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

$$J^i = \begin{bmatrix} J^{1,i} & J^{2,i} & \ldots & J^{L,i} \end{bmatrix}, \quad m = 1, \ldots, L, \quad (1)$$

where

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

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

$$\frac{\partial \xi_i}{\partial \text{vec}(W^m)^T} = \text{vec} \left( \frac{\partial \xi_i}{\partial S_{m,i}} \phi(\text{pad}(Z_{m,i}))^T \right)^T$$
Now we have

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

\[
= \begin{bmatrix}
\text{vec}\left( \frac{\partial z_1^{L+1,i}}{\partial s_{m,i}} \phi(\text{pad}(Z^{m,i}))^T \right)^T \\
\vdots \\
\text{vec}\left( \frac{\partial z_{nL+1}^{L+1,i}}{\partial s_{m,i}} \phi(\text{pad}(Z^{m,i}))^T \right)^T
\end{bmatrix}
\]
If $b^m$ is considered, the result is

$$
\begin{bmatrix}
\frac{\partial z^{L+1,i}}{\partial \text{vec}(W^m)^T} & \frac{\partial z^{L+1,i}}{\partial (b^m)^T} \\
\text{vec} \left( \frac{\partial z^{L+1,i}}{\partial S^m,i} \left[ \phi(\text{pad}(Z^m,i))^T 1_{a_{\text{conv}}^m} b_{\text{conv}}^m \right] \right)^T \\
\text{vec} \left( \frac{\partial z^{nL+1,i}}{\partial S^m,i} \left[ \phi(\text{pad}(Z^m,i))^T 1_{a_{\text{conv}}^m} b_{\text{conv}}^m \right] \right)^T \\
\vdots \\
\end{bmatrix}
$$
We can see that it's more complicated than gradient.

Gradient is a vector but Jacobian is a matrix.
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
If with RELU activation function and max pooling, for gradient we had

$$\frac{\partial \xi_i}{\partial \text{vec}(S_{m,i})^T} = \left( \frac{\partial \xi_i}{\partial \text{vec}(Z_{m+1,i})^T} \odot \text{vec}(I[Z_{m+1,i}])^T \right) P_{pool}^{m,i}.$$
Assume that

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

are available.

$$\frac{\partial z^{L+1,i}_j}{\partial \text{vec}(S^{m,i})^T} = \left(\frac{\partial z^{L+1,i}_j}{\partial \text{vec}(Z^{m+1,i})^T} \odot \text{vec}(l[Z^{m+1,i}]^T)\right) P^{m,i}_{\text{pool}},$$

$$j = 1, \ldots, n_{L+1}.$$
These row vectors can be written together as a matrix

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

Note that

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

duplicates the \( \text{vec}(\mathbf{I}[\mathbf{Z}^{m+1,i}])^T \) vector \( n_{L+1} \) times.
Jacobian Evaluation: Backward Process V

- For gradient, we use

\[
\frac{\partial \xi_i}{\partial S_{m,i}}
\]

to have

\[
\frac{\partial \xi_i}{\partial \text{vec}(Z_{m,i}^T)} = \text{vec} \left( (W^m)^T \frac{\partial \xi_i}{\partial S_{m,i}} \right)^T P_{\phi}^m P_{\text{pad}}^m
\]

and pass it to the previous layer.
Now we need to generate

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

and pass it to the previous layer.

Now we have

\[
\frac{\partial z^{L+1,i}}{\partial \text{vec}(Z^{m,i})^T} = \begin{bmatrix}
\text{vec} \left( (W^m)^T \frac{\partial z^{L+1,i}_1}{\partial S^{m,i}} \right)^T P^m P^m_{\phi P_{\text{pad}}} \\
\vdots \\
\text{vec} \left( (W^m)^T \frac{\partial z^{L+1,i}_{n_L+1}}{\partial S^{m,i}} \right)^T P^m P^m_{\phi P_{\text{pad}}} 
\end{bmatrix}.
\]
We do not discuss details, but list all results below

