Backpropagations in Deep Neural Network
-focused on Convolutional NN -

Kyong-Ha Lee
(bart7449@gmail.com)

May 14, 2021

Why we need backpropagation?

- Partial derivative of the loss function wrt. to each intermediate weights
  - Simple with single-layer architecture like the perceptron
  - Not a simple matter with multi-layer architecture
- Complexity of computational graphs
  - Neural network is a special case of a computational graph
    - A direct acyclic graph where each node computes a function of its incoming node variables
    - Each node computes a comb. of a linear vector multiplication & a (possibly nonlinear) activation function
  - Output is a very complicated composition function of each intermediate weight in the network
  - Hard to express neatly in closed form.
  - Difficult to differentiate!!

Recursive nesting is ugly!

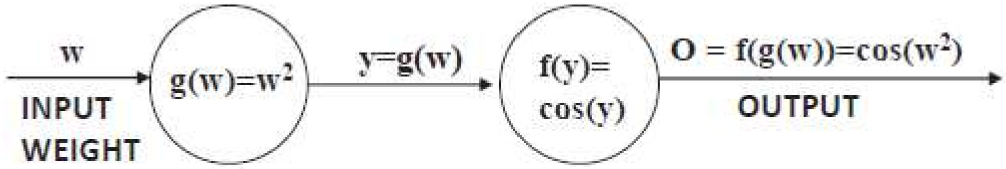
- A computational graph that contains only two nodes in a path and input w.
  - First node computes $y = g(w)$ and the second node computes the output $o = f(y)$
  - Overall composition function is $o = f(g(w))$
  - Setting $f()$ and $g()$ to the sigmoid results in the following

$$f(g(w)) = \frac{1}{1 + \exp\left[-\frac{1}{1+\exp(-w)}\right]}$$

  - Increasing path length increases recursive nesting

Backpropagation along single path

- Univariate chain rule



  - Consider a two-node path with $f(g(w)) = \cos(w^2)$
  - Compute the product of local derivatives

$$\frac{\partial f(g(w))}{\partial w} = \underbrace{\frac{\partial f(y)}{\partial y}}_{-sin(y)} \cdot \underbrace{\frac{\partial g(w)}{\partial w}}_{2w} = -2w \cdot \sin(y) = -2w \cdot \sin(w^2)$$

- Local derivatives are easier to compute
  - Each derivative consider only its input and outputs

Backpropagation along multiple paths

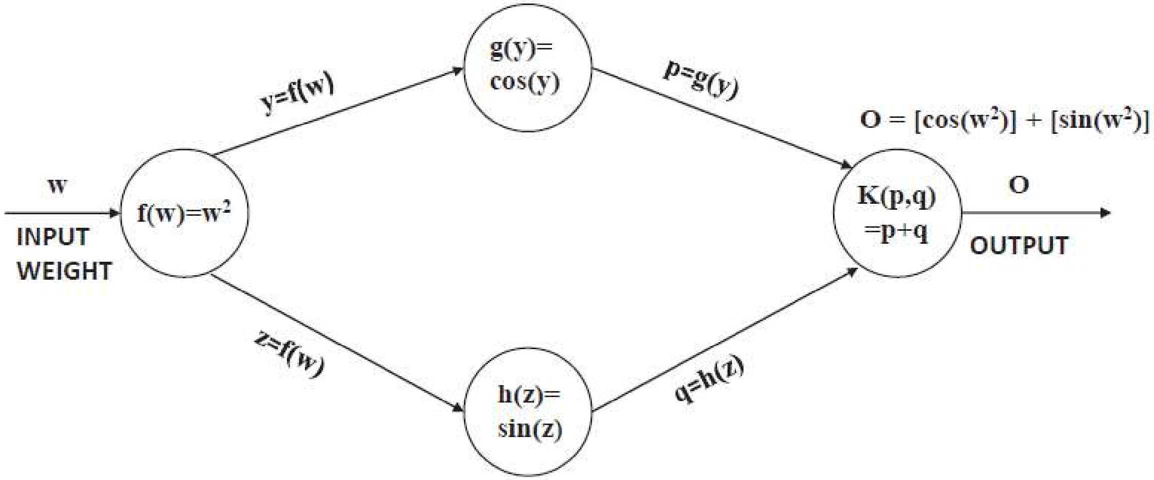
- NN contain multiple nodes in each layer

- Consider the function $f\left(g_1(w), \ldots, g_k(w)\right)$
  – A unit computing the multivariate function $f(\cdot)$ gets inputs from k units computing $g_1(w), \ldots, g_k(w)$

- Multivariate chain rule

$$\frac{\partial f(g_1(w), \ldots g_k(w))}{\partial w} = \sum_{i=1}^{k} \frac{\partial f(g_1(w), \ldots g_k(w))}{\partial g_i(w)} \cdot \frac{\partial g_i(w)}{\partial w}$$

Example of multivariate chain rule



- $\frac{\partial o}{\partial w} = \frac{\partial o}{\partial p} \cdot \frac{\partial p}{\partial y} \cdot \frac{\partial y}{\partial w} + \frac{\partial o}{\partial q} \cdot \frac{\partial q}{\partial z} \cdot \frac{\partial z}{\partial w}$

$$\begin{aligned}\frac{\partial o}{\partial w} &= \underbrace{\frac{\partial K(p,q)}{\partial p}}_{1} \cdot \underbrace{g'(y)}_{\text{-}\sin(y)} \cdot \underbrace{f'(w)}_{2w} + \underbrace{\frac{\partial K(p,q)}{\partial q}}_{1} \cdot \underbrace{h'(z)}_{\cos(z)} \cdot \underbrace{f'(w)}_{2w} \\ &= -2w \cdot \sin(y) + 2w \cdot \cos(z) \\ &= -2w \cdot \sin(w^2) + 2w \cdot \cos(w^2)\end{aligned}$$

- Product of local derivatives along all paths from w to a.

Path-wise Aggregation Lemma

- Let a non-null set P of paths exist from a variable w in the computational graph to output o
  - Local gradient of node with variable $y(j)$ with respect to variable $y(i)$ for directed edge (i, j) is $z(i, j) = \frac{\partial y(i)}{\partial y(i)}$
- The value of $\frac{\partial o}{\partial w}$ is computed by the product of local gradients along each path in P, and summing these products over all paths

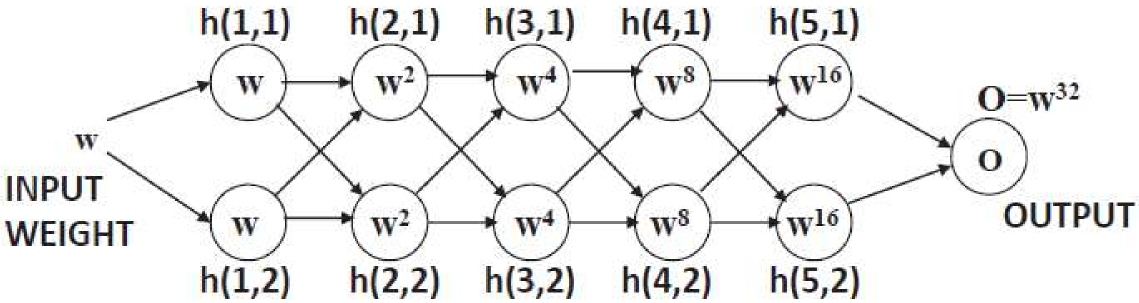
$$S(w, o) = \frac{\partial o}{\partial w} = \sum_{P \in \mathcal{P}} \prod_{(i,j) \in P} z(i, j)$$

