear system

$$G\boldsymbol{d} = -\nabla f(\boldsymbol{\theta})$$

  and $G$ is huge
- $G$'s size is

$$n \times n,$$

  where $n$ is the total number of variables
- It is not possible to store $G$

# Hessian-free Newton Method II

- Thus methods such as Gaussian elimination are not possible
- If $G$ has certain structures, it's possible to use iterative methods to solve the linear system by a sequence of matrix-vector products

$$G\boldsymbol{v}^1, G\boldsymbol{v}^2, \ldots$$

without storing $G$

- This is called Hessian-free in optimization

# Hessian-free Newton Method III

- For example, conjugate gradient (CG) method can be used to solve

$$G\boldsymbol{d} = -\nabla f(\boldsymbol{\theta})$$

  by a sequence of matrix-vector products (Hestenes and Stiefel, 1952)
- We don't discuss details of CG here though the procedure will be shown in a later slide
- You can check Golub and Van Loan (2012) for a good introduction

# Hessian-free Newton Method IV

- For many machine learning methods, $G$ has certain structures
- The cost of Hessian-free Newton is

$$(\#\text{matrix-vector products} + \#\text{function/gradient evaluation}) \times \#\text{iterations}$$

- Usually the number of iterations is small
- At each iteration, to solve the linear system, several matrix-vector products are needed
- In theory, the number of CG steps (matrix-vector products) is $\leq$ the number of variables

# Hessian-free Newton Method V

- Each can be as expensive as one function/gradient evaluation
- Thus, matrix-vector products can be the bottleneck

# Conjugate Gradient Method I

- We would like to solve

$$Ax = b,$$

where $A$ is symmetric positive definite

- The procedure

$k = 0$; $x = 0$; $r = b$; $\rho_0 = \|r\|_2^2$
while $\sqrt{\rho_k} > \epsilon\|b\|_2$ and $k < k_{\max}$
$k = k + 1$
if $k = 1$
$p = r$

# Conjugate Gradient Method II

else

$$\beta = \rho_{k-1}/\rho_{k-2}$$

$$p = r + \beta p$$

end

$$w = Ap$$

$$\alpha = \rho_{k-1}/p^T w$$

$$x = x + \alpha p$$

$$r = r - \alpha w$$

$$\rho_k = \|r\|_2^2$$

end

# Matrix-vector Products I

- Earlier we have shown that the Gauss-Newton matrix is

$$G = \frac{1}{C}\mathcal{I} + \frac{1}{l}\sum_{i=1}^{l}(J^i)^T B^i J^i$$

- We have

$$G\boldsymbol{v} = \frac{1}{C}\boldsymbol{v} + \frac{1}{l}\sum_{i=1}^{l}\left((J^i)^T\left(B^i(J^i\boldsymbol{v})\right)\right). \quad (1)$$

# Matrix-vector Products II

- If we can calculate

$$J^i \boldsymbol{v} \text{ and } (J^i)^T(\cdot)$$

  then $G$ is never explicitly stored
- Therefore, we can apply the conjugate gradient (CG) method by a sequence of matrix-vector products.
- But is this approach really feasible?
- We show that memory can be an issue

# Memory Cost of Storing $J^i$ I

- The Gauss-Newton matrix is

$$G = \frac{1}{C}\mathcal{I} + \frac{1}{\ell}\sum_{i=1}^{\ell}(J^i)^T B^i J^i$$

- Its size is

$$n \times n,$$

where $n$ is the total number of variables

# Memory Cost of Storing $J^i$ II

- But storing $J^i$ needs

$$n_{L+1} \times n \times l,$$

where

$$n_{L+1} : \# \text{ nodes in the output layer } (\# \text{ classes})$$
$$l : \text{number of data}$$