\[
\frac{\partial z^{L+1,i}}{\partial \text{vec}(W^m)^T} = \begin{bmatrix}
\text{vec} \left( \frac{\partial z_{1}^{L+1,i}}{\partial s_{m,i}} (z_{m,i}^T) \right)^T \\
\vdots \\
\text{vec} \left( \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial s_{m,i}} (z_{m,i}^T) \right)^T 
\end{bmatrix}
\]
Jacobian Evaluation: Fully-connected
Layer II

\[
\frac{\partial z^{L+1,i}}{\partial (b^m)^T} = \frac{\partial z^{L+1,i}}{\partial (s^{m,i})^T},
\]
\[
\frac{\partial z^{L+1,i}}{\partial (s^{m,i})^T} = \frac{\partial z^{L+1,i}}{\partial (z^{m+1,i})^T} \odot (1_{n_{L+1}} I [z^{m+1,i}]^T)
\]
\[
\frac{\partial z^{L+1,i}}{\partial (z^{m,i})^T} = \frac{\partial z^{L+1,i}}{\partial (s^{m,i})^T} W^m
\]
Jacobian Evaluation: Fully-connected Layer III

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

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

then we have

$$\frac{\partial z^{L+1,i}}{\partial (s^{L,i})^T} = I_{n_{L+1}}.$$
Gradient versus Jacobian I

- Operations for gradient

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

\[
\frac{\partial \xi_i}{\partial W^m} = \frac{\partial \xi_i}{\partial S_{m,i}} \phi(\text{pad}(Z^{m,i}))^T.
\]

\[
\frac{\partial \xi_i}{\partial \text{vec}(Z^{m,i})^T} = \text{vec} \left( (W^m)^T \frac{\partial \xi_i}{\partial S_{m,i}} \right)^T P_{\phi}^{m} P_{\text{pad}}^{m}.
\]
Gradient versus Jacobian II

For Jacobian we have

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

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

\[ \frac{\partial z^{L+1,i}}{\partial \text{vec}(Z^{m,i})^T} = \begin{bmatrix} \text{vec}\left((W^m)^T \frac{\partial z_1^{L+1,i}}{\partial S^{m,i}}\right)^T P^m P^m \phi \ PAD \\ \vdots \\ \text{vec}\left((W^m)^T \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}}\right)^T P^m P^m \phi \ PAD \end{bmatrix}. \]
Implementation I

- For gradient we did

\[ \Delta \leftarrow \text{mat}(\text{vec}(\Delta)^T P_{\text{pool}}^m) \]

\[ \frac{\partial \xi_i}{\partial W^m} = \Delta \cdot \phi(\text{pad}(Z_m^i))^T \]

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

\[ \Delta \leftarrow \Delta \odot I[Z_{m,i}] \]

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

- We do not really store the Jacobian:

\[
\frac{\partial z^{L+1,i}}{\partial \text{vec}(W^m)^T} = \left[ \text{vec}\left( \frac{\partial z_1^{L+1,i}}{\partial S^m,i} \phi(\text{pad}(Z^{m,i}))^T \right)^T \right]
\]

- Recall Jacobian is used for matrix-vector products

\[
G^S v = \frac{1}{C} v + \frac{1}{|S|} \sum_{i \in S} \left( ((J^i)^T (B^i (J^i v))) \right)
\]
The form

\[
\frac{\partial z^{L+1,i}}{\partial \text{vec}(W^m)^T} = \begin{bmatrix}
\text{vec}(\frac{\partial z_1^{L+1,i}}{\partial S_{m,i}} \phi(\text{pad}(Z_{m,i}))^T)^T \\
\vdots \\
\text{vec}(\frac{\partial z_{nL+1}^{L+1,i}}{\partial S_{m,i}} \phi(\text{pad}(Z_{m,i}))^T)^T
\end{bmatrix}
\]

