Newton Methods for Neural Networks: Gauss Newton Matrix-vector Product

Chih-Jen Lin
National Taiwan University

Outline

1. Backward setting
   - Jacobian evaluation
   - Gauss-Newton Matrix-vector products
2. Forward + backward settings
   - R operator
   - Gauss-Newton matrix-vector product
3. Complexity analysis

Outline

1. Backward setting
   - Jacobian evaluation
   - Gauss-Newton Matrix-vector products
2. Forward + backward settings
   - R operator
   - Gauss-Newton matrix-vector product
3. Complexity analysis

Outline

1. Backward setting
   - Jacobian evaluation
   - Gauss-Newton Matrix-vector products
2. Forward + backward settings
   - R operator
   - Gauss-Newton matrix-vector product
3. Complexity analysis

Jacobian Evaluation: Convolutional Layer I

- For an instance i the Jacobian can be partitioned into L blocks according to layers

$$J^i = \begin{bmatrix} J^{1,i} & J^{2,i} & \dots & J^{L,i} \end{bmatrix}, \; m = 1, \dots, L, \tag{1}$$

where

$$J^{m,i} = \begin{bmatrix} \frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(W^m)^T} & \frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{b}^m)^T} \end{bmatrix}.$$

- The calculation seems to be very similar to that for the gradient.

Jacobian Evaluation: Convolutional Layer II

- For the convolutional layers, recall for gradient we have

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

and

$$\frac{\partial \xi_i}{\partial \text{vec}(W^m)^T} = \text{vec}\left(\frac{\partial \xi_i}{\partial S^{m,i}} \phi(\text{pad}(Z^{m,i}))^T \right)^T$$

Jacobian Evaluation: Convolutional Layer III

- Now we have

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \text{vec}(W^m)^T} = \begin{bmatrix} \frac{\partial z_1^{L+1,i}}{\partial \text{vec}(W^m)^T} \\ \vdots \\ \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial \text{vec}(W^m)^T} \end{bmatrix}$$

$$= \begin{bmatrix} \text{vec}\left(\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}} \phi(\text{pad}(Z^{m,i}))^T\right)^T \\ \vdots \\ \text{vec}\left(\frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}} \phi(\text{pad}(Z^{m,i}))^T\right)^T \end{bmatrix}$$

Jacobian Evaluation: Convolutional Layer IV

- If $\boldsymbol{b}^m$ is considered, the result is

$$
\begin{bmatrix} \frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \text{vec}(W^m)^T} & \frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{b}^m)^T} \end{bmatrix}
= \begin{bmatrix} \text{vec}\left(\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}} \left[\phi(\text{pad}(Z^{m,i}))^T \ \mathbb{1}_{a^m_{\text{conv}} b^m_{\text{conv}}} \right] \right)^T \\ \vdots \\ \text{vec}\left(\frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}} \left[\phi(\text{pad}(Z^{m,i}))^T \ \mathbb{1}_{a^m_{\text{conv}} b^m_{\text{conv}}} \right] \right)^T \end{bmatrix}.
$$

Jacobian Evaluation: Convolutional Layer V

- We can see that it's more complicated than gradient.
- Gradient is a vector but Jacobian is a matrix

Jacobian Evaluation: Backward Process I

- For gradient, earlier we need a backward process to calculate

$$\frac{\partial \xi_i}{\partial S^{m,i}}$$

- Now what we need are

$$\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}}, \ldots, \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}}$$

- The process is similar

Jacobian Evaluation: Backward Process II

- If with RELU activation function and max pooling, for gradient we had

$$
\begin{aligned}
&\frac{\partial \xi_i}{\partial \text{vec}(S^{m,i})^T} \\
&= \left(\frac{\partial \xi_i}{\partial \text{vec}(Z^{m+1,i})^T} \odot \text{vec}(I[Z^{m+1,i}])^T \right) P_{\text{pool}}^{m,i}.
\end{aligned}
$$

Jacobian Evaluation: Backward Process III

- Assume that

$$\frac{\partial z^{L+1,i}}{\partial \mathrm{vec}(Z^{m+1,i})}$$

are available.

$$\frac{\partial z_j^{L+1,i}}{\partial \mathrm{vec}(S^{m,i})^T}$$
$$= \left(\frac{\partial z_j^{L+1,i}}{\partial \mathrm{vec}(Z^{m+1,i})^T} \odot \mathrm{vec}(I[Z^{m+1,i}])^T \right) P_{\text{pool}}^{m,i},$$
$$j = 1, \ldots, n_{L+1}.$$

Jacobian Evaluation: Backward Process IV

- These row vectors can be written together as a matrix

$$
\begin{aligned}
&\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \text{vec}(S^{m,i})^T} \\
= &\left(\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \text{vec}(Z^{m+1,i})^T} \odot \left(\mathbb{1}_{n_{L+1}} \text{vec}(I[Z^{m+1,i}])^T \right) \right) P^{m,i}_{\text{pool}}.
\end{aligned}
$$

- Note that

$$
\mathbb{1}_{n_{L+1}} \text{vec}(I[Z^{m+1,i}])^T
$$

duplicates the $\text{vec}(I[Z^{m+1,i}])^T$ vector n_{L+1} times

Jacobian Evaluation: Backward Process V

- For gradient, we use

$$\frac{\partial \xi_i}{\partial S^{m,i}}$$

to have

$$\frac{\partial \xi_i}{\partial \mathrm{vec}(Z^{m,i})^T} = \mathrm{vec}\left((W^m)^T \frac{\partial \xi_i}{\partial S^{m,i}} \right)^T P_\phi^m P_{\mathrm{pad}}^m$$

and pass it to the previous layer

Jacobian Evaluation: Backward Process VI

- Now we need to generate

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \text{vec}(Z^{m,i})^T}$$

  and pass it to the previous layer.
- Now we have

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \text{vec}(Z^{m,i})^T} = \begin{bmatrix} \text{vec}\left((W^m)^T \frac{\partial z_1^{L+1,i}}{\partial S^{m,i}} \right)^T P_\phi^m P_{\text{pad}}^m \\ \vdots \\ \text{vec}\left((W^m)^T \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}} \right)^T P_\phi^m P_{\text{pad}}^m \end{bmatrix}.$$

