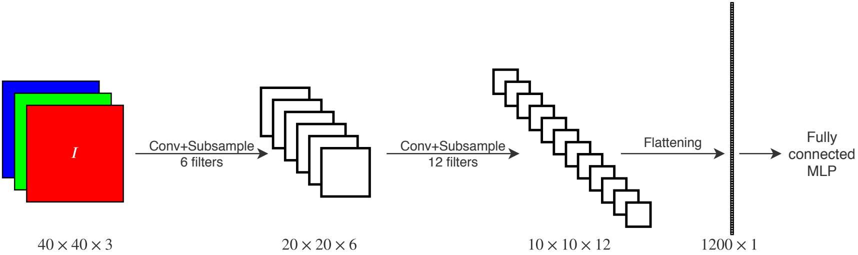
CS-568 Deep Learning

Nazar Khan

PUCIT

Backpropagation in a CNN

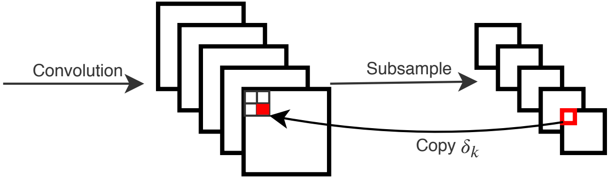
Backpropagation in CNNs



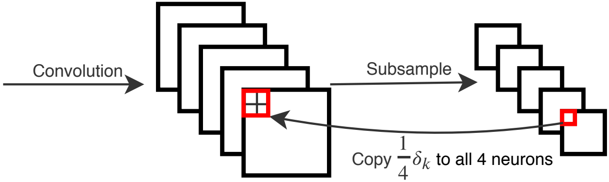
1. Compute $\delta_k = \frac{\partial L}{\partial a_k}$ for each neuron in flattened layer using standard MLP backpropagation.
2. *Directly copy* these δ_ks at corresponding locations of previous subsampling layer.

Backpropagation from subsampling to convolution layer

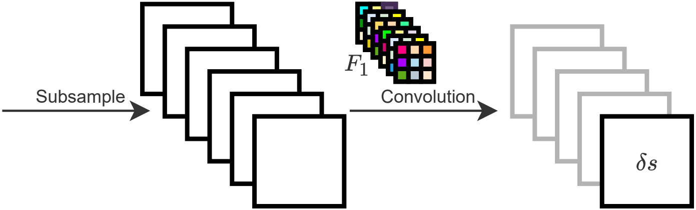
- Record index of pooled neuron during forward pass.
- Backpropagate δ only to this pooled neuron.



- Mean-pooling is different.
  - All neurons are picked with uniform weight in forward pass.
  - So backpropagate δ to each neuron with uniform weight.



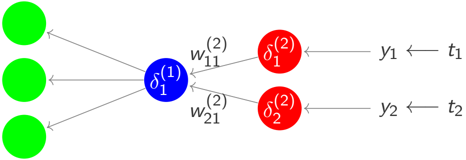
Backpropagation in a convolutional layer



Backpropagation Equation

Recall the backpropagation equation for a traditional neuron.

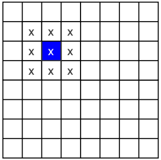
$$\delta_j^{(1)} = h'(a_j) \sum_{k=1}^{K} \delta_k^{(2)} w_{kj}$$



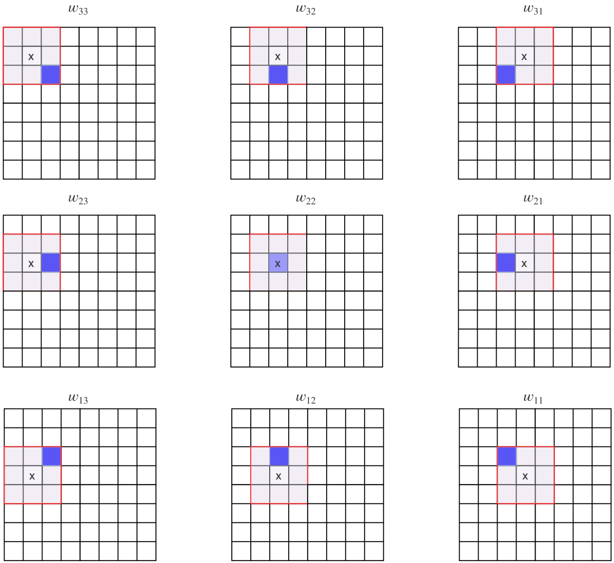
1. Take all neurons *affected* by neuron j.
2. Compute dot-product between their δ values and connecting weights.
3. Multiply result by derivative of activation function of neuron j.