(4)

is like the product of two things
If we have
\[ \frac{\partial z_{L+1,i}}{\partial S_{m,j}}, \ldots, \frac{\partial z_{nL+1,i}}{\partial S_{m,j}}, \text{ and } \phi(\text{pad}(Z^{m,i})) \]
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
  \[ \frac{\partial z_{L+1,i}^1}{\partial S_{m,i}}, \ldots, \frac{\partial z_{L+1,i}^{n_{L+1}}}{\partial S_{m,i}} \]

- 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
In the beginning we have

\[ \Delta \in \mathbb{R}^{d_{m+1}a_{m+1}b_{m+1} \times n_{L+1} \times I} \] (5)

This corresponds to

\[ \frac{\partial \mathbf{Z}^{L+1,i}}{\partial \text{vec}(\mathbf{Z}^{m+1,i})^T} \odot (\mathbf{1}_{n_{L+1}} \text{vec}(I[\mathbf{Z}^{m+1,i}]^T), \forall i = 1, \ldots, I \]
We then calculate

\[ \Delta \leftarrow \text{mat} \begin{pmatrix} (P_{\text{pool}}^m)^T \text{vec}(\Delta_{:,1}) \\ \vdots \\ (P_{\text{pool}}^m)^T \text{vec}(\Delta_{:,l}) \end{pmatrix} \begin{pmatrix} d_{m+1} \\ a_{\text{conv}}^m \\ b_{\text{conv}}^m \\ n_{L+1} \end{pmatrix} \]

Recall that the pooling matrices are different across instances
The above operation corresponds to

\[
\frac{\partial z^{L+1,i}}{\partial \text{vec}(S^{m,i})^T} = \begin{pmatrix}
\frac{\partial z^{L+1,i}}{\partial \text{vec}(Z^{m+1,i})^T} \\
\end{pmatrix} \odot \left( 1_{n_{L+1}} \text{vec}(I[Z^{m+1,i}]^T) \right)
\]

\[P_{m,i}^{\text{pool}}.\]

Now we get

\[
\begin{bmatrix}
\frac{\partial z^{L+1,1}}{\partial S^{m,1}} & \cdots & \frac{\partial z^{L+1,1}}{\partial S^{m,1}} & \cdots & \frac{\partial z^{L+1,1}}{\partial S^{m,1}} \\
\vdots & \ddots & \vdots & \cdots & \vdots \\
\frac{\partial z^{n_{L+1},1}}{\partial S^{m,1}} & \cdots & \frac{\partial z^{n_{L+1},1}}{\partial S^{m,1}} & \cdots & \frac{\partial z^{n_{L+1},1}}{\partial S^{m,1}} \\
\end{bmatrix}
\in \mathbb{R}^{d^{m+1} \times a_{\text{conv}}^m b_{\text{conv}}^m n_{L+1}} \]
Implementation IX

- For gradient, the next step is to calculate

\[
\frac{\partial \xi_i}{\partial W^m} = \ldots
\]

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

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

- Therefore, the next operation is

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

- This is same as

\[ \text{vec} \left( (W^m)^T \begin{bmatrix} \frac{\partial z_{1}^{L+1,1}}{\partial S_{m,1}} & \ldots & \frac{\partial z_{n_{L+1}}^{L+1,1}}{\partial S_{m,1}} & \ldots & \frac{\partial z_{n_{L+1}}^{L+1,l}}{\partial S_{m,l}} \end{bmatrix} \right). \]

- Now \( V \) is a big vector like

\[
\begin{bmatrix}
\mathbf{v}_1^1 \\
\vdots \\
\mathbf{v}_{n_{L+1}}^1 \\
\vdots \\
\mathbf{v}_1^{n_{L+1}} \\
\vdots \\
\mathbf{v}_{n_{L+1}}^{n_{L+1}}
\end{bmatrix}
\]
Note that “\(\mathbf{v}\)” here is not the vector in matrix-vector products. We happen to use the same symbol.