Jacobian Evaluation: Fully-connected Layer I

- We do not discuss details, but list all results below

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \text{vec}(W^m)^T} =$$

$$\begin{bmatrix} \text{vec}\left(\frac{\partial z_1^{L+1,i}}{\partial \boldsymbol{s}^{m,i}}(\boldsymbol{z}^{m,i})^T\right)^T \\ \vdots \\ \text{vec}\left(\frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial \boldsymbol{s}^{m,i}}(\boldsymbol{z}^{m,i})^T\right)^T \end{bmatrix}$$

Jacobian Evaluation: Fully-connected Layer II

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{b}^m)^T} = \frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{s}^{m,i})^T},$$

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{s}^{m,i})^T} = \frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{z}^{m+1,i})^T} \odot \left(\mathbb{1}_{n_{L+1}} I[\boldsymbol{z}^{m+1,i}]^T\right)$$

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{z}^{m,i})^T} = \frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{s}^{m,i})^T} W^m$$

Jacobian Evaluation: Fully-connected Layer III

- For the layer $L + 1$, if using a linear activation function with

  $$\boldsymbol{z}^{L+1,i} = \boldsymbol{s}^{L,i},$$

  then we have

  $$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{s}^{L,i})^T} = \mathcal{I}_{n_{L+1}}.$$

Gradient versus Jacobian I

- Operations for gradient

$$
\begin{aligned}
&\frac{\partial \xi_i}{\partial \text{vec}(S^{m,i})^T} \\
&= \left(\frac{\partial \xi_i}{\partial \text{vec}(Z^{m+1,i})^T} \odot \text{vec}(I[Z^{m+1,i}])^T \right) P_{\text{pool}}^{m,i}.
\end{aligned}
$$

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

$$
\frac{\partial \xi_i}{\partial \text{vec}(Z^{m,i})^T} = \text{vec}\left((W^m)^T \frac{\partial \xi_i}{\partial S^{m,i}} \right)^T P_\phi^m P_{\text{pad}}^m,
$$

Gradient versus Jacobian II

- For Jacobian we have

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(S^{m,i})^T}$$
$$= \left(\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(Z^{m+1,i})^T} \odot \left(\mathbb{1}_{n_{L+1}} \mathrm{vec}(I[Z^{m+1,i}])^T \right) \right) P_{\mathrm{pool}}^{m,i}.$$

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(W^m)^T} = \begin{bmatrix} \mathrm{vec}\left(\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}} \phi(\mathrm{pad}(Z^{m,i}))^T\right)^T \\ \vdots \\ \mathrm{vec}\left(\frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}} \phi(\mathrm{pad}(Z^{m,i}))^T\right)^T \end{bmatrix}$$

Gradient versus Jacobian III

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \text{vec}(Z^{m,i})^T} = \begin{bmatrix} \text{vec}\left((W^m)^T \frac{\partial z_1^{L+1,i}}{\partial S^{m,i}}\right)^T P_\phi^m P_{\text{pad}}^m \\ \vdots \\ \text{vec}\left((W^m)^T \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}}\right)^T P_\phi^m P_{\text{pad}}^m \end{bmatrix}. \tag{2}$$

Implementation I

- For gradient we did

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

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

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

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

- Now for Jacobian we have similar settings but there are some differences

Implementation II

- We do not really store the Jacobian:

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(W^m)^T} = \begin{bmatrix} \mathrm{vec}(\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}} \phi(\mathrm{pad}(Z^{m,i}))^T)^T \\ \vdots \\ \mathrm{vec}(\frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}} \phi(\mathrm{pad}(Z^{m,i}))^T)^T \end{bmatrix}$$

- Recall Jacobian is used for matrix-vector products

$$G^S \boldsymbol{v} = \frac{1}{C}\boldsymbol{v} + \frac{1}{|S|}\sum_{i \in S}\left((J^i)^T\left(B^i(J^i\boldsymbol{v})\right)\right) \tag{3}$$

Implementation III

- The form

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \text{vec}(W^m)^T} = \begin{bmatrix} \text{vec}(\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}} \phi(\text{pad}(Z^{m,i}))^T)^T \\ \vdots \\ \text{vec}(\frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}} \phi(\text{pad}(Z^{m,i}))^T)^T \end{bmatrix} \quad (4)$$

is like the product of two things

Implementation IV

- If we have

$$\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}}, \ldots, \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}}, \text{ and } \phi(\text{pad}(Z^{m,i}))$$

  probably we can do the matrix-vector product without multiplying these two things out
- We will talk about this again later

Implementation V

- We already know how to obtain

$$\phi(\text{pad}(Z^{m,i}))$$

so the remaining issue is on obtaining

$$\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}}, \ldots, \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}}$$

- Further we need to take all data (or data in the selected subset) into account
- In the end what we have is the following procedure

Implementation VI

- In the beginning we have

$$\Delta \in R^{d^{m+1}a^{m+1}b^{m+1} \times n_{L+1} \times l} \tag{5}$$

This corresponds to

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(Z^{m+1,i})^T} \odot \left(\mathbb{1}_{n_{L+1}} \mathrm{vec}(I[Z^{m+1,i}])^T \right), \forall i = 1, \ldots, l$$

Implementation VII

- We then calculate

$$\Delta \leftarrow \text{mat}\left(\begin{bmatrix} (P_{\text{pool}}^{m,1})^T \text{vec}(\Delta_{:,:,1}) \\ \vdots \\ (P_{\text{pool}}^{m,l})^T \text{vec}(\Delta_{:,:,l}) \end{bmatrix}\right)_{d^{m+1} \times a_{\text{conv}}^m b_{\text{conv}}^m n_{L+1} l}$$

- Recall that the pooling matrices are different across instances

Implementation VIII

- The above operation corresponds to

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \text{vec}(S^{m,i})^T}$$
$$= \left(\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \text{vec}(Z^{m+1,i})^T} \odot \left(\mathbb{1}_{n_{L+1}} \text{vec}(I[Z^{m+1,i}])^T \right) \right) P_{\text{pool}}^{m,i}.$$

