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

$$J^i = [J^{1,i}, J^{2,i}, \ldots, J^{L,i}], \quad m = 1, \ldots, L,$$

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

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

$$
\begin{bmatrix}
\frac{\partial z^{L+1,i}}{\partial \text{vec}(W^m)^T} & \frac{\partial z^{L+1,i}}{\partial (b^m)^T} \\
\end{bmatrix}
\begin{bmatrix}
\text{vec} \left( \frac{\partial z^{L+1,i}_{1}}{\partial S^{m,i}} \left[ \phi(\text{pad}(Z^{m,i}))^T 1_{a_{\text{conv}}^m} b_{\text{conv}}^m \right] \right) \\
\vdots \\
\text{vec} \left( \frac{\partial z^{L+1,i}_{n_{L+1}}}{\partial S^{m,i}} \left[ \phi(\text{pad}(Z^{m,i}))^T 1_{a_{\text{conv}}^m} b_{\text{conv}}^m \right] \right)
\end{bmatrix}^T.
$$
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
  \[ \frac{\partial Z^{L+1,i}_{1}}{\partial S^{m,i}}, \ldots, \frac{\partial Z^{L+1,i}_{n_{L+1}}}{\partial S^{m,i}} \]

- The process is similar
If with RELU activation function and max pooling, for gradient we had

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

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

- These row vectors can be written together as a matrix

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

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

- 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.
For gradient, we use
\[
\frac{\partial \xi_i}{\partial S_{m,i}}
\]
to have
\[
\frac{\partial \xi_i}{\partial \text{vec}(Z_{m,i}^T)} = \text{vec} \left( (W^m)^T \frac{\partial \xi_i}{\partial S_{m,i}} \right)^T P^m \phi P^m_{\text{pad}}
\]
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
\[
\frac{\partial z^{L+1,i}}{\partial \text{vec}(Z^{m,i})^T} = 
\begin{bmatrix}
\text{vec} \left( (W^m)^T \frac{\partial z_1^{L+1,i}}{\partial S^{m,i}} \right)^T P_m P^m \phi P^m_{\text{pad}} \\
\vdots \\
\text{vec} \left( (W^m)^T \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S^{m,i}} \right)^T P_m P^m \phi P^m_{\text{pad}}
\end{bmatrix}.
\]
Jacobian Evaluation: Fully-connected Layer I

We do not discuss details, but list all results below

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

\[
\begin{align*}
\frac{\partial z^{L+1,i}}{\partial (b^m)^T} &= \frac{\partial z^{L+1,i}}{\partial (s^{m,i})^T}, \\
\frac{\partial z^{L+1,i}}{\partial (s^{m,i})^T} &= \frac{\partial z^{L+1,i}}{\partial (z^{m+1,i})^T} \odot (1_{n_{L+1}} I [z^{m+1,i}]^T) \\
\frac{\partial z^{L+1,i}}{\partial (z^{m,i})^T} &= \frac{\partial z^{L+1,i}}{\partial (s^{m,i})^T} W^m
\end{align*}
\]
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

$$\frac{\partial z_{L+1,i}}{\partial (s_{L,i})^T} = I_{n_{L+1}}.$$

Chih-Jen Lin (National Taiwan Univ.)
Gradient versus Jacobian I

Operations for gradient

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

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

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

- For Jacobian we have

\[
\frac{\partial z^{L+1,i}}{\partial \text{vec}(S_{m,i})^T} = \left( \frac{\partial z^{L+1,i}}{\partial \text{vec}(Z_{m+1,i})^T} \otimes \left( \mathbb{1}_{n_{L+1}} \text{vec}(I[Z_{m+1,i}])^T \right) \right) P_{m,i}^{\text{pool}}.
\]

\[
\frac{\partial z^{L+1,i}}{\partial \text{vec}(W^m)^T} = \begin{bmatrix}
\text{vec}\left( \frac{\partial z_{1}^{L+1,i}}{\partial S_{m,i}} \phi(\text{pad}(Z_{m,i}))^T \right)^T \\
\vdots \\
\text{vec}\left( \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial S_{m,i}} \phi(\text{pad}(Z_{m,i}))^T \right)^T
\end{bmatrix}
\]
Gradient versus Jacobian III