From (2), we then calculate

\[
\begin{align*}
(v_1^1)^T \phi^m P^m_{\text{pad}} \\
\vdots \\
(v_{n_{L+1}}^1)^T \phi^m P^m_{\text{pad}} \\
\vdots \\
(v_{n_{L+1}}^l)^T \phi^m P^m_{\text{pad}}
\end{align*}
\]
For each resulting vector, we convert it to

$$\text{mat} \left( \mathbf{v}^T P^m_\phi P^m_{\text{pad}} \right) \in \mathbb{R}^{d^m \times a^m b^m}$$

This corresponds to

$$\frac{\partial z_{L+1,i}}{\partial Z_{m,i}}, \ldots, \frac{\partial z_{n_{L+1},i}}{\partial Z_{m,i}}, i = 1, \ldots, l$$
Finally,

$$\Delta \leftarrow \Delta \odot \left[ I[Z^{m,1}] \cdots I[Z^{m,1}] \cdots I[Z^{m,l}] \cdots I[Z^{m,l}] \right]_{nL+1 \times nL+1}$$

This is equivalent to

$$\frac{\partial z^{L+1,i}}{\partial \text{vec}(Z^{m,i})^T} \odot \left( \mathbf{1}_{nL+1} \text{vec}(I[Z^{m,i}])^T \right), \forall i = 1, \ldots, l$$
Note that in the beginning of the calculation, we assume that in (5)

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

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

- For doing several CG steps, we should store

\[ \frac{\partial z^{L+1,i}}{\partial S^{m,i}}, \ldots, \frac{\partial z^{L+1,i}}{\partial S^{n_{L+1},i}}, i = 1, \ldots, l \]

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

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

\[ Gv \]

though the situation of using \( G^S \) (i.e., a subset of data) is the same

The Gauss-Newton matrix is

\[
G = \frac{1}{C}I + \frac{1}{l} \sum_{i=1}^{l} \begin{bmatrix} (J^{1,i})^T \\ \vdots \\ (J^{L,i})^T \end{bmatrix} B^i \begin{bmatrix} J^{1,i} \\ \vdots \\ J^{L,i} \end{bmatrix}
\]
Gauss-Newton Matrix-Vector Products II

The Gauss-Newton matrix vector product is

\[ G \mathbf{v} = \frac{1}{C} \mathbf{v} + \frac{1}{l} \sum_{i=1}^{l} \begin{bmatrix} (J^1,i)^T \\ \vdots \\ (J^L,i)^T \end{bmatrix} B^i \begin{bmatrix} J^1,i \\ \vdots \\ J^L,i \end{bmatrix} \begin{bmatrix} \mathbf{v}^1 \\ \vdots \\ \mathbf{v}^L \end{bmatrix} \]

\( (8) \)
where

\[ \mathbf{v} = \begin{bmatrix} \mathbf{v}^1 \\ \vdots \\ \mathbf{v}^L \end{bmatrix} \]

- Each \( \mathbf{v}^m, m = 1, \ldots, L \) has the same length as the number of variables (including bias) at the \( m \)th layer.
For the convolutional layers,

\[ J^{m,i} \mathbf{v}^m = \begin{bmatrix} \text{vec} \left( \frac{\partial z^{L+1,i}}{\partial S^{m,i}} \left[ \phi(\text{pad}(Z^{m,i}))^T \mathbf{1}_{a_{\text{conv}}^{m}} b_{\text{conv}}^{m} \right] \right)^T \mathbf{v}^m \\ \vdots \\ \text{vec} \left( \frac{\partial z^{n_{L+1},i}}{\partial S^{m,i}} \left[ \phi(\text{pad}(Z^{m,i}))^T \mathbf{1}_{a_{\text{conv}}^{m}} b_{\text{conv}}^{m} \right] \right)^T \mathbf{v}^m \end{bmatrix} \in \mathbb{R}^{n_{L+1} \times 1} \]