- Now we get

$$\begin{bmatrix} \frac{\partial z_1^{L+1,1}}{\partial S^{m,1}} & \dots & \frac{\partial z_{n_{L+1}}^{L+1,1}}{\partial S^{m,1}} & \dots & \frac{\partial z_{n_{L+1}}^{L+1,l}}{\partial S^{m,l}} \end{bmatrix}$$
$$\in R^{d^{m+1} \times a_{\text{conv}}^m b_{\text{conv}}^m n_{L+1} l}$$

Implementation IX

- For gradient, the next step is to calculate

$$\frac{\partial \xi_i}{\partial W^m} = \cdots$$

  but here for Jacobian we have mentioned that we do not explicitly get

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \text{vec}(W^m)^T}$$

- Therefore, the next operation is

$$V \leftarrow \text{vec}((W^m)^T \Delta) \in R^{hhd^m a^m_{\text{conv}} b^m_{\text{conv}} n_{L+1} l \times 1}$$

Implementation X

- This is same as

$$\text{vec}\left((W^m)^T \begin{bmatrix} \frac{\partial z_1^{L+1,1}}{\partial S^{m,1}} & \cdots & \frac{\partial z_{n_{L+1}}^{L+1,1}}{\partial S^{m,1}} & \cdots & \frac{\partial z_{n_{L+1}}^{L+1,l}}{\partial S^{m,l}} \end{bmatrix}\right).$$

- Now V is a big vector like

$$\begin{bmatrix} \boldsymbol{v}_1^1 \\ \vdots \\ \boldsymbol{v}_{n_{L+1}}^1 \\ \vdots \\ \boldsymbol{v}_{n_{L+1}}^l \end{bmatrix}$$

Implementation XI

Note that "$\boldsymbol{v}$" here is not the vector in matrix-vector products. We happen to use the same symbol

- From (2), we then calculate

$$(\boldsymbol{v}_1^1)^T P_\phi^m P_{\text{pad}}^m$$

$$\vdots$$

$$(\boldsymbol{v}_{n_{L+1}}^1)^T P_\phi^m P_{\text{pad}}^m$$

$$\vdots$$

$$(\boldsymbol{v}_{n_{L+1}}^l)^T P_\phi^m P_{\text{pad}}^m$$

Implementation XII

- For each resulting vector, we convert it to

$$\text{mat}\left(\boldsymbol{v}^T P_\phi^m P_{\text{pad}}^m\right)_{d^m \times a^m b^m}$$

This corresponds to

$$\frac{\partial z_1^{L+1,i}}{\partial Z^{m,i}}, \ldots, \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial Z^{m,i}}, i = 1, \ldots, l$$

Implementation XIII

- Finally,

$$\Delta \leftarrow \Delta \odot \left[\underbrace{I[Z^{m,1}] \cdots I[Z^{m,1}]}_{n_{L+1}} \cdots \underbrace{I[Z^{m,l}] \cdots I[Z^{m,l}]}_{n_{L+1}} \right] \quad (6)$$

This is equivalent to

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \text{vec}(Z^{m,i})^T} \odot \left(\mathbb{1}_{n_{L+1}} \text{vec}(I[Z^{m,i}])^T \right), \forall i = 1, \ldots, l$$

Implementation XIV

- Note that in the beginning of the calculation, we assume that in (5)

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \text{vec}(Z^{m+1,i})^T} \odot \left(\mathbb{1}_{n_{L+1}} \text{vec}(I[Z^{m+1,i}])^T \right), \forall i = 1, \ldots, l$$

is available. The calculation here is to provide information for the previous layer

MATLAB Implementation I

```
dzdS{m} = vTP(model, net, m, num_data,
          dzdS{m}, 'pool_Jacobian');

dzdS{m} = reshape(dzdS{m},
          model.ch_input(m+1), []);

V = model.weight{m}' * dzdS{m};
dzdS{m-1} = vTP(model, net, m, num_data,
            V, 'phi_Jacobian');

% vTP_pad
```

MATLAB Implementation II

```
dzdS{m-1} = reshape(dzdS{m-1},
  model.ch_input(m), model.ht_pad(m),
  model.wd_pad(m), []);
p = model.wd_pad_added(m);
dzdS{m-1} = dzdS{m-1}(:, p+1:p+model.ht_input(m
  p+1:p+model.wd_input(m), :);

dzdS{m-1} =
  reshape(dzdS{m-1}, [], nL, num_data)
  .* reshape(net.Z{m} > 0, [], 1, num_data);
```

MATLAB Implementation III

- In the last line for doing (6), we do not need to repeat each $I[Z^{m,i}]$ n_{L+1} times. For `.*`, MATLAB does the expansion automatically

Discussion I

- For doing several CG steps, we should store

$$\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}}, \ldots, \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}}, i = 1, \ldots, l$$

  in (4).
- The reason is that it's used for all CG steps (Jacobian matrix remains the same)
- Recalculating them at each CG step is too expensive

Discussion II

- The memory cost is

$$l \times n_{L+1} \times \left(\sum_{m=1}^{L^c} d^{m+1} a^m_{\text{conv}} b^m_{\text{conv}} + \sum_{m=L^c+1}^{L} n_{m+1} \right) \quad (7)$$

- It is proportional to
  - Number of classes
  - Number of data for the subsampled Hessian
- This memory cost is high
- Thus later we will consider a different approach to reduce the memory consumption

Outline

1. Backward setting
   - Jacobian evaluation
   - Gauss-Newton Matrix-vector products
2. Forward + backward settings
   - R operator
   - Gauss-Newton matrix-vector product
3. Complexity analysis

Gauss-Newton Matrix-Vector Products I

- We check

$$Gv$$

  though the situation of using G^S (i.e., a subset of data) is the same
- The Gauss-Newton matrix is

$$G = \frac{1}{C}\mathcal{I} + \frac{1}{l}\sum_{i=1}^{l} \begin{bmatrix} (J^{1,i})^T \\ \vdots \\ (J^{L,i})^T \end{bmatrix} B^i \begin{bmatrix} J^{1,i} & \dots & J^{L,i} \end{bmatrix}$$