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

- For gradient we did

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

\[
\frac{\partial \xi_i}{\partial W_m} = \Delta \cdot \phi(\text{pad}(Z_m^i))^	op
\]

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

\[
\Delta \leftarrow \Delta \odot I[Z_m^i]
\]

- Now for Jacobian we have similar settings but there are some differences
We do not really store the Jacobian:

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

Recall Jacobian is used for matrix-vector products

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

The form

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

is like the product of two things
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
  \[ \frac{\partial Z^{L+1, i}_1}{\partial S^{m, i}}, \ldots, \frac{\partial Z^{L+1, i}_{n_{L+1}}}{\partial S^{m, i}} \]

- 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

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

(5)

This corresponds to

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

- We then calculate

\[
\Delta \leftarrow \text{mat} \left( \begin{bmatrix}
(P_{\text{pool}}^{m,1})^T \text{vec}(\Delta_{:,1,1}) \\
\vdots \\
(P_{\text{pool}}^{m,l})^T \text{vec}(\Delta_{:,l,1})
\end{bmatrix} \right)
\]

\[d^{m+1} \times a_{\text{conv}}^m b_{\text{conv}}^m n_{L+1} l\]

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

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

Now we get

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

Chih-Jen Lin (National Taiwan Univ.)
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

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

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

Now $V$ is a big vector like

$$\begin{bmatrix} v_1^1 & \cdots & v_{n_{L+1}}^1 & \cdots & v_{n_{L+1}}^{L+1} \end{bmatrix}$$
Note that “v” here is not the vector in matrix-vector products. We happen to use the same symbol.

- From (2), we then calculate

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

\[ \text{mat} \left( v^T P^m \phi P^m_{\text{pad}} \right)_{d^m \times a^m b^m} \]

This corresponds to

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

\[ \Delta \leftarrow \Delta \odot \]

\[
\begin{bmatrix}
I[Z_m,1] \cdots I[Z_m,1] \\
\vdots \\
I[Z_m,l] \cdots I[Z_m,l]
\end{bmatrix}_{nL+1} \\
\begin{bmatrix}
I[Z_m,1] \cdots I[Z_m,1] \\
\vdots \\
I[Z_m,l] \cdots I[Z_m,l]
\end{bmatrix}_{nL+1}
\]

(6)

This is equivalent to

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

\[
G = \frac{1}{C} I + \frac{1}{l} \sum_{i=1}^{l} \begin{bmatrix} (J^{1,i})^T \\ \vdots \\ (J^{L,i})^T \end{bmatrix} B_i \begin{bmatrix} J^{1,i} & \ldots & J^{L,i} \end{bmatrix}
\]
The Gauss-Newton matrix vector product is

$$Gv = \frac{1}{C}v + \frac{1}{l} \sum_{i=1}^{l} \begin{bmatrix} (J_{1,i})^T \\ \vdots \\ (J_{L,i})^T \end{bmatrix} B^i \begin{bmatrix} J_{1,i} \\ \vdots \\ J_{L,i} \end{bmatrix} v^1 \ldots \begin{bmatrix} v^1 \\ \vdots \\ v^L \end{bmatrix} = \frac{1}{C}v + \frac{1}{l} \sum_{i=1}^{l} \begin{bmatrix} (J_{1,i})^T \\ \vdots \\ (J_{L,i})^T \end{bmatrix} \left( B^i \sum_{m=1}^{L} J_{m,i}^T v^m \right),$$

(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.
Jacobian-vector Product I

- For the convolutional layers,

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

- By this formulation, we need
Jacobian-vector Product II

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

$$
\text{vec} \left( \frac{\partial z^{L+1,i}}{\partial s^{m,i}} \begin{bmatrix} \phi(\text{pad}(Z^{m,i}))^T & 1_{a_m^{\text{conv}}b_m^{\text{conv}}} \end{bmatrix} \right)^T
$$