By this formulation, we need
Jacobian-vector Product II

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

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

$$
\vdots
$$

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

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

\[
\begin{align*}
\text{vec} \left( \begin{pmatrix} 
\frac{\partial z^{L+1,i}}{\partial S_{m,i}} \\
\vdots \\
\frac{\partial z^{L+1,1}}{\partial S_{m,1}} 
\end{pmatrix} \begin{bmatrix} 
\phi(\text{pad}(Z^{m,i}))^T \\
\mathbf{1}_{a_{conv}} \\
\mathbf{1}_{b_{conv}} 
\end{bmatrix} \right) \\
= \frac{\partial z^{L+1,i}}{\partial \text{vec}(S_{m,i})} \times \\
\text{vec} \left( \text{mat}(\mathbf{v}^m)_{d^{m+1}} \times (h^m h^m d^{m+1}) \begin{bmatrix} 
\phi(\text{pad}(Z^{m,i}))^T \\
\mathbf{1}_{a_{conv}} \\
\mathbf{1}_{b_{conv}} 
\end{bmatrix} \right).
\end{align*}
\]
If all elements are considered together

\[ J^{m,i} \mathbf{v}^m = \frac{\partial z^{L+1,i}}{\partial \text{vec}(S^{m,i})^T} \times \text{vec} \left( \text{mat}(\mathbf{v}^m)_{d^{m+1}} \times (h^m h^m d^{m+1}) \right) \left[ \phi(\text{pad}(Z^{m,i})) \right]^T \left[ \begin{array}{c} 1 \quad a^m_{\text{conv}} b^m_{\text{conv}} \end{array} \right]. \] (9)

This involves
- One matrix-matrix product
- One matrix-vector product
After deriving (9), from (8), we sum results of all layers

\[
\sum_{m=1}^{L} J^{m,i} v^m
\]

Next we calculate

\[
q^i = B^i \left( \sum_{m=1}^{L} J^{m,i} v^m \right).
\]  

This is usually easy
We mentioned earlier that if the squared loss is used

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

is a diagonal matrix
Finally, we calculate

$$
(J^{m,i})^T q^i \\
= \left[ \text{vec} \left( \frac{\partial Z^{L+1,i}}{\partial S_{m,i}} \left[ \phi(\text{pad}(Z^{m,i}))^T 1_{a_{\text{conv}}^{m} b_{\text{conv}}^{m}} \right] \right) \right] \cdots \\
\text{vec} \left( \frac{\partial Z_{nL+1,i}}{\partial S_{m,i}} \left[ \phi(\text{pad}(Z^{m,i}))^T 1_{a_{\text{conv}}^{m} b_{\text{conv}}^{m}} \right] \right) q^i
$$
Transposed Jacobian-vector Products IV

\[
= \sum_{j=1}^{n_L+1} q_j^i \text{vec} \left( \frac{\partial z_{L+1,i}^j}{\partial S_{m,i}} \left[ \phi(\text{pad}(Z_{m,i}^m))^T 1_{a_{conv} b_{conv}} \right] \right)
\]

\[
= \text{vec} \left( \sum_{j=1}^{n_L+1} q_j^i \left( \frac{\partial z_{L+1,i}^j}{\partial S_{m,i}} \left[ \phi(\text{pad}(Z_{m,i}^m))^T 1_{a_{conv} b_{conv}} \right] \right) \right)
\]

\[
= \text{vec} \left( \sum_{j=1}^{n_L+1} q_j^i \frac{\partial z_{L+1,i}^j}{\partial S_{m,i}} \left[ \phi(\text{pad}(Z_{m,i}^m))^T 1_{a_{conv} b_{conv}} \right] \right)
\]
Transposed Jacobian-vector Products $V$

\[
= \text{vec} \left( \text{mat} \left( \left( \frac{\partial z^{L+1,i}}{\partial \text{vec}(S^{m,i})^T} \right)^T q^i \right) \right) \times d^{m+1} \times a^m_{\text{conv}} b^m_{\text{conv}} \\
\left[ \phi(\text{pad}(Z^{m,i}))^T \mathbb{1}_{a^m_{\text{conv}} b^m_{\text{conv}}} \right].
\] (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

\[
J^{m,i}v^m = \frac{\partial z^{L+1,i}}{\partial (s^{m,i})^T} \text{mat}(v^m)_{n_{m+1} \times (n_m+1)} \left[ z^{m,i} \right].
\]

\[
(J^{m,i})^T q^i = \text{vec} \left( \left( \frac{\partial z^{L+1,i}}{\partial (s^{m,i})^T} \right)^T q^i \left[ (z^{m,i})^T 1_1 \right] \right).
\]
As before, we must handle all instances together.