Gauss-Newton Matrix-Vector Products II

- The Gauss-Newton matrix vector product is

$$
\begin{aligned}
& G\boldsymbol{v} \\
= & \frac{1}{C}\boldsymbol{v} + \frac{1}{l}\sum_{i=1}^{l} \begin{bmatrix} \left(J^{1,i}\right)^T \\ \vdots \\ \left(J^{L,i}\right)^T \end{bmatrix} B^i \begin{bmatrix} J^{1,i} & \dots & J^{L,i} \end{bmatrix} \begin{bmatrix} \boldsymbol{v}^1 \\ \vdots \\ \boldsymbol{v}^L \end{bmatrix} \\
= & \frac{1}{C}\boldsymbol{v} + \frac{1}{l}\sum_{i=1}^{l} \begin{bmatrix} \left(J^{1,i}\right)^T \\ \vdots \\ \left(J^{L,i}\right)^T \end{bmatrix} \left(B^i \sum_{m=1}^{L} J^{m,i}\boldsymbol{v}^m \right),
\end{aligned} \tag{8}
$$

Gauss-Newton Matrix-Vector Products III

where

$$\boldsymbol{v} = \begin{bmatrix} \boldsymbol{v}^1 \\ \vdots \\ \boldsymbol{v}^L \end{bmatrix}$$

- Each $\boldsymbol{v}^m, m = 1, \ldots, L$ has the same length as the number of variables (including bias) at the mth layer.

Jacobian-vector Product I

- For the convolutional layers,

$$
\begin{aligned}
& J^{m,i} \boldsymbol{v}^m \\
= & \begin{bmatrix} \text{vec}\left(\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}} \left[\phi(\text{pad}(Z^{m,i}))^T \; \mathbb{1}_{a^m_{\text{conv}} b^m_{\text{conv}}} \right] \right)^T \boldsymbol{v}^m \\ \vdots \\ \text{vec}\left(\frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}} \left[\phi(\text{pad}(Z^{m,i}))^T \; \mathbb{1}_{a^m_{\text{conv}} b^m_{\text{conv}}} \right] \right)^T \boldsymbol{v}^m \end{bmatrix} \\
\in & R^{n_{L+1} \times 1}
\end{aligned}
$$

- By this formulation, we need

Jacobian-vector Product II

- a for loop to generate n_{L+1} vectors

$$\text{vec}\left(\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}}\left[\phi(\text{pad}(Z^{m,i}))^T \mathbb{1}_{a^m_{\text{conv}} b^m_{\text{conv}}}\right]\right)^T$$

$$\vdots$$

$$\text{vec}\left(\frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}}\left[\phi(\text{pad}(Z^{m,i}))^T \mathbb{1}_{a^m_{\text{conv}} b^m_{\text{conv}}}\right]\right)^T$$

- the product between the above matrix and a vector $\boldsymbol{v}^m$

Jacobian-vector Product III

- Is there a way to avoid a for loop?
- For a language like MATLAB/Octave, we hope to avoid for loops
- Also we hope the code can be simpler and shorter
- We use the following property

$$\text{vec}(AB)^T\text{vec}(C) = \text{vec}(A)^T\text{vec}(CB^T)$$

Jacobian-vector Product IV

- The first element is

$$\mathrm{vec}\left(\underbrace{\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}}}_{A}\underbrace{\left[\phi(\mathrm{pad}(Z^{m,i}))^T \ \mathbb{1}_{a_{\mathrm{conv}}^m b_{\mathrm{conv}}^m}\right]}_{B}\right)^T \underbrace{\boldsymbol{v}^m}_{\mathrm{vec}(C)}$$

$$= \frac{\partial z_1^{L+1,i}}{\partial \mathrm{vec}(S^{m,i})^T} \times$$

$$\mathrm{vec}\left(\mathrm{mat}(\boldsymbol{v}^m)_{d^{m+1}\times(h^m h^m d^m+1)}\begin{bmatrix}\phi(\mathrm{pad}(Z^{m,i}))\\ \mathbb{1}_{a_{\mathrm{conv}}^m b_{\mathrm{conv}}^m}^T\end{bmatrix}\right).$$

Jacobian-vector Product V

- If all elements are considered together

$$
\begin{aligned}
& J^{m,i} \boldsymbol{v}^m \\
= & \frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \text{vec}(S^{m,i})^T} \times \\
& \text{vec}\left(\text{mat}(\boldsymbol{v}^m)_{d^{m+1} \times (h^m h^m d^m + 1)} \begin{bmatrix} \phi(\text{pad}(Z^{m,i})) \\ \mathbb{1}^T_{a^m_{\text{conv}} b^m_{\text{conv}}} \end{bmatrix} \right).
\end{aligned} \tag{9}
$$

This involves

- One matrix-matrix product
- One matrix-vector product

Transposed Jacobian-vector Products I

- After deriving (9), from (8), we sum results of all layers

$$\sum_{m=1}^{L} J^{m,i} \boldsymbol{v}^m$$

- Next we calculate

$$\boldsymbol{q}^i = B^i \left(\sum_{m=1}^{L} J^{m,i} \boldsymbol{v}^m\right). \tag{10}$$

- This is usually easy

Transposed Jacobian-vector Products II

- We mentioned earlier that if the squared loss is used

$$B^i = \begin{bmatrix} 2 & & \\ & \vdots & \\ & & 2 \end{bmatrix}$$

is a diagonal matrix

Transposed Jacobian-vector Products III

- Finally, we calculate

$$
\begin{aligned}
&(J^{m,i})^T \boldsymbol{q}^i \\
=&\Bigg[\text{vec}\left(\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}}\left[\phi(\text{pad}(Z^{m,i}))^T \ \mathbb{1}_{a^m_{\text{conv}} b^m_{\text{conv}}}\right]\right)\cdots \\
&\text{vec}\left(\frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}}\left[\phi(\text{pad}(Z^{m,i}))^T \ \mathbb{1}_{a^m_{\text{conv}} b^m_{\text{conv}}}\right]\right)\Bigg]\boldsymbol{q}^i
\end{aligned}
$$