: 

$$
\text{vec} \left( \frac{\partial z^{n_{L+1,i}}}{\partial s^{m,i}} \begin{bmatrix} \phi(\text{pad}(Z^{m,i}))^T & 1_{a_m^{\text{conv}}b_m^{\text{conv}}} \end{bmatrix} \right)^T
$$

- the product between the above matrix and a vector $\nu^m$
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

\[
\text{vec} \left( \frac{\partial z_{L+1,i}}{\partial S_{m,i}} \right) A \left[ \phi(\text{pad}(Z_{m,i}))^T 1_{a_{\text{conv}}} b_{\text{conv}} \right] B \text{vec}(C) = \frac{\partial z_{L+1,i}}{\partial \text{vec}(S_{m,i})^T} \times \text{vec} \left( \text{mat}(v^m)_{d^m+1 \times (h^m h^m d^m+1)} \right) \left[ \phi(\text{pad}(Z_{m,i})) \right]^T 1_{a_{\text{conv}}} b_{\text{conv}} \right).
\]
Jacobian-vector Product V

- If all elements are considered together

\[
J_{m,i} v^m = \frac{\partial z^{L+1,i}}{\partial \text{vec}(S_{m,i})^T} \times \text{vec} \left( \text{mat}(v^m)_{d^{m+1}} \times (h^m h^m d^m + 1) \right) \left[ \phi(\text{pad}(Z^{m,i})) \right]^T \\
\]

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

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

Next we calculate

$$q^i = B^i (\sum_{m=1}^{L} J^{m,i} \mathbf{v}_m).$$

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

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

is a diagonal matrix
Finally, we calculate

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

\[ \begin{align*}
= & \sum_{j=1}^{n_{L+1}} q_j^i \text{vec} \left( \frac{\partial z_{L+1, i}^j}{\partial s_{m, i}} \left[ \phi(\text{pad}(Z^m, i))^T \mathbf{1}_{a_{\text{conv}}}^m b_{\text{conv}}^m \right] \right) \\
= & \text{vec} \left( \sum_{j=1}^{n_{L+1}} q_j^i \left( \frac{\partial z_{L+1, i}^j}{\partial s_{m, i}} \left[ \phi(\text{pad}(Z^m, i))^T \mathbf{1}_{a_{\text{conv}}}^m b_{\text{conv}}^m \right] \right) \right) \\
= & \text{vec} \left( \left( \sum_{j=1}^{n_{L+1}} q_j^i \frac{\partial z_{L+1, i}^j}{\partial s_{m, i}} \right) \left[ \phi(\text{pad}(Z^m, i))^T \mathbf{1}_{a_{\text{conv}}}^m b_{\text{conv}}^m \right] \right)
\end{align*} \]
Transposed Jacobian-vector Products V

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

\[ \left[ \phi(\text{pad}(Z^{m,i}))^T \mathbb{1} a^m_{\text{conv}} b^m_{\text{conv}} \right]. \] (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

\[ J_{m,i}^m \mathbf{v}^m = \frac{\partial \mathbf{z}^{L+1,i}}{\partial (\mathbf{s}^{m,i})^T} \text{mat}(\mathbf{v}^m)_{n_{m+1} \times (n_{m+1})} \begin{bmatrix} \mathbf{z}^{m,i} \\ \mathbf{1}_1 \end{bmatrix}. \]

\[ (J_{m,i}^m)^T \mathbf{q}^i = \text{vec} \left( \left( \frac{\partial \mathbf{z}^{L+1,i}}{\partial (\mathbf{s}^{m,i})^T} \right)^T \mathbf{q}^i \begin{bmatrix} (\mathbf{z}^{m,i})^T & \mathbf{1}_1 \end{bmatrix} \right). \]
Implementation I

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

$$\begin{bmatrix}
\sum_{m=1}^{L} J_{m,1} v^m \\
\vdots \\
\sum_{m=1}^{L} J_{m,l} v^m
\end{bmatrix} \in \mathbb{R}^{nL+1 \times 1}$$