We discuss only

\[
\begin{bmatrix}
\sum_{m=1}^{L} J^{m,1} \nu^m \\
\vdots \\
\sum_{m=1}^{L} J^{m,l} \nu^m 
\end{bmatrix} \in \mathbb{R}^{nL+1 \times 1}
\]

Following earlier derivation.
Implementation II

\[
\begin{bmatrix}
J_{m,1} v^m \\
\vdots \\
J_{m,l} v^m
\end{bmatrix} = \begin{bmatrix}
\frac{\partial z^{L+1,1}}{\partial \text{vec}(S_{m,1})^T} \text{vec} \left( \text{mat}(v^m) \left[ \phi(\text{pad}(Z_{m,1}^m)) \right] \right) \\
\vdots \\
\frac{\partial z^{L+1,l}}{\partial \text{vec}(S_{m,l})^T} \text{vec} \left( \text{mat}(v^m) \left[ \phi(\text{pad}(Z_{m,l}^m)) \right] \right)
\end{bmatrix} = \begin{bmatrix}
\frac{\partial z^{L+1,1}}{\partial \text{vec}(S_{m,1})^T} p_{m,1} \\
\vdots \\
\frac{\partial z^{L+1,l}}{\partial \text{vec}(S_{m,l})^T} p_{m,l}
\end{bmatrix},
\]
Implementation III

- where

\[ \text{mat}(\mathbf{v}^m) \in \mathbb{R}^{d^m+1 \times (h^m h^m d^m+1)} \]

and

\[ p^{m,i} = \text{vec} \left( \text{mat}(\mathbf{v}^m) \left[ \begin{array}{c} \phi(\text{pad}(Z^{m,i})) \\ 1^T a^m_{\text{conv}} b^m_{\text{conv}} \end{array} \right] \right) . \quad (12) \]
Implementation IV

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

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

- To get

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

we need \( l \) matrix-vector products

- There is no good way to transform it to matrix-matrix operations
To avoid a for loop over all data, here we implement the matrix-vector product

$$J^{m,i} \mathbf{v}^m = \frac{\partial z^{L+1,i}}{\partial \text{vec}(S^{m,i})} p^{m,i}$$

by a summation of all rows of the following matrix

$$\begin{bmatrix}
\frac{\partial z_1^{L+1,i}}{\partial \text{vec}(S^{m,i})} & \cdots & \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial \text{vec}(S^{m,i})} \\
\mathbf{p}^{m,i} & \cdots & \mathbf{p}^{m,i}
\end{bmatrix} d^{m+1} a^{m}_{\text{conv}} b^{m}_{\text{conv}} \times n_{L+1}$$
For example, summing up all elements of the first column is the inner product between the first row of

\[ \frac{\partial z^{L+1,i}}{\partial \text{vec}(S^{m,i})^T} \]

and \( p^{m,i} \).

Then all the \( l \) matrix-vector products

\[ J^{m,i} v^m = \frac{\partial z^{L+1,i}}{\partial \text{vec}(S^{m,i})^T} p^{m,i}, \quad i = 1, \ldots, l. \]
Implementation VIII

can be done in one line by

$$
\begin{bmatrix}
\frac{\partial z^{L+1,i}}{\partial \text{vec}(S^{m,i})} & \cdots & \frac{\partial z^{nL+1,i}}{\partial \text{vec}(S^{m,i})} \\
\cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots \\
\cdots & \cdots & \cdots \\
\end{bmatrix}
$$

- The code (convolutional layers) is like
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 $p_{m,i}^m$ we do not duplicate it $n_{L+1}$ times. Instead, for .*, MATLAB does the expansion automatically
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
We mentioned before that two types of autodiff are forward and reverse modes.