Backpropagation in a convolutional layer

- Consider a neuron in a convolutional layer.
- In the forward pass, the blue neuron affects all neurons marked by x in the next layer.

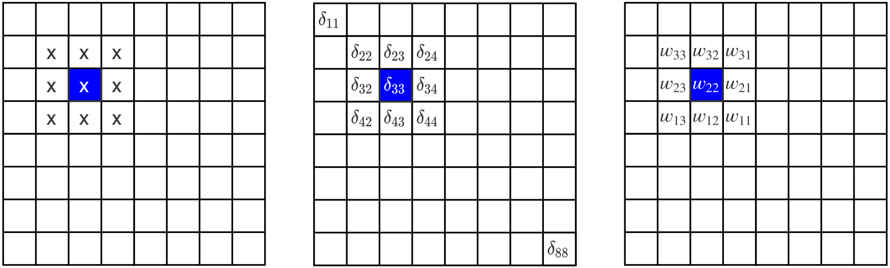


- *Notice the flipped role of weights*.



Backpropagation in a convolutional layer

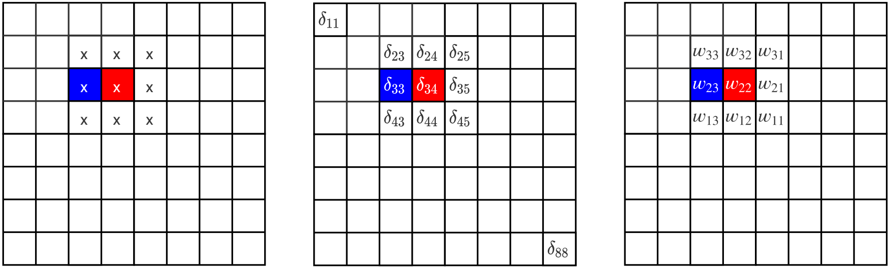
- In the backward pass, the blue neuron computes the dot-product between δ values at the x-locations and connecting weights.



- The connecting weights are a horizontally and vertically flipped version of the weights used in the forward convolution pass.

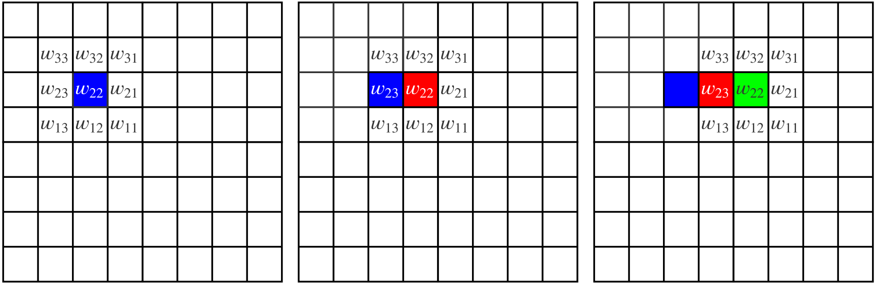
Backpropagation in a convolutional layer

- The adjacent red neuron affects a new but overlapping set of x-locations *using the same connecting weights*.



- *Since the weights are shared*, the only difference is between the x-locations.

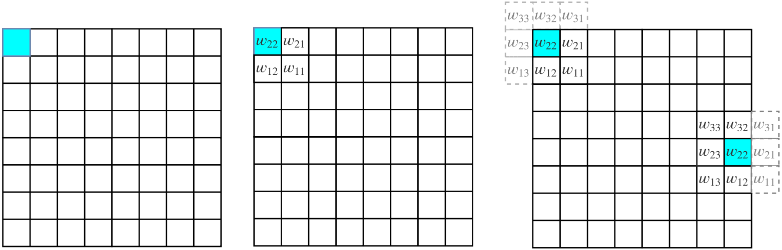
Backpropagation in a convolutional layer



- Equivalent to convolving the δ-map by flipped weights.
- Therefore, backpropagation of δ values from a convolution layer is
  1. *just a convolution of the δ-map using flipped weights*,
  2. followed by multiplication with derivatives of activation functions.