- Following earlier derivation
Implementation II

\[
\begin{bmatrix}
J^{m,1} & v^m \\
& \\
& \\
J^{m,l} & v^m \\
\end{bmatrix} = \\
\begin{bmatrix}
\frac{\partial z^{L+1,1}}{\partial \text{vec}(S^{m,1})^T} \text{vec} \left( \text{mat}(v^m) \left[ \phi(\text{pad}(Z^{m,1})) \right] \right) \\
& \\
& \\
& \\
\frac{\partial z^{L+1,l}}{\partial \text{vec}(S^{m,l})^T} \text{vec} \left( \text{mat}(v^m) \left[ \phi(\text{pad}(Z^{m,l})) \right] \right) \\
& \\
& \\
& \\
\frac{\partial z^{L+1,1}}{\partial \text{vec}(S^{m,1})^T} p^{m,1} \\
& \\
& \\
& \\
\frac{\partial z^{L+1,l}}{\partial \text{vec}(S^{m,l})^T} p^{m,l} \\
\end{bmatrix}
\]
where

\[ \text{mat} (v^m) \in \mathbb{R}^{d_m+1 \times (h_m h_m d_m+1)} \]

and

\[ p^{m,i} = \text{vec} \left( \text{mat}(v^m) \left[ \begin{array}{c} \phi(\text{pad}(Z^m,i)) \\ 1_T \end{array} \right] \right) \]  \quad (12)
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

$$\text{mat}(v^m) \begin{bmatrix} \phi(\text{pad}(Z^{m,1})) & \cdots & \phi(\text{pad}(Z^{m,l})) \\ 1^T a_{\text{conv}}^m b_{\text{conv}}^m & \cdots & 1^T a_{\text{conv}}^m b_{\text{conv}}^m \end{bmatrix}$$

$$\in \mathbb{R}^{d_{m+1} \times a_{\text{conv}}^m b_{\text{conv}}^m l}$$
Implementation V

To get

\[
\begin{bmatrix}
\frac{\partial z_{L+1,1}}{\partial \text{vec}(S_{m,1})^T} p_{m,1}^m \\
\vdots \\
\frac{\partial z_{L+1,l}}{\partial \text{vec}(S_{m,l})^T} p_{m,l}^m 
\end{bmatrix},
\]

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

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

by a summation of all rows of the following matrix

\[
\begin{bmatrix}
\frac{\partial z_1^{L+1,i}}{\partial \text{vec}(S^{m,i})} & \cdots & \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial \text{vec}(S^{m,i})}
\end{bmatrix}
\begin{bmatrix}
p^{m,i} \\
\vdots \\
p^{m,i}
\end{bmatrix}
\]

\[ d^{m+1} a_{\text{conv}}^m b_{\text{conv}}^m \times n_{L+1} \times n_{L+1} \]
Implementation VII

- For example, summing up all elements of the first column is the inner product between the first row of $\frac{\partial z^{L+1,i}}{\partial \text{vec}(S^{m,i})^T}$ 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

\[
\begin{bmatrix}
\frac{\partial z_{1}^{L+1,i}}{\partial \text{vec}(S^{m,i})} & \cdots & \frac{\partial z_{n_{L+1}}^{L+1,i}}{\partial \text{vec}(S^{m,i})}
\end{bmatrix}
\circ
\begin{bmatrix}
\cdots p^{m,i} \cdots p^{m,i} \cdots 
\end{bmatrix}
\]

- 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
  \[ \text{sum(:,1);} \]
  sums up all rows

- For \( p^{m,i} \) 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
Reverse versus Forward Autodiff I

- We mentioned before that two types of autodiff are forward and reverse modes.
- For the Jacobian evaluation, at layer $m$,

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

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,

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

where the memory cost is

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

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
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

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

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

Clearly,

$$\mathcal{R}_v \{ M(\theta) \} = \left( \mathcal{R}_v \{ M(\theta)^T \} \right)^T . \quad (15)$$
R Operator III