Transposed Jacobian-vector Products IV

$$
\begin{aligned}
&= \sum_{j=1}^{n_{L+1}} q_j^i \text{vec}\left(\frac{\partial z_j^{L+1,i}}{\partial S^{m,i}} \left[\phi(\text{pad}(Z^{m,i}))^T \mathbb{1}_{a_{\text{conv}}^m b_{\text{conv}}^m} \right] \right) \\
&= \text{vec}\left(\sum_{j=1}^{n_{L+1}} q_j^i \left(\frac{\partial z_j^{L+1,i}}{\partial S^{m,i}} \left[\phi(\text{pad}(Z^{m,i}))^T \mathbb{1}_{a_{\text{conv}}^m b_{\text{conv}}^m} \right] \right) \right) \\
&= \text{vec}\left(\left(\sum_{j=1}^{n_{L+1}} q_j^i \frac{\partial z_j^{L+1,i}}{\partial S^{m,i}} \right) \left[\phi(\text{pad}(Z^{m,i}))^T \mathbb{1}_{a_{\text{conv}}^m b_{\text{conv}}^m} \right] \right)
\end{aligned}
$$

Transposed Jacobian-vector Products V

$$= \operatorname{vec}\left(\operatorname{mat}\left(\left(\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \operatorname{vec}(S^{m,i})^T}\right)^T \boldsymbol{q}^i\right)_{d^{m+1} \times a_{\text{conv}}^m b_{\text{conv}}^m} \times \left[\phi(\operatorname{pad}(Z^{m,i}))^T \, \mathbb{1}_{a_{\text{conv}}^m b_{\text{conv}}^m}\right]\right). \tag{11}$$

This needs a matrix-vector product and then a matrix-matrix product

Fully-connected Layers I

- Similar to the results of the convolutional layers, for the fully-connected layers we have

$$J^{m,i}\boldsymbol{v}^m = \frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{s}^{m,i})^T} \text{mat}(\boldsymbol{v}^m)_{n_{m+1}\times(n_m+1)} \begin{bmatrix} \boldsymbol{z}^{m,i} \\ \mathbb{1}_1 \end{bmatrix}.$$

$$(J^{m,i})^T \boldsymbol{q}^i = \text{vec}\left(\left(\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{s}^{m,i})^T} \right)^T \boldsymbol{q}^i \left[(\boldsymbol{z}^{m,i})^T \ \mathbb{1}_1 \right] \right).$$

Implementation I

- As before, we must handle all instances together
- We discuss only

$$\begin{bmatrix} \sum_{m=1}^{L} J^{m,1} \boldsymbol{v}^m \\ \vdots \\ \sum_{m=1}^{L} J^{m,l} \boldsymbol{v}^m \end{bmatrix} \in R^{n_{L+1}l \times 1}$$

- Following earlier derivation

Implementation II

$$
\begin{bmatrix} J^{m,1}\boldsymbol{v}^m \\ \vdots \\ J^{m,l}\boldsymbol{v}^m \end{bmatrix}
= \begin{bmatrix}
\frac{\partial \boldsymbol{z}^{L+1,1}}{\partial \mathrm{vec}(S^{m,1})^T} \mathrm{vec}\left(\mathrm{mat}(\boldsymbol{v}^m) \begin{bmatrix} \phi(\mathrm{pad}(Z^{m,1})) \\ \mathbb{1}^T_{a^m_{\mathrm{conv}} b^m_{\mathrm{conv}}} \end{bmatrix} \right) \\
\vdots \\
\frac{\partial \boldsymbol{z}^{L+1,l}}{\partial \mathrm{vec}(S^{m,l})^T} \mathrm{vec}\left(\mathrm{mat}(\boldsymbol{v}^m) \begin{bmatrix} \phi(\mathrm{pad}(Z^{m,l})) \\ \mathbb{1}^T_{a^m_{\mathrm{conv}} b^m_{\mathrm{conv}}} \end{bmatrix} \right)
\end{bmatrix}
$$

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

Implementation III

- where

$$\text{mat}(\boldsymbol{v}^m) \in R^{d^{m+1} \times (h^m h^m d^m + 1)}$$

and

$$\boldsymbol{p}^{m,i} = \text{vec}\left(\text{mat}(\boldsymbol{v}^m) \begin{bmatrix} \phi(\text{pad}(Z^{m,i})) \\ \mathbb{1}^T_{a^m_{\text{conv}} b^m_{\text{conv}}} \end{bmatrix}\right). \qquad (12)$$

Implementation IV

- To get $\boldsymbol{p}^{m,i}$, a matrix-matrix product is needed. For all $i = 1, \ldots, l$ the calculation can be done by a matrix-matrix product

$$\text{mat}(\boldsymbol{v}^m) \begin{bmatrix} \phi(\text{pad}(Z^{m,1})) & \cdots & \phi(\text{pad}(Z^{m,l})) \\ \mathbb{1}^T_{a^m_{\text{conv}} b^m_{\text{conv}}} & \cdots & \mathbb{1}^T_{a^m_{\text{conv}} b^m_{\text{conv}}} \end{bmatrix}$$
$$\in R^{d^{m+1} \times a^m_{\text{conv}} b^m_{\text{conv}} l}$$

Implementation V

- 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

Implementation VI

- To avoid a for loop over all data, here we implement the matrix-vector product

$$J^{m,i}\boldsymbol{v}^m = \frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(S^{m,i})^T}\boldsymbol{p}^{m,i} \tag{13}$$

by a summation of all rows of the following matrix

$$\begin{bmatrix} \frac{\partial z_1^{L+1,i}}{\partial \mathrm{vec}(S^{m,i})} \cdots \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial \mathrm{vec}(S^{m,i})} \end{bmatrix}_{d^{m+1}a^m_{\mathrm{conv}}b^m_{\mathrm{conv}}\times n_{L+1}} \odot$$

$$\begin{bmatrix} \boldsymbol{p}^{m,i} \cdots \boldsymbol{p}^{m,i} \end{bmatrix}_{d^{m+1}a^m_{\mathrm{conv}}b^m_{\mathrm{conv}}\times n_{L+1}}.$$

