Convolutional Neural Networks

SRC® JUMP

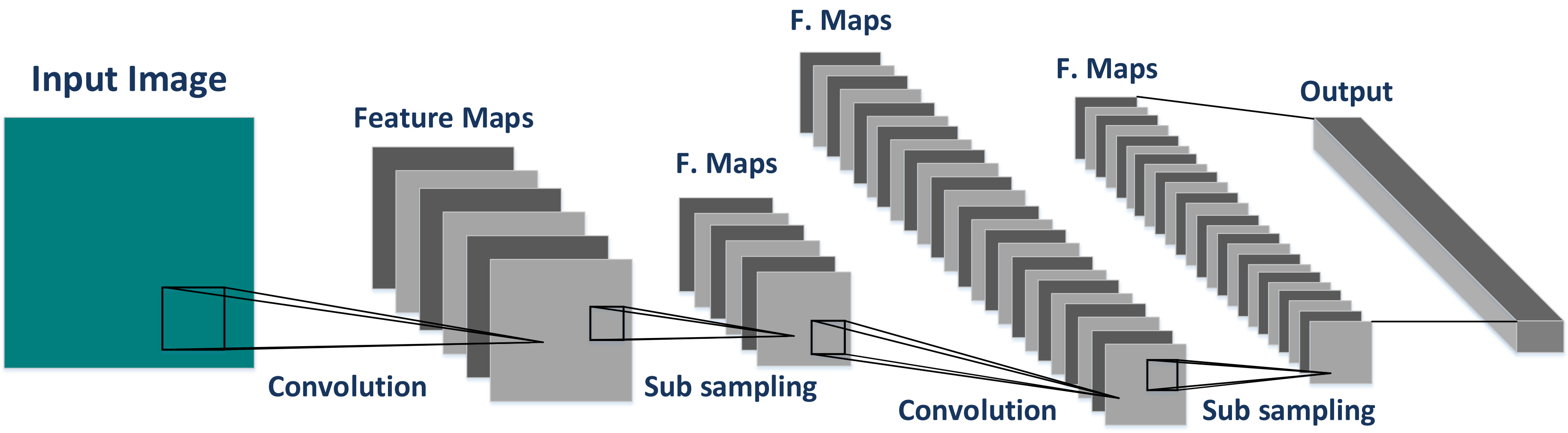
C-BRIC

Convolutional Neural Networks

- Convolutional Neural Networks (CNNs) are a class of deep, feed-forward artificial neural networks
- Used for image/video classification by feature extraction
- Inspired by biological processes in that the connectivity pattern between neurons resembles the organization of the animal visual cortex.
- CNNs use relatively little pre-processing compared to other image classification algorithms
- State of the art powerful deep neural networks are CNNs Ex: ResNet, VGG, AlexNet

Structure

- CNNs consist of set of convolutional layers followed by subsampling layers. A set of fully connected layers at the end
- Convolutional layers extract the features of the input images
- The final fully connected layers classify the input images depending upon the extracted features

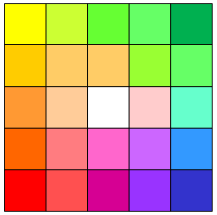


Convolution

Convolution $(I * K)$

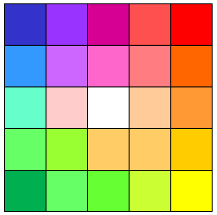
Image, I

$*$



Kernel, K

Convolution

$(I * K) =$ $\otimes$ 

Toy example (3×3 kernel)