EXPTIME algorithm for computing partial derivatives

- The path aggregation lemma provides a simple way to compute the derivative wrt. Intermediate variable w
  - Use computational graph to compute each value $y(i)$ of nodes i in a forward phase

1. Compute local derivative $z(i,j) = \frac{\partial y(j)}{\partial y(i)}$ on each edge (i,j) in the network
2. Identify the set P of all paths from the node with variable w to the output o
3. For each path $p \in P$, compute the product $M(P) = \prod_{(i,j) \in P} z(i,j)$ of the local derivatives on that path
4. Add up these values over all paths $p \in P$

Example : Deep computational graph with product nodes



EACH NODE COMPUTES THE PRODUCT OF ITS INPUTS

$$\frac{\partial O}{\partial w} = \sum_{j_1,j_2,j_3,j_4,j_5 \in \{1,2\}^5} \prod \underbrace{h(1,j_1)}_{w} \underbrace{h(2,j_2)}_{w^2} \underbrace{h(3,j_3)}_{w^4} \underbrace{h(4,j_4)}_{w^8} \underbrace{h(5,j_5)}_{w^{16}}$$
$$= \sum_{\text{All 32 paths}} w^{31} = 32w^{31}$$

- Impractical with increasing depth
  - Million paths for 3-layered NN with 100 nodes in each layer
- *Dynamic programming* idea of backpropagation rescues us from the complexity

Differentiating Composition Functions

- Repetitiveness
  - NN compute composition functions repeatedly, caused by nodes appearing in multiple paths
  - Natural and intuitive way to differentiate such functions is not most efficient
- Natural approach : top-down
  - $f(w) = \sin(w^2) + \cos(w^2)$
  - Not need to differentiate w^2 twice
- Dynamic programming collapses repetitive computations to reduce EXPTIME into PTIME complexity

Dynamic programming update

- $A(i)$: the set of nodes at the ends of outgoing edges from node i
- $S(i, o)$: the intermediate variable indicating the same path aggregative function from i to o

$$S(i, o) \Leftarrow \sum_{j \in A(i)} \boxed{S(j, o)} \cdot z(i, j)$$

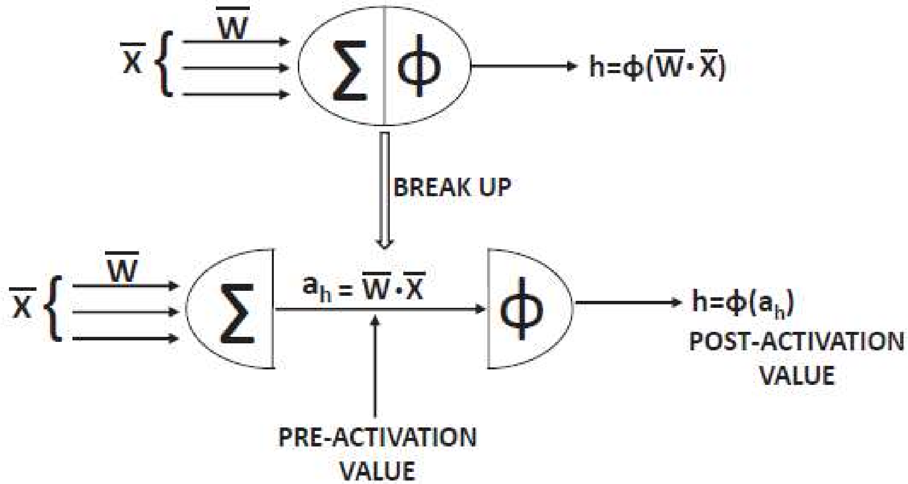
Recursive call

- Initialize $S(o, o)$ to 1 and compute backwards to reach $S(w, o)$
  - Intermediate computations $S(i, o)$ are also useful for computing derivatives in other layers

$$\frac{\partial o}{\partial y(i)} = \sum_{j \in A(i)} \frac{\partial o}{\partial y(j)} \cdot \frac{\partial y(j)}{\partial y(i)}$$

Pre-activation variables to create Computational graph

- Compute derivative $\delta(i, o)$ of loss L at o wrt. pre-activation variable at node i
- We always compute loss derivatives $\delta(i, o)$ wrt. *activations* in *nodes* during dynamic program -ing rather than weights
  - Loss derivative wrt. weight *w* from node i to node j is given by the product of $\delta(i, o)$ and hidden variable at i



- Key points: $z(i,j) = w_{ij} \cdot \Phi_i'$, Initialize $S(o,o) = \delta(o,o) = \frac{\partial L}{\partial o}\Phi_o'$

$$\delta(i,o) = S(i,o) = \Phi_i' \sum_{j \in A(i)} w_{ij} S(j,o) = \Phi_i' \sum_{j \in A(i)} w_{ij}\delta(j,o)$$

Post-activation variables to create computation graph

- The variables in the computation graph are hidden values *after* activation function
- Compute derivative $\Delta(i, o)$ of loss L at o wrt. Post-activation variable at node i
- Key points: $z(i,j) = w_{ij} \cdot \Phi'_j$, Initialize $S(o,o) = \Delta(o,o) = \frac{\partial L}{\partial o}$

$$\Delta(i,o) = S(i,o) = \sum_{j \in A(i)} w_{ij} S(j,o) \Phi'_j = \sum_{j \in A(i)} w_{ij} \Delta(j,o) \Phi'_j$$