Implementation VII

- For example, summing up all elements of the first column is the inner product between the first row of

$$\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(S^{m,i})^T} \text{ and } \boldsymbol{p}^{m,i}$$

- Then all the l matrix-vector products

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

Implementation VIII

can be done in one line by

$$\left[\cdots \frac{\partial z_1^{L+1,i}}{\partial \text{vec}(S^{m,i})} \cdots \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial \text{vec}(S^{m,i})} \cdots \right] \odot$$

$$\left[\cdots \boldsymbol{p}^{m,i} \cdots \boldsymbol{p}^{m,i} \cdots \right]$$

- The code (convolutional layers) is like

Implementation IX

```
for m = LC : -1 : 1
  var_range = var_ptr(m) : var_ptr(m+1) - 1;
  ab = model.ht_conv(m)*model.wd_conv(m);
  d = model.ch_input(m+1);

  p = reshape(v(var_range), d, []) *
      [net.phiZ{m}; ones(1, ab*num_data)];
  p = sum(reshape(net.dzdS{m}, d*ab, nL,
          []) .*
          reshape(p, d*ab, 1, []),1);
  Jv = Jv + p(:);
```

Implementation X

```
end
```

- Note that

  ```
  sum(:,1);
  ```

  sums up all rows

- For $\boldsymbol{p}^{m,i}$ we do not duplicate it n_{L+1} times. Instead, for `.*`, MATLAB does the expansion automatically

Outline

Outline

Reverse versus Forward Autodiff I

- We mentioned before that two types of autodiff are

  forward and reverse modes
- For the Jacobian evaluation, at layer m,

$$J^{m,i} = \left[\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \mathrm{vec}(W^m)^T} \quad \frac{\partial \boldsymbol{z}^{L+1,i}}{\partial (\boldsymbol{b}^m)^T} \right],$$

  naturally we follow the gradient calculation to use the reverse mode
- But this may not be a good decision

Reverse versus Forward Autodiff II

- In particular, we must store $J^{m,i}, \forall i$, or more precisely,

$$\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}}, \ldots, \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}}, i = 1, \ldots, l,$$

where the memory cost is

$$l \times n_{L+1} \times \left(\sum_{m=1}^{L^c} d^{m+1} a_{\text{conv}}^m b_{\text{conv}}^m + \sum_{m=L^c+1}^{L} n_{m+1} \right)$$

This memory cost is higher than other stored information

Reverse versus Forward Autodiff III

- For example, the $Z^{m,i}, \forall i$ stored from the forward process takes

$$l \times \left(\sum_{m=1}^{L^c} d^m a^m b^m + \sum_{m=L^c+1}^{L+1} n_m \right),$$

which is independent to the number of classes.

- We will show a solution to address this memory difficulty
- First, for the Jacobian-vector product, we will use the forward mode of automatic differentiation

Reverse versus Forward Autodiff IV

- Recall earlier we said that by the forward mode, the Jacobian-vector product can be done in just one pass

R Operator I

- Consider $g(\boldsymbol{\theta}) \in R^{k\times 1}$. Following Pearlmutter (1994), we define

$$\mathcal{R}_{\boldsymbol{v}}\{g(\boldsymbol{\theta})\} \equiv \frac{\partial g(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}^T}\boldsymbol{v} = \begin{bmatrix} \nabla g_1(\boldsymbol{\theta})^T \boldsymbol{v} \\ \vdots \\ \nabla g_k(\boldsymbol{\theta})^T \boldsymbol{v} \end{bmatrix}. \tag{14}$$

- Note that

$$\begin{bmatrix} \nabla g_1(\boldsymbol{\theta})^T \\ \vdots \\ \nabla g_k(\boldsymbol{\theta})^T \end{bmatrix}$$

is the Jacobian of $g(\boldsymbol{\theta})$

R Operator II

- This definition can be extended to a matrix $M(\boldsymbol{\theta}) \in R^{k\times t}$ by

$$\mathcal{R}_{\boldsymbol{v}}\{M(\boldsymbol{\theta})\} \equiv \operatorname{mat}\left(\mathcal{R}_{\boldsymbol{v}}\{\operatorname{vec}(M(\boldsymbol{\theta}))\}\right)_{k\times t}$$

$$= \operatorname{mat}\left(\frac{\partial \operatorname{vec}(M(\boldsymbol{\theta}))}{\partial \boldsymbol{\theta}^T}\boldsymbol{v}\right)_{k\times t} = \begin{bmatrix} \nabla M_{11}^T\boldsymbol{v} & \cdots & \nabla M_{1t}^T\boldsymbol{v} \\ \vdots & \ddots & \vdots \\ \nabla M_{k1}^T\boldsymbol{v} & \cdots & \nabla M_{kt}^T\boldsymbol{v} \end{bmatrix}$$

- Clearly,

$$\mathcal{R}_{\boldsymbol{v}}\{M(\boldsymbol{\theta})\} = \left(\mathcal{R}_{\boldsymbol{v}}\{M(\boldsymbol{\theta})^T\}\right)^T. \quad (15)$$

R Operator III

- If $h(\cdot)$ is a scalar function, we let

$$h(M(\boldsymbol{\theta})) = \begin{bmatrix} h(M_{11}) & \cdots & h(M_{1t}) \\ \vdots & \ddots & \vdots \\ h(M_{k1}) & \cdots & h(M_{kt}) \end{bmatrix}$$

and

$$h'(M(\boldsymbol{\theta})) = \begin{bmatrix} h'(M_{11}) & \cdots & h'(M_{1t}) \\ \vdots & \ddots & \vdots \\ h'(M_{k1}) & \cdots & h'(M_{kt}) \end{bmatrix}.$$