$$1.0 + 1.0 + 1.1$$
$$+0.1 + 0.0 + 0.0$$
$$+0.0 + 0.1 + 0.1$$

1	1	1	1	1	1
1	0	0	0	0	1
1	0	0	0	0	1
1	0	0	0	0	1
1	0	0	0	0	1
1	1	1	1	1	1

Image

$\otimes$

0	0	1
1	0	0
0	1	1

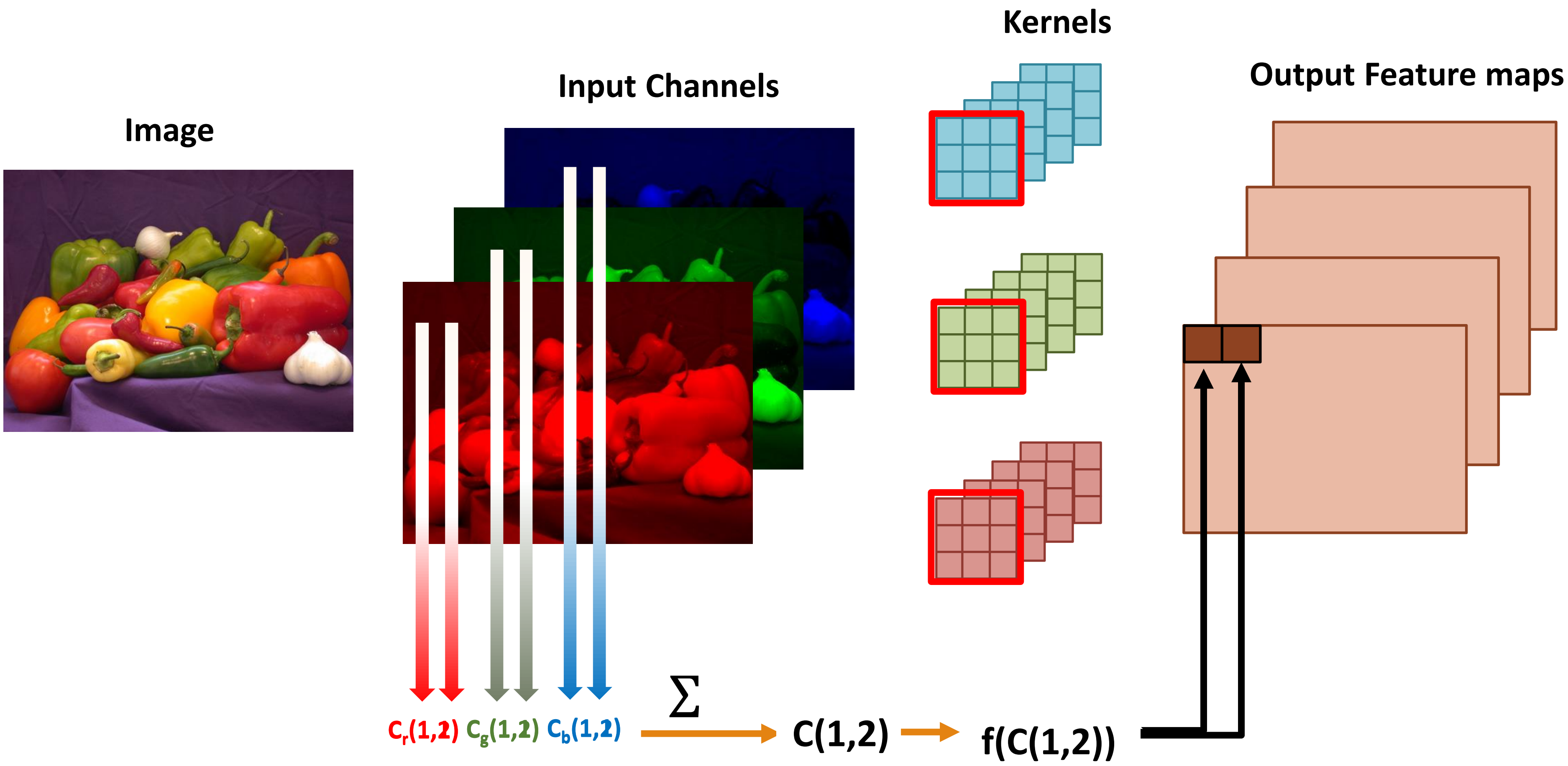
Flipped Kernel

=

2	1	1	2
1	0	0	2
1	0	0	2
3	2	2	3

Feature map

Convolutional Layer Structure



Kernels

Output Feature maps