  – Comparison to the pre-activation approach

$$\delta(i,o) = \Phi'_i \sum_{j \in A(i)} w_{ij} \delta(j,o)$$

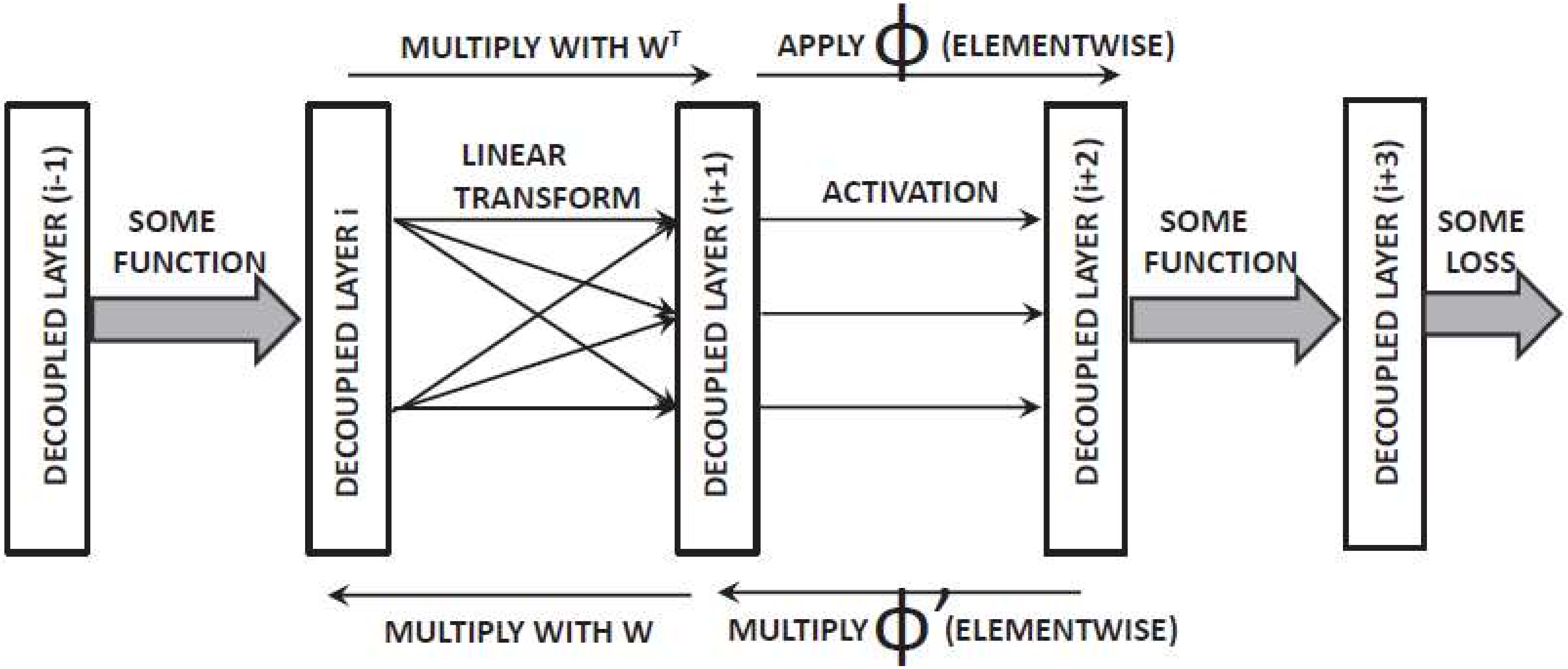
Variables for both pre-activation and post-activation values

- Computational graph from the variables in NN
  - Graph of pre-activation variables
  - Graph of post-activation variables
  - Graph of both
- Using both pre-activation and post-activation variables
  - A nice way of decoupling linear layer (matrix multiplication) and activation function during backpropagation

- Simplified approach where each layer is treated as ***a single node with a vector variable***
  - Update can be computed in vector and matrix multiplications
  - NN are back-propagated by using ***layer-wise on vectors***
    - Most real implementations follow this approach
  - We can treat an entire layer as a node with a vector variable

Vector-centric and decoupled view of single layer

- Linear matrix multiplication and activation function are separate
- Use of vector-to-vector chain rule to backpropagate on a single path



Converting scalar updates to vector form

- **Recap** : when the partial derivative of node q wrt. to node p is $z(p,q)$, the dynamic programming update is:

$$S(p,o) = \sum_{q \in Next\ Layer} S(q,o) \cdot z(p,q)$$

- We can write the above update in vector form by creating a single column vector $\bar{g}_i$ for layer $i \Longrightarrow$ contains $S(p,o)$ for all values of p

$$\bar{g}_i = Z g_{i+1}$$

- The matrix $Z = [z(p,q)]$ is the transpose of the Jacobian matrix J , *i.e.*, $\mathrm{Z} = J^T$

The Jacobian matrix[1/2]

- Suppose $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ is a function such that each of its fir-order partial derivatives exist on $\mathbb{R}^n$
- Jacobian matrix defined to be an $m \times n$ matrix, denoted by J, whose (k, r)-th entry is $J_{kr} = \frac{\partial f_k}{\partial x_r}$

$$F_1(x_1, \ldots, x_n), \ldots, F_m(x_1, \ldots, x_n)$$

$$J = \begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial x_1} & \cdots & \frac{\partial F_m}{\partial x_n} \end{bmatrix}$$

- Example

$$F(x, y) = \begin{bmatrix} x^2 y \\ 5x + \sin(y) \end{bmatrix} \qquad J_F(x, y) = \begin{bmatrix} \frac{\partial F_1}{\partial x} & \frac{\partial F_1}{\partial y} \\ \frac{\partial F_2}{\partial x} & \frac{\partial F_2}{\partial y} \end{bmatrix} = \begin{bmatrix} 2xy & x^2 \\ 5 & \cos(y) \end{bmatrix}$$

The Jacobian matrix[2/2]

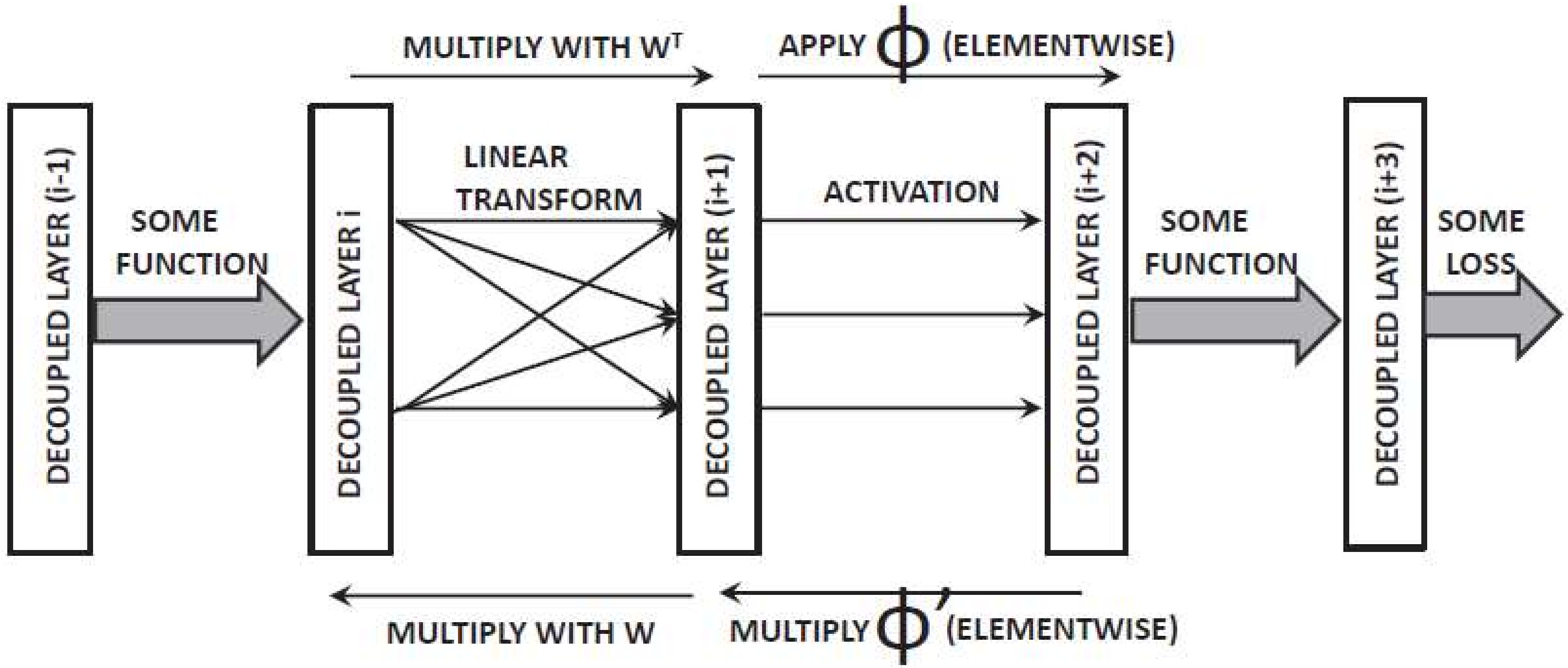
- Consider layer i and layer $(i+1)$ with activations $\overline{z_i}$ and z_{i+1}
  - The k-th activation in layer $(i+1)$ is obtained by applying an arbitrary function $f_k(\cdot)$ on the vector of activations in layer i
- Then, Jacobian matrix entries is

$$J_{kr} = \frac{\partial f_k(\bar{z}_i)}{\partial \bar{z}_i{}^{(r)}}$$

- Backpropagation updates:

$$\overline{g}_i = J^T \overline{g}_{i+1}$$

Effect on linear layer and activation function



– Multiplication with transposed weight matrix for linear layer
– Element-wise multiplication with derivative for activation layer

