Summary of Operations I

- We show convolutional layers only and the bias term is omitted.
- Also we assume that RELU activation and max pooling are used.
- Operations in order

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

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

(2)

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

(3)

- Note that after (1), we change a vector \( \frac{\partial \xi_i}{\partial \text{vec}(S_{m,i})}^T \) to a matrix \( \frac{\partial \xi_i}{\partial S_{m,i}} \) because in (2) and (3), matrix form is needed.
- In (1), information of the next layer is used.
Instead we can do

\[ \frac{\partial \xi_i}{\partial \text{vec}(Z^{m,i})^T} \odot \text{vec}(I[Z^{m,i}])^T \]

in the end of the current layer.

This becomes the information passed to the previous layer.

Then only information in the current layer is used.
Finally an implementation for one convolutional layer:

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

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

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

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

A sample segment of code in MATLAB
Summary of Operations V

for m = LC : -1 : 1
    dXidS = reshape(vTP(param, model, net, m,
                   dXidS, 'pool_gradient'),
                   model.ch_input(m+1), []);

phiZ = padding_and_phiZ(model, net, m); 
net.dlossdW{m} = dXidS*phiZ';
net.dlossdb{m} = dXidS*ones(model.wd_conv(m)*
                          model.ht_conv(m)*S_k, 1);

if m > 1
v = model.weight{m}' * dXidS;
dXidS = vTP(model, net, m, num_data, v, 
   'phi_gradient');

% vTP_pad
dXidS = reshape(dXidS, model.ch_input(m), 
   model.ht_pad(m), 
   model.wd_pad(m), []);
p = model.wd_pad_added(m);
dXidS = dXidS(:, p+1:p+model.ht_input(m), 
   p+1:p+model.wd_input(m), :);
% activation function
\[ dXidS = \text{reshape}(dXidS, \text{model.ch_input}(m), []) \times (\text{net.Z}\{m\} > 0); \]
end
Storing $\phi(\text{pad}(Z^{m,i}))$

- From the above summary, we see that $\phi(\text{pad}(Z^{m,i}))$ is calculated twice in both forward and backward processes.
- If this expansion is expensive, we can store it.
- But memory is a concern as this is a huge matrix.
- So this setting of storing $\phi(\text{pad}(Z^{m,i}))$ trades space for time. It’s more suitable for CPU environments.
To see where the computational bottleneck is, it’s important to check the complexity of major operations.

- Assume $l$ is the number of data (for the case of calculating the whole gradient).
- For stochastic gradient, $l$ becomes the size of a mini-batch.
Complexity II

- Forward:

\[ S_{m,i}^m = W^m \text{mat}(P_{\phi}^m P_{pad}^m \text{vec}(Z_{m,i}^m)) = W^m \phi(\text{pad}(Z_{m,i}^m)) \]

\[ \phi(\text{pad}(Z_{m,i}^m)) : \mathcal{O}(l \times h^m h^m d^m a^m_{\text{conv}} b^m_{\text{conv}}) \quad (4) \]

\[ W^m \phi(\cdot) : \mathcal{O}(l \times d^{m+1} h^m h^m d^m a^m_{\text{conv}} b^m_{\text{conv}}) \]

\[ Z_{m+1,i}^m = \text{mat}(P_{\text{pool}}^{m,i} \text{vec}((\sigma(S_{m,i}^m)))) \]

\[ \mathcal{O}(l \times d^{m+1} a^m_{\text{conv}} b^m_{\text{conv}}) = \mathcal{O}(l \times h^m h^m d^{m+1} a^{m+1} b^{m+1}) \]
Complexity III

See also (4) as for pooling we also have a $\phi$ to generate sub-images

- Backward:

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

Size of $\Delta$ same as $S^{m,i}$ so cost is

$$O(l \times d^{m+1} a_{\text{conv}}^m b_{\text{conv}}^m)$$

$$\frac{\partial \xi_i}{\partial W^m} = \Delta^T \phi(\text{pad}(Z^{m,i}))$$
\[ \mathcal{O}(l \times d_{m+1} \times a_{\text{conv}}^m b_{\text{conv}}^m \times h^m h^m d^m) . \]

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

\[ (W^m)^T \Delta : \mathcal{O}(l \times h^m h^m d^m \times d_{m+1} a_{\text{conv}}^m b_{\text{conv}}^m) \]

\[ \text{vec}(\cdot) P^m_{\phi} : \mathcal{O}(l \times h^m h^m d^m a_{\text{conv}}^m b_{\text{conv}}^m) \] (5)

For (5) we convert a matrix of

\[ h^m h^m d^m \times a_{\text{conv}}^m b_{\text{conv}}^m \]

to a smaller matrix

\[ d^m \times a_{\text{pad}}^m b_{\text{pad}}^m \]
We see that matrix-matrix products are the bottleneck

If so, why check others?

The issue is that matrix-matrix products may be better optimized
Recall we have

$$Z^{m+1,i} = \text{mat}(P_{\text{pool}}^{m,i} \text{vec}(\sigma(S^{m,i})))_{d^{m+1} \times a^{m+1} b^{m+1}},$$

We note that

$$P_{\text{pool}}^{m,i}$$ is not a constant 0/1 matrix

It depends on $$\sigma(S^{m,i})$$ to decide the positions of 0 and 1.
Thus like the RELU activation function, max pooling is another place to cause that $f(\theta)$ is not differentiable.

However, it is almost differentiable around the current point.

Consider

$$f(A) = \max \left( \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \right)$$

and

$$A_{11} > A_{12}, A_{21}, A_{22}$$
Then

\[ \nabla f(A) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \text{at } A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \]

This explains why we can use \( P_{\text{pool}}^{m,i} \) in function and gradient evaluations.