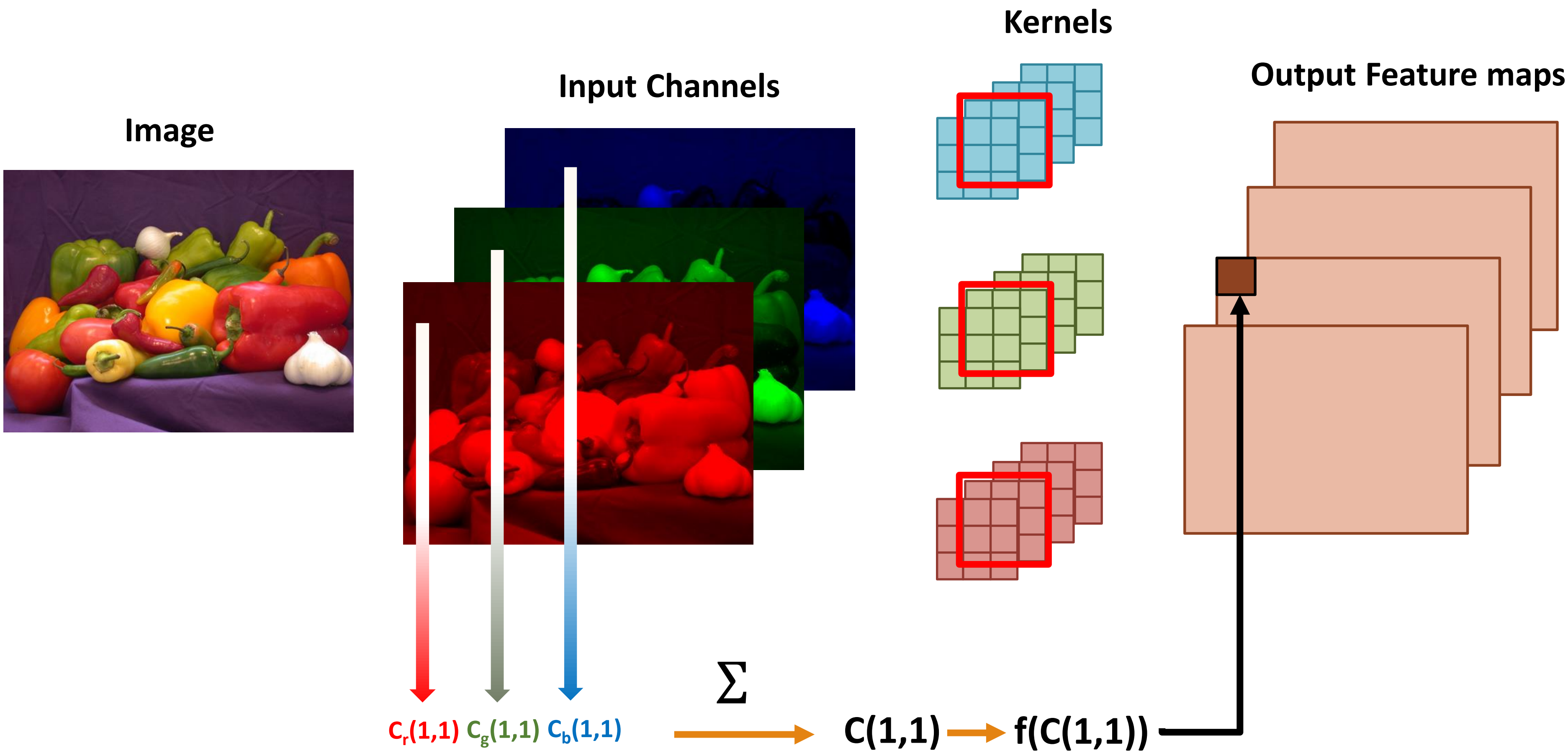
Input Channels

Image

$C_r(1,2)$ $C_g(1,2)$ $C_b(1,2)$ $\xrightarrow{\Sigma}$ $C(1,2) \rightarrow f(C(1,2))$

f is the activation function. Typical functions include Sigmoid, ReLU, tanh

Convolutional Layer Structure

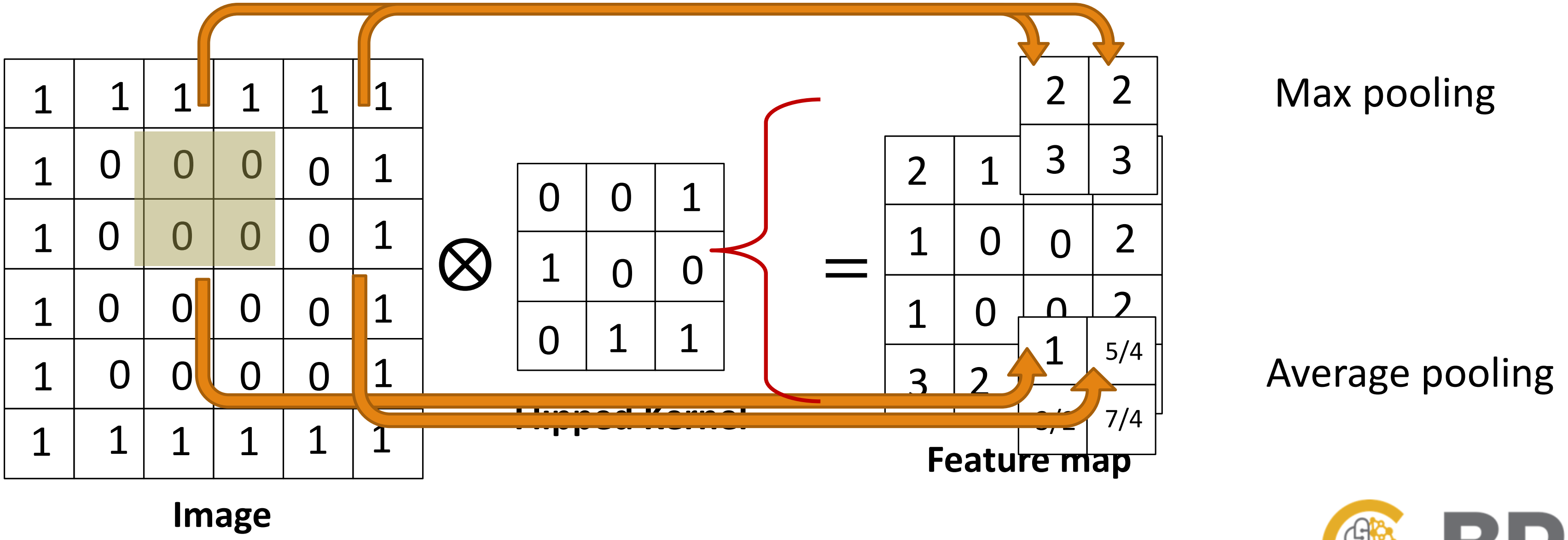


Subsampling (pooling)