Forward & Backward propagation

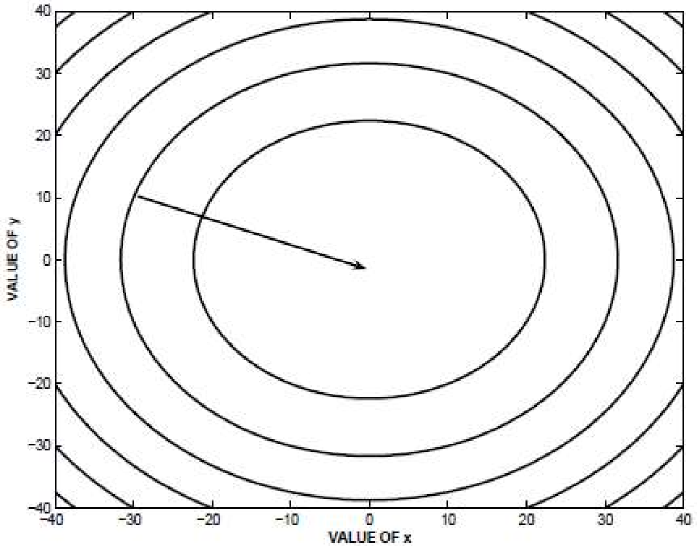
Function	Forward	Backward
Linear	$\overline{z}_{i+1} = W^T \overline{z}_i$	$\overline{g}_i = W \overline{g}_{i+1}$
Sigmoid	$\overline{z}_{i+1}$ =sigmoid($\overline{z}_i$)	$\overline{g}_i = \overline{g}_{i+1} \odot \overline{z}_{i+1} \odot (1 - \overline{z}_{i+1})$
Tanh	$\overline{z}_{i+1}$ =tanh($\overline{z}_i$)	$\overline{g}_i = \overline{g}_{i+1} \odot (1 - \overline{z}_{i+1} \odot \overline{z}_{i+1})$
ReLU	$\overline{z}_{i+1} = \overline{z}_i \odot I(\overline{z}_i > 0)$	$\overline{g}_i = \overline{g}_{i+1} \odot I(\overline{z}_i > 0)$
Hard Tanh	Set to ± 1 ($\notin [-1, +1]$) Copy ($\in [-1, +1]$)	Set to 0 ($\notin [-1, +1]$) Copy ($\in [-1, +1]$)
Max	Maximum of inputs	Set to 0 (non-maximal inputs) Copy (maximal input)
Arbitrary function $f_k(\cdot)$	$\overline{z}_{i+1}^{(k)} = f_k(\overline{z}_i)$	$\overline{g}_i = J^T \overline{g}_{i+1}$ J is Jacobian (Equation 10)

- Two types of Jacobians
  - Linear layers are dense and activation layers are sparse
  - Maximization function is used in max-pooling

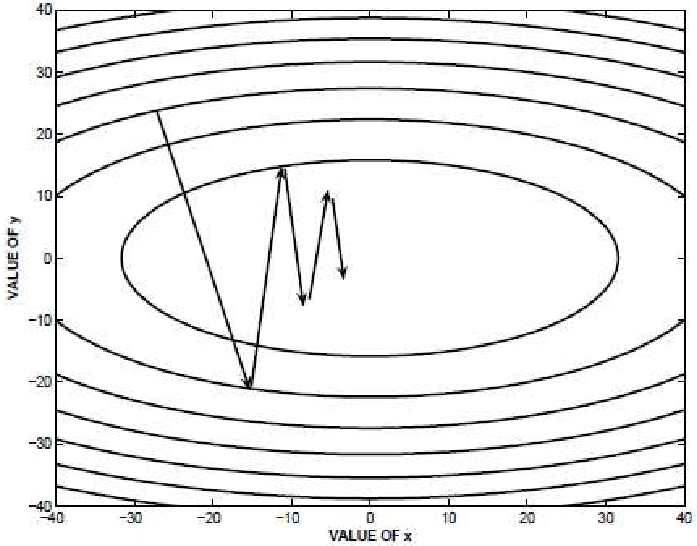
Effect of varying slopes in gradient descent

- Neural network learning is a *multivariable* optimization problem
- Different weights have different magnitudes of partial derivatives
- Widely varying magnitudes of partial derivatives affect learning
- Gradient descent works best when the different weights have derivatives of similar magnitude
  - The path of steepest descent in most loss functions is only an instantaneous direction of best movement, not the correct direction of descent in the longer term

Example



Loss function is circular bowl
$$L = x^2 + y^2$$



Loss function is elliptical bowl
$$L = x^2 + 4y^2$$

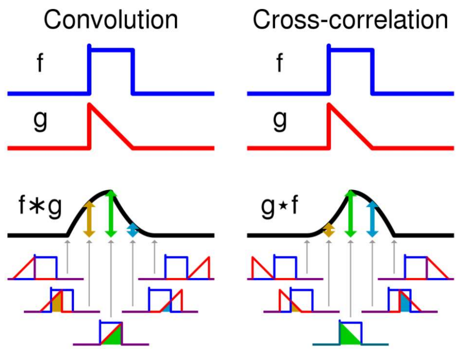
- Loss functions with varying sensitivity to different attributes

Vanishing and exploding gradient problems

- An extreme manifestation of varying sensitivity occurs in deep NN
- The weights/activation derivatives in different layers affect the back-propagated gradient in a multiplicative way
  - With increasing depth, this effect is magnified
  - Partial derivatives can either increase or decrease with depth

- Some fixes
    - Stronger initializations with pre-training
    - Second-order learning methods that make use of second order derivatives(or *curvature* of the loss function)

Convolution & Cross correlation



- Cross-correlation
  - Given an input image *I* and filter(kernel) *K* of dimensions $k_1 \times k_2$,

$$(I \otimes K)_{ij} = \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} I(i+m, j+n)K(m,n) \quad (1)$$

- Convolution
  - Given an input image *I* and filter(kernel) *K* of dimensions $k_1 \times k_2$,