For the Jacobian evaluation, at layer $m$,

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

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,

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

where the memory cost is

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

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

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

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.
Recall earlier we said that by the forward mode, the Jacobian-vector product can be done in just one pass.
Consider $g(\theta) \in \mathbb{R}^{k\times 1}$. Following Pearlmutter (1994), we define

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

(14)

Note that

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

is the Jacobian of $g(\theta)$.
This definition can be extended to a matrix $M(\theta) \in \mathbb{R}^{k \times t}$ by

$$
\mathcal{R}_v \{ M(\theta) \} \equiv \mat(\mathcal{R}_v \{ \text{vec}(M(\theta)) \})_{k \times t}
$$

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

Clearly,

$$
\mathcal{R}_v \{ M(\theta) \} = (\mathcal{R}_v \{ M(\theta)^T \})^T. \quad (15)
$$
If \( h(\cdot) \) is a scalar function, we let

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

and

\[
h'(M(\theta)) = \begin{bmatrix} h'(M_{11}) & \cdots & h'(M_{1t}) \\ \vdots & \ddots & \vdots \\ h'(M_{k1}) & \cdots & h'(M_{kt}) \end{bmatrix}.
\]
R Operator IV

Because

\[ \nabla (h(M_{ij}(\theta)))^T \mathbf{v} = h'(M_{ij}) \nabla (M_{ij})^T \mathbf{v}, \]

we have

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

(16)
where \( \odot \) stands for the Hadamard product (i.e., component-wise product).

- If \( M(\theta) \) and \( T(\theta) \) have the same size,

\[
\mathcal{R}_v\{M(\theta) + T(\theta)\} = \mathcal{R}_v\{M(\theta)\} + \mathcal{R}_v\{T(\theta)\}.
\]  

(17)

- Lastly, we have

\[
\mathcal{R}_v\{U(\theta)M(\theta)\} = \mathcal{R}_v\{U(\theta)\}M(\theta) + U(\theta)\mathcal{R}_v\{M(\theta)\}
\]  

(18)
Proof: Note that

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

With

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

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

\[
\nabla \left( U_{ip}M_{pj} \right)^T \mathbf{v} = \left( (\nabla U_{ip})^T \mathbf{v} \right) M_{pj} + U_{ip} \left( (\nabla M_{pj})^T \mathbf{v} \right).
\]
The summation

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

leads to the \((i,j)\) component of

$$\mathcal{R}_v \{ U(\theta) \} M(\theta)$$

Thus we have (18)

- For simplicity, subsequently we use \(\mathcal{R}\{g(\theta)\}\) to denote \(\mathcal{R}_v \{ g(\theta) \}\)
R Operator for $J^i \nu$ I

- We have

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

- We consider the following forward operations by assuming that

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

is available from the previous layer.
From (18), we have

\[
\mathcal{R}\{\phi(\text{pad}(Z^{m,i}))\} \\
\quad = \mathcal{R}\left\{ \text{mat}\left( P^{m,i}_\phi P^{m,i}_{\text{pad}} \text{vec}(Z^{m,i}) \right) \right\} \\
\quad = \text{mat}\left( \mathcal{R}\{P^{m,i}_\phi P^{m,i}_{\text{pad}} \text{vec}(Z^{m,i})\} \right) \\
\quad = \text{mat}\left( P^{m,i}_\phi P^{m,i}_{\text{pad}} \mathcal{R}\{\text{vec}(Z^{m,i})\} \right) \\
\quad = h^m h^m d^m \times a^m_{\text{conv}} b^m_{\text{conv}}
\]
From (17) and (18), we have