- If

$$n < n_{L+1} \times l,$$

then storing $J^i, \forall i$ needs more spaces than $G$

# Memory Cost of Storing $J^i$ III

- Then the Hessian-free method cannot work
- A related question is why in calculating the gradient we didn't get $J^i$ and calculate

$$\nabla f(\boldsymbol{\theta})$$

$$= \frac{1}{C}\boldsymbol{\theta} + \frac{1}{l}\sum_{i=1}^{l}(J^i)^T \nabla_{\boldsymbol{z}^{L+1,i}} \xi(\boldsymbol{z}^{L+1,i}; \boldsymbol{y}^i, Z^{1,i})$$

- Instead we use backpropagation without explicitly storing $J^i, \forall i$
- For gradient, $J^i$ is used only once

# Memory Cost of Storing $J^i$ IV

- However, in each Newton iteration we need $J^i$ several times
- $J^i$ is used in every matrix-vector product
- Some techniques can be used to alleviate the memory problem of storing $J^i, \forall i$
  - Subsampled Hessian Newton method
  - Forward and reverse modes of automatic differentiation

# Outline



# Subsampled Hessian Newton Method I

- We know the gradient needs a sum over all data

$$\nabla f(\boldsymbol{\theta}) = \frac{1}{C}\boldsymbol{\theta} + \frac{1}{l}\sum_{i=1}^{l} \nabla_{\boldsymbol{\theta}} \xi_i$$

- In stochastic gradient, we do mini-batch
- Like mini-batch, in Newton we can use a subset of data for

matrix-vector products

and

function/gradient evaluation

# Subsampled Hessian Newton Method II

- This is possible: subsampled Newton method (Byrd et al., 2011; Martens, 2010; Wang et al., 2015)
- Assume the large number of data points are from the same distribution
- We can select a subset $S \subset \{1, \ldots, l\}$ and have

$$G^S = \frac{1}{C}\mathcal{I} + \frac{1}{|S|}\sum_{i \in S}(J^i)^T B^i J^i \approx G.$$

# Subsampled Hessian Newton Method III

- Then the matrix-vector product becomes

$$G^S \boldsymbol{v} = \frac{1}{C}\boldsymbol{v} + \frac{1}{|S|}\sum_{i\in S}\left((J^i)^T\left(B^i(J^i\boldsymbol{v})\right)\right) \qquad (2)$$

- The cost of storing $J^i$ is reduced from

$$n_{L+1} \times n \times l$$

to

$$n_{L+1} \times n \times |S|$$

# Subsampled Hessian Newton Method IV

- Typically a choice may be

$$|S| = (0.05 \text{ or } 0.01) \times l$$

- The selection of the size $|S|$ is still an issue worth investigation
- At this moment we consider

subset for matrix-vector products

and

full set for function/gradient evaluation

- Reason: no matter what a subset $S$ is chosen,

# Subsampled Hessian Newton Method V

$G^S$ is still positive definite

- Then

$$G^S \boldsymbol{d} = -\nabla f(\boldsymbol{\theta})$$

leads to

$$\nabla f(\boldsymbol{\theta})^T \boldsymbol{d} = -\nabla f(\boldsymbol{\theta})^T (G^S)^{-1} \nabla f(\boldsymbol{\theta}) < 0$$

- If we use a subset for the gradient, then the above inequality may not hold
- Then the situation becomes more complicated

# Subsampled Hessian Newton Method VI

- Note that if using the full set for function/gradient evaluations, we have theoretical asymptotic convergence to a stationary point (Byrd et al., 2011)

# Outline



# Levenberg-Marquardt method I

- Besides backtracking line search, in optimization another way to adjust the direction is the Levenberg-Marquardt method (Levenberg, 1944; Marquardt, 1963)
- It modifies the linear system to

$$(G^S + \lambda \mathcal{I})\boldsymbol{d} = -\nabla f(\boldsymbol{\theta})$$

- The value $\lambda$ is decided by how good the function reduction is.
- It's updated by the following settings.