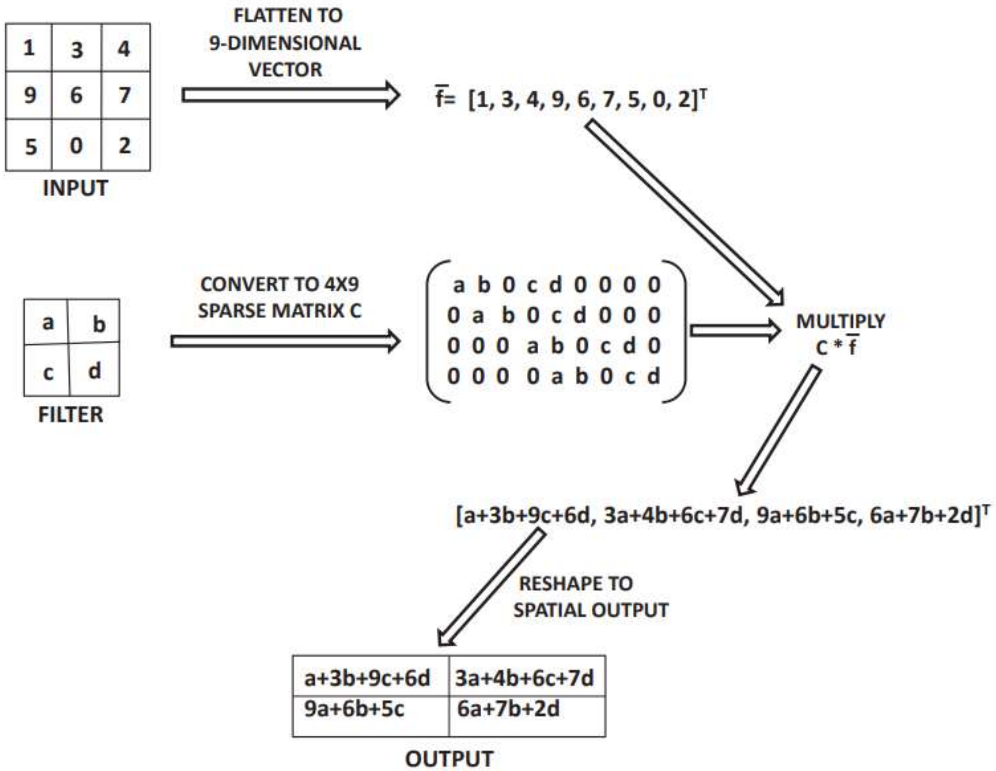
$$(I * K)_{ij} = \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} I(i-m, j-n)K(m,n) \quad (2)$$

$$= \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} I(i+m, j+n)K(-m,-n) \quad (3)$$

  - Convolution is the same as cross-correlation with a flipped kernel for a kernel K where $(-m, -n) \ == \ K(m,n)$

Convolution as a matrix multiplication

- Convolution can be presented as a matrix multiplication.
  - Useful during forward and backward propagation.
  - Backward propagation can be presented as transposed matrix multiplication



Convolutional neural Network

- Convolutional procedure in CNN

$$(I * K)_{ij} = \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} \sum_{c=1}^{C} K_{m,n,c} \cdot I_{i+m,j+n,c} + b \qquad (4)$$

  – channel C, input map $\mathrm{I} \in \mathbb{R}^{H \times W \times C}$ with height H, width W, a bank of filters K of dimension $k_1 \times k_2$ and biases $\mathrm{b} \in \mathbb{R}^{D}$
  – Same as cross-correlation, except that kernel is “flipped”

- For simplicity, we set C=1. Then,

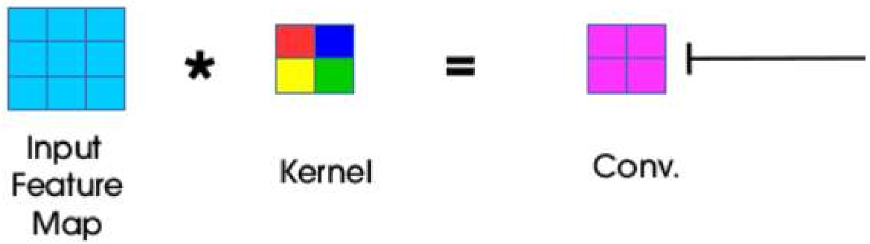
$$(I * K)_{ij} = \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} K_{m,n} \cdot I_{i+m,j+n} + b \qquad (5)$$

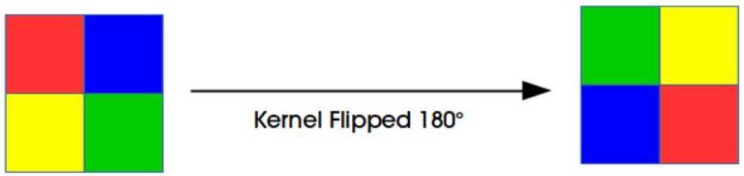
Notations

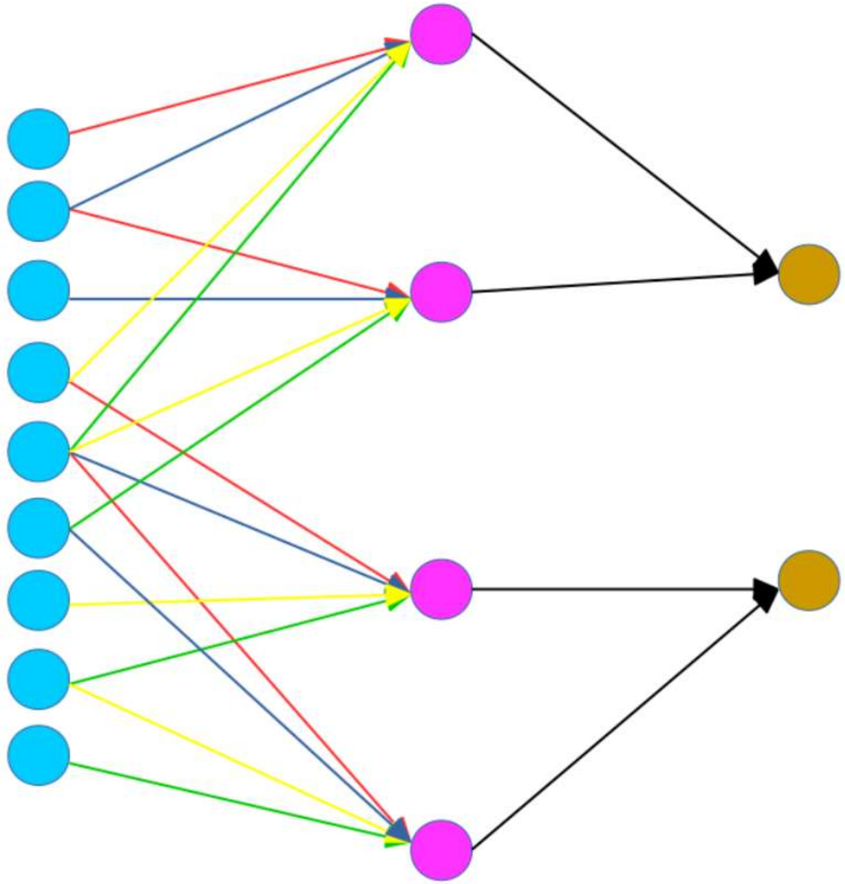
- L : l-th layer. $l = L$ is the last(output) layer
- $w_{m,n}^{l}$: weight matrix connecting neurons of layer l with neurons of layer $l-1$
- $x_{i,j}^{l}$: convolved input vector at layer l plus the bias represented as $x_{i,j}^{l} = \sum_{m}\sum_{n} w_{m,n}^{l} o_{i+m,j+n}^{l-1} + b^{l}$
- $o_{i,j}^{l}$: output vector at layer l given by $o_{i,j}^{l} = f(x_{i,j}^{l})$
- $f(\cdot)$: activation function. Application of the activation to the convolved input vector at layer l is given by $f(x_{i,j}^{l})$

Forward propagation (1/2)