Backpropagation in a convolutional layer

- What about boundary neurons? Who did they affect?



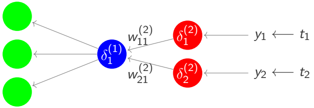
- Equivalent to convolving the δ-map by flipped weights *using zero-padding*.
- Therefore, backpropagation of δ values from a convolution layer is
  1. *just a convolution of the δ-map using flipped weights with zero-padding*,
  2. followed by multiplication with derivatives of activation functions.

Computing gradients in convolutional layer

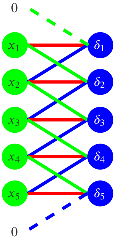
- Consider a *valid* convolution of an $n \times n$ array with another $n \times n$ array.
  - *Size* of the result will be 1×1.
- Now consider a *valid* convolution of an $(n+1) \times (n+1)$ array with an $n \times n$ array.
  - What will be the *size* of the result?
- Now consider a *valid* convolution of an $(n+2) \times (n+2)$ array with an $n \times n$ array.
  - What will be the *size* of the result?

Computing gradients in convolutional layer
1D case

- Backpropagation computes the per-neuron δ-maps only.
- Per-weight derivatives are computed as the product of a *traditional* neuron's δ value and its input.



- Consider 1D convolutional layer with 3×1 filter.

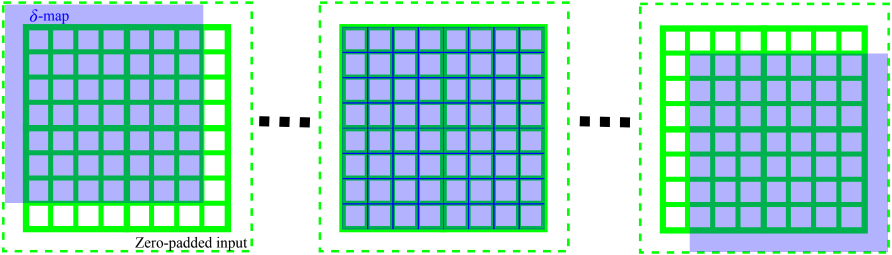


$$\left.\begin{aligned}\frac{\partial L}{\partial w_1} &= \delta_1 0 + \delta_2 x_1 + \delta_3 x_2 + \delta_4 x_3 + \delta_5 x_4 \\ \frac{\partial L}{\partial w_2} &= \delta_1 x_1 + \delta_2 x_2 + \delta_3 x_3 + \delta_4 x_4 + \delta_5 x_5 \\ \frac{\partial L}{\partial w_3} &= \delta_1 x_2 + \delta_2 x_3 + \delta_3 x_4 + \delta_4 x_5 + \delta_5 0\end{aligned}\right\} \Longrightarrow \begin{matrix}\begin{bmatrix}\delta_1 & \delta_2 & \delta_3 & \delta_4 & \delta_5\end{bmatrix} \\ \star \text{ (valid)} \\ \begin{bmatrix}0 & x_1 & x_2 & x_3 & x_4 & x_5 & 0\end{bmatrix}\end{matrix}$$

- Verify that $\frac{\partial L}{\partial b} = \sum \delta_i$.

Computing gradients in convolutional layer
2D case

1. Zero-pad the input array with $\lfloor \frac{K}{2} \rfloor$ zeros on each side[1].
2. Perform *valid* convolution of the *zero-padded input array* by the δ-map of the next layer to obtain a $K \times K$ array of derivatives of the convolution weights.



3. Derivative of bias is just the sum of the δ-map.

[1] Assuming square $K \times K$ convolution filter where K is odd

Summary

- From FC to Subsampling:

  Direct copying of δ-values.

- From Subsampling to Conv:

  Direct copy or weighted combination of δ-map.

- From Conv:

  conv2d(zeropad(δ-map), fliplr(flipud(F)), 'valid')

- Gradients of convolution filter F:

  conv2d(zeropad(input array), δ-map, 'valid')

- Gradient of bias:

  sum of δ-map