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

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

$$R_v\{M(\theta) + T(\theta)\} = R_v\{M(\theta)\} + R_v\{T(\theta)\}. \tag{17}$$

- Lastly, we have

$$R_v\{U(\theta)M(\theta)\} = R_v\{U(\theta)\}M(\theta) + U(\theta)R_v\{M(\theta)\} \tag{18}$$
Proof: Note that

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

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,

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

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

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

\[ R_v \{ U(\theta) \} M(\theta) \]

Thus we have (18)

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

From (18), we have

\[
R\{\phi(pad(Z^{m,i}))\} \\
= R \left\{ \text{mat} \left( P^m_i P^m_i \text{pad} \text{vec} \left( Z^{m,i} \right) \right) \right\} \\
= \text{mat} \left( R\{P^m_i P^m_i \text{pad} \text{vec} \left( Z^{m,i} \right) \} \right) \\
= \text{mat} \left( P^m_i P^m_i \text{pad} R\{\text{vec} \left( Z^{m,i} \right) \} \right) h^m h^m d^m \times a_{\text{conv}}^m b_{\text{conv}}^m
\]
From (17) and (18), we have

\[ \mathcal{R}\{ S^{m,i} \} \]
\[ = \mathcal{R}\{ W^m \phi(pad(Z^{m,i})) + b^m 1^T a_{conv} b_{conv} \} \]
\[ = \mathcal{R}\{ W^m \phi(pad(Z^{m,i})) \} + \mathcal{R}\{ b^m 1^T a_{conv} b_{conv} \} \]
\[ = \mathcal{R}\{ W^m \} \phi(pad(Z^{m,i})) + W^m \mathcal{R}\{ \phi(pad(Z^{m,i})) \} + \mathcal{R}\{ b^m \} 1^T a_{conv} b_{conv} \]
\[ = V^m_w \phi(pad(Z^{m,i})) + W^m \mathcal{R}\{ \phi(pad(Z^{m,i})) \} + \mathcal{R}\{ b^m \} 1^T a_{conv} b_{conv} , \]
R Operator for $J^i \mathbf{v} IV$

where we have

\[ R\{W^m\} = V^m_W, \]
\[ R\{b^m\} = v^m_b. \]

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 $J^i \nu \nu V$

- We further split $\nu^m$ to $V^m_W$ (a matrix form) and $\nu^m_b$.
- 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
  \[
  \mathcal{R}\{Z^{m+1,i}\} = \mathcal{R}\{\text{mat}(P^{m,i}_{\text{pool}} \text{vec}(\sigma(S^{m,i})))\}
  = \text{mat}(\mathcal{R}\{P^{m,i}_{\text{pool}} \text{vec}(\sigma(S^{m,i})))\})
  = \text{mat}\left(P^{m,i}_{\text{pool}} \mathcal{R}\{\text{vec}(\sigma(S^{m,i}))\}\right)_{d^{m+1} \times a^{m+1} b^{m+1}}.
  \]
R Operator for $J^i \mathbf{v}$ VI

- 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
Gauss-Newton Matrix-vector Product I

- From the above discussion, we have known how to calculate
  \[ J^i \mathbf{v} \]

- Calculate
  \[ B^i(J^i \mathbf{v}) \]

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

- Now for
  \[(J^i)^T (B^i J^i v),\]
  if we define
  \[u = B^i J^i v,\]
  then
  \[(J^i)^T u = \left( \frac{\partial z^{L+1,i}}{\partial \theta} \right)^T u.\]

- But earlier the gradient calculation is
  \[(J^i)^T \nabla_{z^{L+1,i}} \xi(z^{L+1,i}; y^i, Z^{1,i}) = \left( \frac{\partial z^{L+1,i}}{\partial \theta} \right)^T \frac{\partial \xi_i}{\partial z^{L+1,i}}.\]
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

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

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

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

For “# CG × 2”, the “2” is from (9) and (11)
Complexity Analysis V

- If using R operators, then

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

products are needed, where “3” are from the forward process

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

and

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

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 $Jv$
- 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