R Operator IV

- Because

$$\nabla(h(M_{ij}(\boldsymbol{\theta})))^T \boldsymbol{v} = h'(M_{ij})\nabla(M_{ij})^T \boldsymbol{v},$$

we have

$$\begin{aligned}\mathcal{R}_{\boldsymbol{v}}\{h(M(\boldsymbol{\theta}))\} &= \begin{bmatrix} \nabla h(M_{11})^T \boldsymbol{v} & \cdots & \nabla h(M_{1t})^T \boldsymbol{v} \\ \vdots & \ddots & \vdots \\ \nabla h(M_{k1})^T \boldsymbol{v} & \cdots & \nabla h(M_{kt})^T \boldsymbol{v} \end{bmatrix} \\ &= h'(M(\boldsymbol{\theta})) \odot \mathcal{R}_{\boldsymbol{v}}\{M(\boldsymbol{\theta})\},\end{aligned} \tag{16}$$

R Operator V

where $\odot$ stands for the Hadamard product (i.e., component-wise product).

- If $M(\boldsymbol{\theta})$ and $T(\boldsymbol{\theta})$ have the same size,

$$\mathcal{R}_{\boldsymbol{v}}\{M(\boldsymbol{\theta}) + T(\boldsymbol{\theta})\} = \mathcal{R}_{\boldsymbol{v}}\{M(\boldsymbol{\theta})\} + \mathcal{R}_{\boldsymbol{v}}\{T(\boldsymbol{\theta})\}. \tag{17}$$

- Lastly, we have

$$\mathcal{R}_{\boldsymbol{v}}\{U(\boldsymbol{\theta})M(\boldsymbol{\theta})\} = \mathcal{R}_{\boldsymbol{v}}\{U(\boldsymbol{\theta})\}M(\boldsymbol{\theta}) + U(\boldsymbol{\theta})\mathcal{R}_{\boldsymbol{v}}\{M(\boldsymbol{\theta})\} \tag{18}$$

R Operator VI

Proof: Note that

$$(\mathcal{R}\{U(\boldsymbol{\theta})M(\boldsymbol{\theta})\})_{ij} = \nabla \left((U(\boldsymbol{\theta})M(\boldsymbol{\theta}))_{ij} \right)^T \boldsymbol{v}.$$

With

$$(U(\boldsymbol{\theta})M(\boldsymbol{\theta}))_{ij} = \sum_{p=1}^{m} U_{ip} M_{pj},$$

we have both $U_{ip} \in R^1$ and $M_{pj} \in R^1$. Then,

$$\nabla \left(U_{ip} M_{pj} \right)^T \boldsymbol{v} = \left((\nabla U_{ip})^T \boldsymbol{v} \right) M_{pj} + U_{ip} \left((\nabla M_{pj})^T \boldsymbol{v} \right).$$

R Operator VII

The summation

$$\sum_{p=1}^{m} \left((\nabla U_{ip})^T \boldsymbol{v} \right) M_{pj}$$

leads to the (i, j) component of

$$\mathcal{R}_{\boldsymbol{v}}\{U(\boldsymbol{\theta})\}M(\boldsymbol{\theta})$$

Thus we have (18)

- For simplicity, subsequently we use $\mathcal{R}\{g(\boldsymbol{\theta})\}$ to denote $\mathcal{R}_{\boldsymbol{v}}\{g(\boldsymbol{\theta})\}$

R Operator for $J^i \boldsymbol{v}$ I

- We have

$$J^i \boldsymbol{v} = \frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \boldsymbol{\theta}^T} \boldsymbol{v} = \mathcal{R}\{\boldsymbol{z}^{L+1,i}\}.$$

- We consider the following forward operations by assuming that

$$\mathcal{R}\{Z^{m,i}\}$$

is available from the previous layer

R Operator for $J^i v$ II

- From (18), we have

$$
\begin{aligned}
&\mathcal{R}\{\phi(\text{pad}(Z^{m,i}))\}\\
=&\mathcal{R}\left\{\text{mat}\left(P_\phi^{m,i} P_{\text{pad}}^{m,i} \text{vec}\left(Z^{m,i}\right)\right)\right\}\\
=&\text{mat}\left(\mathcal{R}\{P_\phi^{m,i} P_{\text{pad}}^{m,i} \text{vec}\left(Z^{m,i}\right)\}\right)\\
=&\text{mat}\left(P_\phi^{m,i} P_{\text{pad}}^{m,i} \mathcal{R}\{\text{vec}\left(Z^{m,i}\right)\}\right)_{h^m h^m d^m \times a_{\text{conv}}^m b_{\text{conv}}^m}
\end{aligned}
$$

R Operator for $J^i \boldsymbol{v}$ III

- From (17) and (18), we have

$$
\begin{aligned}
&\mathcal{R}\{S^{m,i}\} \\
=&\mathcal{R}\{W^m \phi(\text{pad}(Z^{m,i})) + \boldsymbol{b}^m \mathbb{1}^T_{a^m_{\text{conv}} b^m_{\text{conv}}}\} \\
=&\mathcal{R}\{W^m \phi(\text{pad}(Z^{m,i}))\} + \mathcal{R}\{\boldsymbol{b}^m \mathbb{1}^T_{a^m_{\text{conv}} b^m_{\text{conv}}}\} \\
=&\mathcal{R}\{W^m\}\phi(\text{pad}(Z^{m,i})) + W^m \mathcal{R}\{\phi(\text{pad}(Z^{m,i}))\} + \\
&\mathcal{R}\{\boldsymbol{b}^m\}\mathbb{1}^T_{a^m_{\text{conv}} b^m_{\text{conv}}} \\
=&V^m_W \phi(\text{pad}(Z^{m,i})) + W^m \mathcal{R}\{\phi(\text{pad}(Z^{m,i}))\} + \\
&\boldsymbol{v}^m_b \mathbb{1}^T_{a^m_{\text{conv}} b^m_{\text{conv}}},
\end{aligned}
$$

R Operator for $J^i \boldsymbol{v}$ IV

where we have

$$\begin{aligned}\mathcal{R}\{W^m\} =& V_W^m,\\ \mathcal{R}\{\boldsymbol{b}^m\} =& \boldsymbol{v}_b^m.\end{aligned}$$

- Note that

$$\boldsymbol{v} = \begin{bmatrix} \boldsymbol{v}^1 \\ \vdots \\ \boldsymbol{v}^L \end{bmatrix},$$

and each $\boldsymbol{v}^m, m = 1, \dots, L$ has the same length as the number of variables (including bias) at the mth layer.