- Kernel is flipped and slide across the input feature map





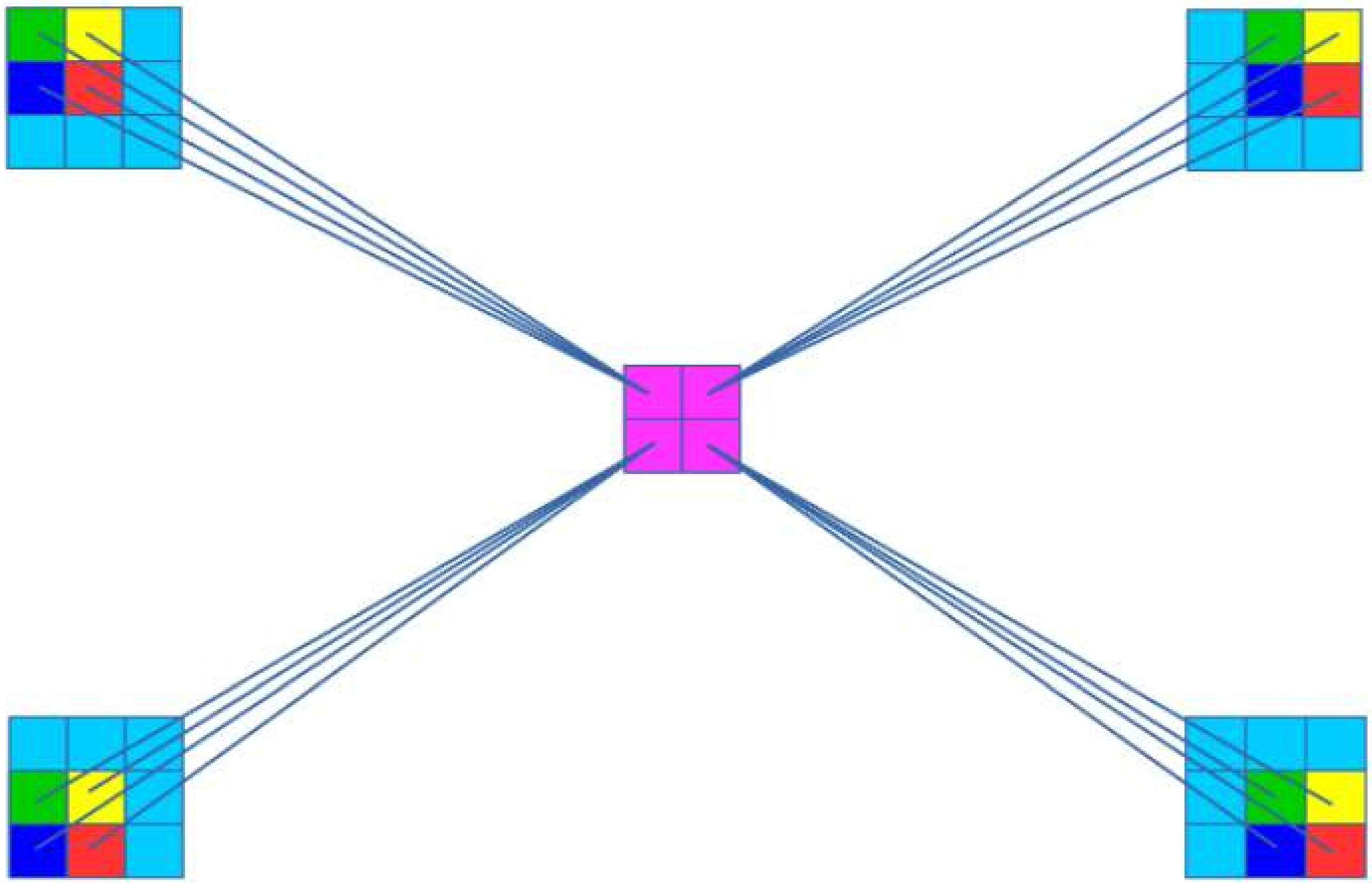


$$x_{i,j}^{l} = \mathrm{rot}_{180^\circ}\left\{w_{m,n}^{l}\right\} * o_{i,j}^{l-1} + b_{i,j}^{l} \tag{6}$$

$$x_{i,j}^{l} = \sum_{m}\sum_{n} w_{m,n}^{l} o_{i+m,j+n}^{l-1} + b_{i,j}^{l} \tag{7}$$

$$o_{i,j}^{l} = f(x_{i,j}^{l}) \tag{8}$$

Forward propagation (2/2)



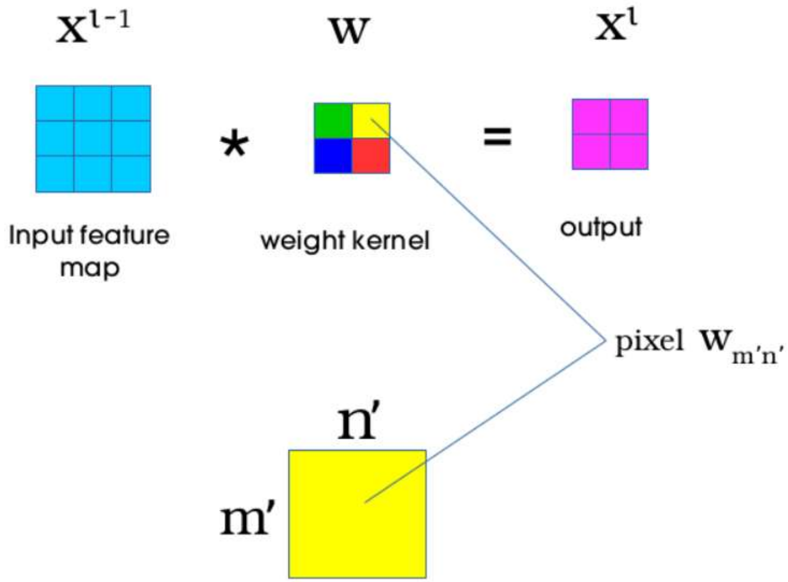
Backpropagation (1/9)

- Error
  - Assume to use MSE; For a total of P prediction, predicted outputs y and targeted outputs t,

$$E = \frac{1}{2}\sum_{i}^{P}(\,t_i - y_i)^{\,2}$$

Two updates for weights and deltas.

- $\frac{\partial E}{\partial w^{l}_{m',n'}}$: the measurement of how the change in a single pixel $w^{l}_{m',n'}$ in the weight kernel affects the loss function E



Backpropagation (2/9)

- Convolution btw. input map of dimension $H \times W$ and the weight kernel of dimension $k_1 \times k_2$ produces the output feature map of size $(H - k_1 - 1) \times (H - k_2 - 1)$
- Gradient component for individual weight

$$\frac{\partial E}{\partial w^l_{m',n'}} = \sum_{i=0}^{H-k_1} \sum_{j=0}^{W-k_2} \frac{\partial E}{\partial x^l_{i,j}} \frac{\partial x^l_{i,j}}{\partial w^l_{m',n'}}$$
$$= \sum_{i=0}^{H-k_1} \sum_{j=0}^{W-k_2} \delta^l_{i,j} \frac{\partial x^l_{i,j}}{\partial w^l_{m',n'}} \tag{10}$$