\[
\mathcal{R}\{S^{m,i}\} \\
= \mathcal{R}\{W^m \phi(\text{pad}(Z^{m,i})) + b^m 1^T_{a_{\text{conv}}b_{\text{conv}}}\} \\
= \mathcal{R}\{W^m \phi(\text{pad}(Z^{m,i}))\} + \mathcal{R}\{b^m 1^T_{a_{\text{conv}}b_{\text{conv}}}\} \\
= \mathcal{R}\{W^m\} \phi(\text{pad}(Z^{m,i})) + W^m \mathcal{R}\{\phi(\text{pad}(Z^{m,i}))\} + \\
\mathcal{R}\{b^m\} 1^T_{a_{\text{conv}}b_{\text{conv}}} \\
= V_W^m \phi(\text{pad}(Z^{m,i})) + W^m \mathcal{R}\{\phi(\text{pad}(Z^{m,i}))\} + \\
\nu^m b^m 1^T_{a_{\text{conv}}b_{\text{conv}}},
\]
Forward + backward settings

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

where we have

$$\mathcal{R}\{W^m\} = V^m_W,$$
$$\mathcal{R}\{b^m\} = v^m_b.$$

- Note that

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

and each $v^m, m = 1, \ldots, L$ has the same length as the number of variables (including bias) at the $m$th layer.
R Operator for $J^i \nu \ V$

- We further split $\nu^m$ to $V^m_W$ (a matrix form) and $\nu^m_b$
- From (16), we have

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

- From (18), we have

$$R\{Z^{m+1,i}\}$$
$$= R\{\text{mat}(P^{m,i}_{\text{pool}} \ \text{vec}(\sigma(S^{m,i}))))\}$$
$$= \text{mat}(R\{P^{m,i}_{\text{pool}} \ \text{vec}(\sigma(S^{m,i}))))$$
$$= \text{mat} \left( P^{m,i}_{\text{pool}} \ R\{\text{vec}(\sigma(S^{m,i}))\} \right)_{d^{m+1} \times a^{m+1} b^{m+1}}.$$
We can continue this process until we get

\[ J^i \mathbf{v} = \mathcal{R}\{z^{L+1,i}\} \]

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
From the above discussion, we have known how to calculate

\[ J^i v \]

Calculate

\[ B^i (J^i v) \]

is known to be easy
Gauss-Newton Matrix-vector Product II

Now for

$$(J^i)^T (B^i J^i \mathbf{v}),$$

if we define

$$\mathbf{u} = B^i J^i \mathbf{v},$$

then

$$(J^i)^T \mathbf{u} = \left( \frac{\partial z^{L+1,i}}{\partial \theta^T} \right)^T \mathbf{u}.$$

But earlier the gradient calculation is

$$(J^i)^T \nabla_{z^{L+1,i}} \xi(z^{L+1,i}; y^i, Z^{1,i}) = \left( \frac{\partial z^{L+1,i}}{\partial \theta^T} \right)^T \frac{\partial \xi_i}{\partial z^{L+1,i}}.$$
Thus the same backward procedure can be used. All we need is to replace
\[ \frac{\partial \xi_i}{\partial z^{L+1,i}} \]
with
\[ u \]
Therefore, we do not need to explicitly derive \( J^i \) at all.
Thus for \((J^i)^T u\), 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

1. Backward setting
   - Jacobian evaluation
   - Gauss-Newton Matrix-vector products

2. Forward + backward settings
   - R operator
   - Gauss-Newton matrix-vector product

3. Complexity analysis
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

\[
\frac{\partial z_{L+1,i}}{\partial S_{m,i}}, \ldots, \frac{\partial z_{n_{L+1}}}{\partial S_{m,i}}
\]

are stored, then

\[n_{L+1} \times 3 + \#CG \times 2\]

If not, then

\[\#CG \times (n_{L+1} \times 3 + 2)\]
Note that “3” comes from one product in the forward process and two in the backward process (the same as the situation in Gradient calculation).

For example,

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

And “2” are from (9) and (11)
Complexity Analysis IV

- If using R operators, then

\[ \#CG \times (3 + 2), \]

where “3” are from the forward process

\[ W^m \phi(pad(Z^{m,i})), \]

and

\[ V^m_W \phi(pad(Z^{m,i})), W^m R \{ \phi(pad(Z^{m,i})) \}, \]

and “2” are from the backward process

- 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 $J_\nu$
- 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