R Operator for $J^i \boldsymbol{v}$ V

- We further split $\boldsymbol{v}^m$ to V_W^m (a matrix form) and $\boldsymbol{v}_b^m$
- From (16), we have

$$\mathcal{R}\{\sigma(S^{m,i})\} = \sigma'(S^{m,i}) \odot \mathcal{R}\{S^{m,i}\}. \tag{19}$$

- From (18), we have

$$\begin{aligned}
&\mathcal{R}\{Z^{m+1,i}\} \\
=&\mathcal{R}\{\text{mat}(P_{\text{pool}}^{m,i}\text{vec}(\sigma(S^{m,i})))\} \\
=&\text{mat}(\mathcal{R}\{P_{\text{pool}}^{m,i}\text{vec}(\sigma(S^{m,i}))\}) \\
=&\text{mat}\left(P_{\text{pool}}^{m,i}\mathcal{R}\{\text{vec}\left(\sigma(S^{m,i})\right)\}\right)_{d^{m+1}\times a^{m+1}b^{m+1}}.
\end{aligned}$$

R Operator for $J^i \boldsymbol{v}$ VI

- We can continue this process until we get

$$J^i \boldsymbol{v} = \mathcal{R}\{\boldsymbol{z}^{L+1,i}\}.$$

- Clearly, we do not need to store

$$\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}}, \ldots, \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}}$$

  as before, so the memory issue is solved

- But how about

$$(J^i)^T(\cdot)?$$

  We will explain later that they are not needed

Outline

1. Backward setting
   - Jacobian evaluation
   - Gauss-Newton Matrix-vector products
2. Forward + backward settings
   - R operator
   - Gauss-Newton matrix-vector product
3. Complexity analysis

Gauss-Newton Matrix-vector Product I

- From the above discussion, we have known how to calculate

$$J^i \boldsymbol{v}$$

- Calculate

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

  is known to be easy

Gauss-Newton Matrix-vector Product II

- Now for
$$(J^i)^T(B^i J^i \boldsymbol{v}),$$
if we define
$$\boldsymbol{u} = B^i J^i \boldsymbol{v},$$
then
$$(J^i)^T \boldsymbol{u} = \left(\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \boldsymbol{\theta}^T}\right)^T \boldsymbol{u}.$$
- But earlier the gradient calculation is
$$(J^i)^T \nabla_{\boldsymbol{z}^{L+1,i}} \xi(\boldsymbol{z}^{L+1,i}; \boldsymbol{y}^i, Z^{1,i}) = \left(\frac{\partial \boldsymbol{z}^{L+1,i}}{\partial \boldsymbol{\theta}^T}\right)^T \frac{\partial \xi_i}{\partial \boldsymbol{z}^{L+1,i}}$$

Gauss-Newton Matrix-vector Product III

- Thus the same backward procedure can be used
- All we need is to replace

$$\frac{\partial \xi_i}{\partial \boldsymbol{z}^{L+1,i}}$$

with

$$\boldsymbol{u}$$

- Therefore, we do not need to explicitly derive J^i at all.

Gauss-Newton Matrix-vector Product IV

- Thus for $(J^i)^T \boldsymbol{u}$, there is no need to store

$$\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}}, \ldots, \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}}$$

Outline

Complexity Analysis I

- We have known from past slides that matrix-matrix products are the bottleneck (though in our cases some slow MATLAB functions are also bottlenecks in practice)
- For simplicity, in our analysis we just count the number of matrix-matrix products without worrying about their sizes

Complexity Analysis II

- For approaches solely by backward settings, if

$$\frac{\partial z_1^{L+1,i}}{\partial S^{m,i}}, \ldots, \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}}$$

are stored, then the complexity of a Newton iteration is proportional to

$$(n_{L+1} + 1) + \#\text{CG} \times 2,$$

where # CG is the number of CG steps in that iteration

Complexity Analysis III

- If not, then
$$\#\text{CG} \times ((n_{L+1} + 1) + 2)$$
Note that here we assume that $Z^{m,i}$ are not stored either, so at each CG step, a forward process is needed
- Therefore, "1" of "$n_{L+1} + 1$" comes from one product in the forward process. In the backward process we need n_{L+1} products
$$\text{vec}\left((W^m)^T \left[\frac{\partial z_1^{L+1,1}}{\partial S^{m,1}} \quad \dots \quad \frac{\partial z_{n_{L+1}}^{L+1,1}}{\partial S^{m,1}} \quad \dots \quad \frac{\partial z_{n_{L+1}}^{L+1,l}}{\partial S^{m,l}}\right]\right).$$

Complexity Analysis IV

- The situation is slightly different from the Gradient calculation, which needs "3" products (one in forward and two in backward).

  The reason is that now we do not need

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

- For "# CG $\times$ 2", the "2" is from (9) and (11)

Complexity Analysis V

- If using R operators, then

$$\#\mathrm{CG} \times (3 + 2)$$

products are needed, where "3" are from the forward process

$$W^m \phi(\mathrm{pad}(Z^{m,i})),$$

and

$$V_W^m \phi(\mathrm{pad}(Z^{m,i})), W^m \mathcal{R}\{\phi(\mathrm{pad}(Z^{m,i}))\},$$

and "2" are from the backward process

Complexity Analysis VI

- Clearly, under the same memory consumption, the one using R operators is much more efficient

Discussion I

- At this moment in the Python code we are not using the forward mode for $J\boldsymbol{v}$
- It was not available before
- However, since version 2.10 released in January 2020, this functionality is provided:

  https://www.tensorflow.org/api_docs/python/tf/autodiff/ForwardAccumulator
- It will be interesting to do the implementation and make a comparison