– In Eq. 10, $x^l_{i,j}$ is equivalent to $\sum_m \sum_n w^l_{m,n}\, o^{l-1}_{i+m,j+n} + b^l$

$$\frac{\partial x^l_{i,j}}{\partial w^l_{m',n'}} = \frac{\partial}{\partial w^l_{m',n'}} \left(\sum_m \sum_n w^l_{m,n} o^{l-1}_{i+m,j+n} + b^l \right) \tag{11}$$

Backpropagation (3/9)

$$\begin{aligned}\frac{\partial x_{i,j}^{l}}{\partial w_{m',n'}^{l}} &= \frac{\partial}{\partial w_{m',n'}^{l}}\left(w_{0,0}^{l}o_{i+0,j+0}^{l-1} + \cdots + w_{m',n'}^{l}o_{i+m',j+n'}^{l-1} + \cdots + b^{l}\right)\\ &= \frac{\partial}{\partial w_{m',n'}^{l}}\left(w_{m',n'}^{l}o_{i+m',j+n'}^{l-1}\right)\\ &= o_{i+m',j+n'}^{l-1} \qquad (12)\end{aligned}$$

- Substituting Eq. 12 in Eq.10 give the following results:

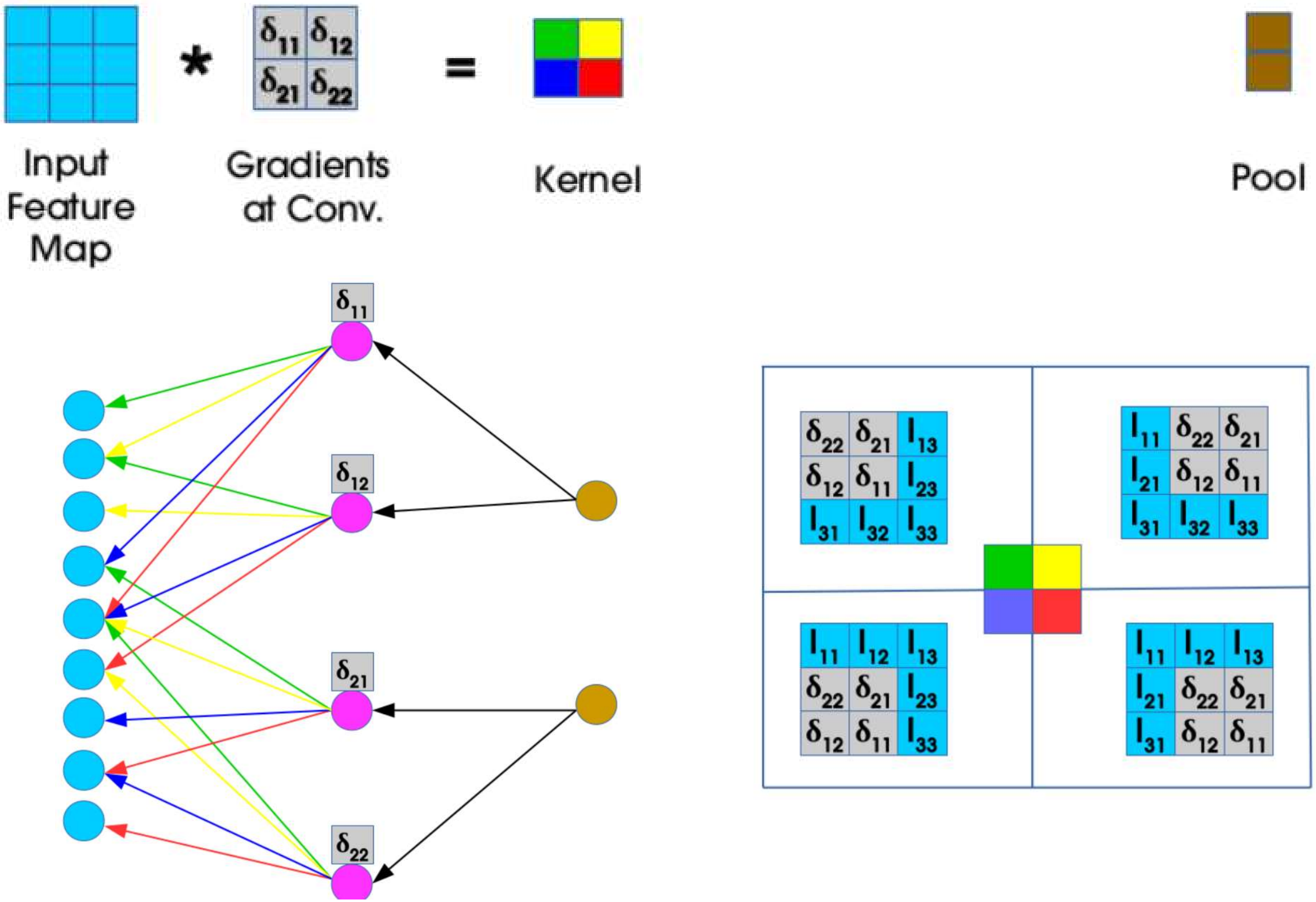
$$\frac{\partial E}{\partial w_{m',n'}^{l}} = \sum_{i=0}^{H-k_1}\sum_{j=0}^{W-k_2}\delta_{i,j}^{l}o_{i+m',j+n'}^{l-1} \qquad (13)$$

$$= \mathrm{rot}_{180^\circ}\left\{\delta_{i,j}^{l}\right\} * o_{m',n'}^{l-1} \qquad (14)$$

$$\begin{bmatrix}\delta_{11} & \delta_{12}\\ \delta_{21} & \delta_{22}\end{bmatrix} \xrightarrow[\text{Error matrix Flipped }180^\circ]{} \begin{bmatrix}\delta_{22} & \delta_{21}\\ \delta_{12} & \delta_{11}\end{bmatrix}$$

  – Eq.13 is a result of weight sharing in the network.
    - A collection of all the gradient $\delta_{i,j}^{l}$ coming from all the outputs in layer l

Backpropagation (4/9)

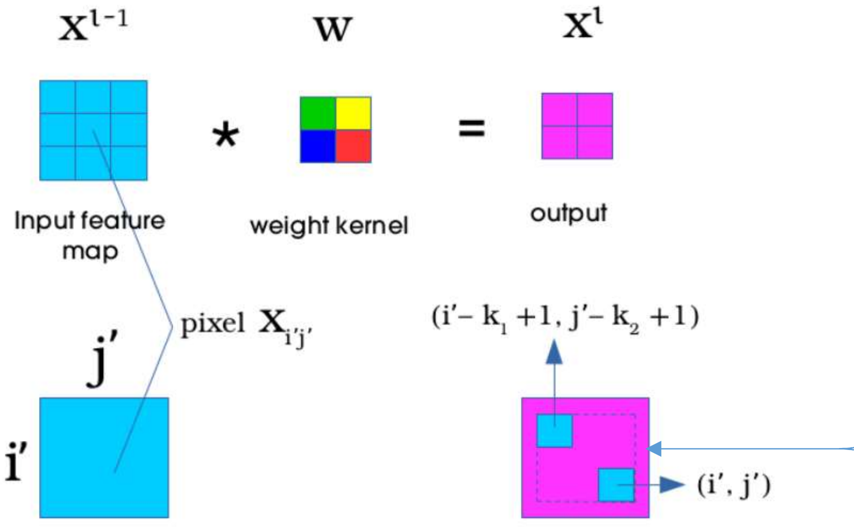


Backpropagation (5/9)