# Levenberg-Marquardt method II

- If $\boldsymbol{\theta} + \boldsymbol{d}$ is the next iterate after line search, we define

$$\rho = \frac{f(\boldsymbol{\theta} + \boldsymbol{d}) - f(\boldsymbol{\theta})}{\nabla f(\boldsymbol{\theta})^T \boldsymbol{d} + \frac{1}{2} \boldsymbol{d}^T G^S \boldsymbol{d}}$$

  as the ratio of

$$\frac{\text{actual function reduction}}{\text{predicted reduction}}$$

# Levenberg-Marquardt method III

- By using $\rho$, the parameter $\lambda_{\text{next}}$ for the next iteration is decided by

$$\lambda_{\text{next}} = \begin{cases} \lambda \times \text{drop} & \rho > \rho_{\text{upper}}, \\ \lambda & \rho_{\text{lower}} \leq \rho \leq \rho_{\text{upper}}, \\ \lambda \times \text{boost} & \text{otherwise}, \end{cases}$$

  where

$$\text{drop} < 1, \text{boost} > 1$$

  are given constants.

- In our code you can see

# Levenberg-Marquardt method IV

```
param.drop = 2/3;
param.boost = 3/2;
```

and

$$\rho_{\text{upper}} = 0.75, \rho_{\text{lower}} = 0.25$$

- If the function-value reduction is not satisfactory, $\lambda$ is enlarged and the resulting direction is closer to the negative gradient.
- In optimization practice, if backtracking line search has been applied, usually there is no need to apply this LM method

# Levenberg-Marquardt method V

- However, some past works (e.g., Martens, 2010; Wang et al., 2018) on fully-connected networks seem to show that applying both is useful
- The use of LM in training NN is still an issue to be investigated

# Function and Gradient Evaluation I

- Recall in gradient evaluation the following main steps are conducted:

$$\Delta \leftarrow \text{mat}(\text{vec}(\Delta)^T P^{m,i}_{\text{pool}})$$

$$\frac{\partial \xi_i}{\partial W^m} = \Delta \cdot \phi(\text{pad}(Z^{m,i}))^T$$

$$\Delta \leftarrow \text{vec}\left((W^m)^T \Delta\right)^T P^m_\phi P^m_{\text{pad}}$$

$$\Delta \leftarrow \Delta \odot I[Z^{m,i}]$$

# Function and Gradient Evaluation II

- Clearly we must store $Z_i$, or even $\phi(\text{pad}(Z^{m,i}), \forall i$ after the forward process.
- This is fine for stochastic gradient as we use a small batch of data
- However, for Newton we need the full gradient so we can check the sufficient decrease condition
- The memory cost is then

$$\propto \ \# \text{ total data}$$

- This is not feasible

# Function and Gradient Evaluation III

- Fortunately we can calculate the gradient by the sum of sub-gradients

$$\frac{\partial f}{\partial W^m} = \frac{1}{C} W^m + \frac{1}{l} \sum_{i=1}^{l} \frac{\partial \xi_i}{\partial W^m}, \tag{3}$$

$$\frac{\partial f}{\partial \boldsymbol{b}^m} = \frac{1}{C} \boldsymbol{b}^m + \frac{1}{l} \sum_{i=1}^{l} \frac{\partial \xi_i}{\partial \boldsymbol{b}^m}. \tag{4}$$

- Thus we can split the index set $\{1, \ldots, l\}$ of data to, for example, $R$ equal-sized subsets $S_1, \ldots, S_R$

# Function and Gradient Evaluation IV

- We sequentially calculate the result corresponding to each subset and accumulate them for the final output.
- For example, to have $Z^{m,i}$ needed in the backward process for calculating the gradient, we must store them after the forward process for function evaluation.
- By using a subset, only $Z^{m,i}$ with $i$ in this subset are stored, so the memory usage can be dramatically reduced.

# The Overall Procedure I

- Let's check the Newton method code at `https://github.com/cjlin1/simpleNN/blob/master/MATLAB/opt/newton.m`

# Discussion I

- We have known that at each iteration

$$G^S = \frac{1}{C}\mathcal{I} + \frac{1}{|S|}\sum_{i\in S}(J^i)^T B^i J^i$$

  is considered
- The remaining issues are
  - How to calculate

$$J^i, \forall i \in S$$

  - How to calculate

$$(J^i)^T \left(B^i (J^i \boldsymbol{v})\right)$$