- Deltas are provided by an equation of the form

$$\delta^{l}_{i,j} = \frac{\partial E}{\partial x^{l}_{i,j}} \qquad (15)$$

- $\frac{\partial E}{\partial x^{l}_{i',j'}}$: measurement of how the change in a single pixel $x^{l}_{i',j'}$ in the input map affects the loss function E



Regions in the output bounded by the dashed lines are affected by pixel $x_{i'j'}$

Backpropagation (6/9)

$$\begin{aligned}\frac{\partial E}{\partial x^{l}_{i',j'}} &= \sum_{i,j \in Q} \frac{\partial E}{\partial x^{l+1}_{Q}} \frac{\partial x^{l+1}_{Q}}{\partial x^{l}_{i',j'}} \\ &= \sum_{i,j \in Q} \delta^{l+1}_{Q} \frac{\partial x^{l+1}_{Q}}{\partial x^{l}_{i',j'}} \qquad (16)\end{aligned}$$

- Where Q represents the output region bounded by dashed lines
  - composed of pixels in the output that are affected by the single pixel $x_{i' j'}$

Backpropagation (7/9)

- A more formal way is:

$$\frac{\partial E}{\partial x^{l}_{i',j'}} = \sum_{m=0}^{k_1-1}\sum_{n=0}^{k_2-1} \frac{\partial E}{\partial x^{l+1}_{i'-m,j'-n}} \frac{\partial x^{l+1}_{i'-m,j'-n}}{\partial x^{l}_{i',j'}}$$
$$= \sum_{m=0}^{k_1-1}\sum_{n=0}^{k_2-1} \delta^{l+1}_{i'-m,j'-n} \frac{\partial x^{l+1}_{i'-m,j'-n}}{\partial x^{l}_{i',j'}} \qquad (17)$$

- In the region Q, the height ranges from $i' - 0\ to\ i' - (k_1 - 1)$ and width $j' - 0$ to $j' - (k_2 - 1)$

Backpropagation (8/9)

In Eq. 17 , $x^{l+1}_{i'-m,j'-n}$ is equivalent to $\sum_{m'}\sum_{n'} w^{l+1}_{m',n'} o^{l}_{i'-m+m',j'-n+n'} + b^{l+1}$ and expanding this part of the equation gives us:

$$\begin{aligned}\frac{\partial x^{l+1}_{i'-m,j'-n}}{\partial x^{l}_{i',j'}} &= \frac{\partial}{\partial x^{l}_{i',j'}}\left(\sum_{m'}\sum_{n'} w^{l+1}_{m',n'} o^{l}_{i'-m+m',j'-n+n'} + b^{l+1}\right) \\ &= \frac{\partial}{\partial x^{l}_{i',j'}}\left(\sum_{m'}\sum_{n'} w^{l+1}_{m',n'} f\left(x^{l}_{i'-m+m',j'-n+n'}\right) + b^{l+1}\right) \qquad (18)\end{aligned}$$

$$\begin{aligned}\frac{\partial x^{l+1}_{i'-m,j'-n}}{\partial x^{l}_{i',j'}} &= \frac{\partial}{\partial x^{l}_{i',j'}}\left(w^{l+1}_{m',n'} f\left(x^{l}_{0-m+m',0-n+n'}\right) + \cdots + w^{l+1}_{m,n} f\left(x^{l}_{i',j'}\right) + \cdots + b^{l+1}\right) \\ &= \frac{\partial}{\partial x^{l}_{i',j'}}\left(w^{l+1}_{m,n} f\left(x^{l}_{i',j'}\right)\right) \\ &= w^{l+1}_{m,n}\frac{\partial}{\partial x^{l}_{i',j'}}\left(f\left(x^{l}_{i',j'}\right)\right) \\ &= w^{l+1}_{m,n} f'\left(x^{l}_{i',j'}\right) \qquad (19)\end{aligned}$$

Backpropagation (9/9)

Substituting Eq. 19 in Eq. 17 gives us the following results:

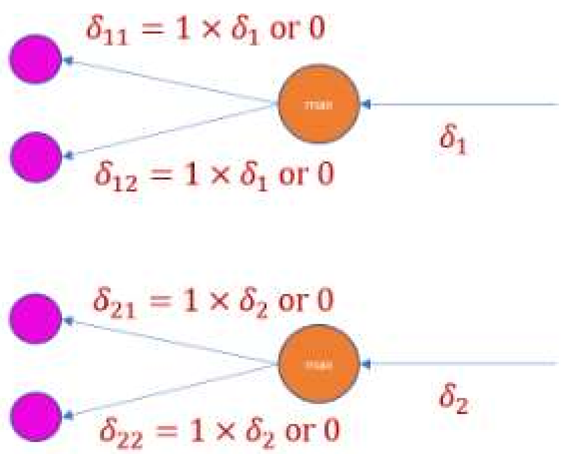
$$\frac{\partial E}{\partial x^{l}_{i',j'}} = \sum_{m=0}^{k_1-1}\sum_{n=0}^{k_2-1} \delta^{l+1}_{i'-m,j'-n} w^{l+1}_{m,n} f'\left(x^{l}_{i',j'}\right) \tag{20}$$

For backpropagation, we make use of the flipped kernel and as a result we will now have a convolution that is expressed as a cross-correlation with a flipped kernel:

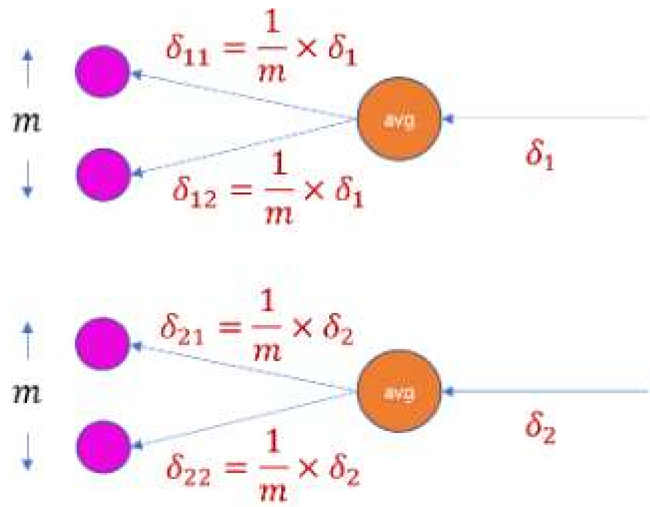
$$\begin{aligned}\frac{\partial E}{\partial x^{l}_{i',j'}} &= \sum_{m=0}^{k_1-1}\sum_{n=0}^{k_2-1} \delta^{l+1}_{i'-m,j'-n} w^{l+1}_{m,n} f'\left(x^{l}_{i',j'}\right) \\ &= \mathrm{rot}_{180^\circ}\left\{\sum_{m=0}^{k_1-1}\sum_{n=0}^{k_2-1} \delta^{l+1}_{i'+m,j'+n} w^{l+1}_{m,n}\right\} f'\left(x^{l}_{i',j'}\right) &\quad (21) \\ &= \delta^{l+1}_{i',j'} * \mathrm{rot}_{180^\circ}\left\{w^{l+1}_{m,n}\right\} f'\left(x^{l}_{i',j'}\right) &\quad (22)\end{aligned}$$

Pooling

- Max-pooling: the error is just assigned to where it comes from the winning unit
- Avg.-pooling : error is multiplied by m=NxN and assigned to the whole pooling block(all units get this same vlue)



Max-pooling



Avg-pooling