Reduce the size of the output feature maps. Two typical pooling types

- Max pooling – Take the maximum out of a set of selected outputs
- Average pooling – Take the average of a set of selected outputs

Toy example (2 × scaling)



1	1	1	1	1	1
1	0	0	0	0	1
1	0	0	0	0	1
1	0	0	0	0	1
1	0	0	0	0	1
1	1	1	1	1	1

Image

⊗

0	0	1
1	0	0
0	1	1

Flipped Kernel

=

2	1		
1	0	0	2
1	0	0	2
3	2		

Feature map

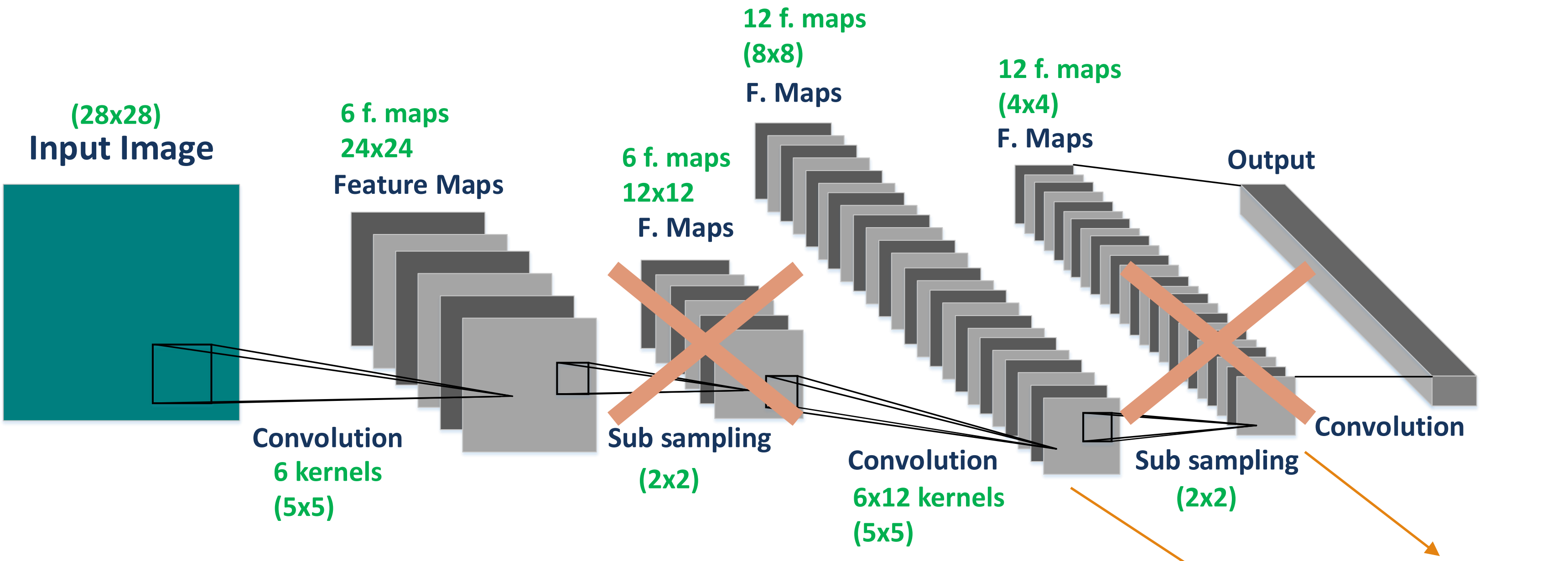
2	2
3	3

Max pooling

1	5/4
3/2	7/4

Average pooling

Subsampling (pooling)



Feature vector size = 20x20x12=4800

Feature vector size = 4x4x12=192

Subsampling reduced the size of the final feature vector from 4800 to 192

Subsampling (pooling)

Why subsampling?

- Reduce variance
- Reduces computational complexity (reduced dimensions)
- Extract features from neighborhood

Max pooling **vs** **Average pooling**

Max pooling

- Preserves more variance
- Some information might be lost

Average pooling

- Features may get smoothened (less variance)
- Has contribution from all the pixels in the feature maps

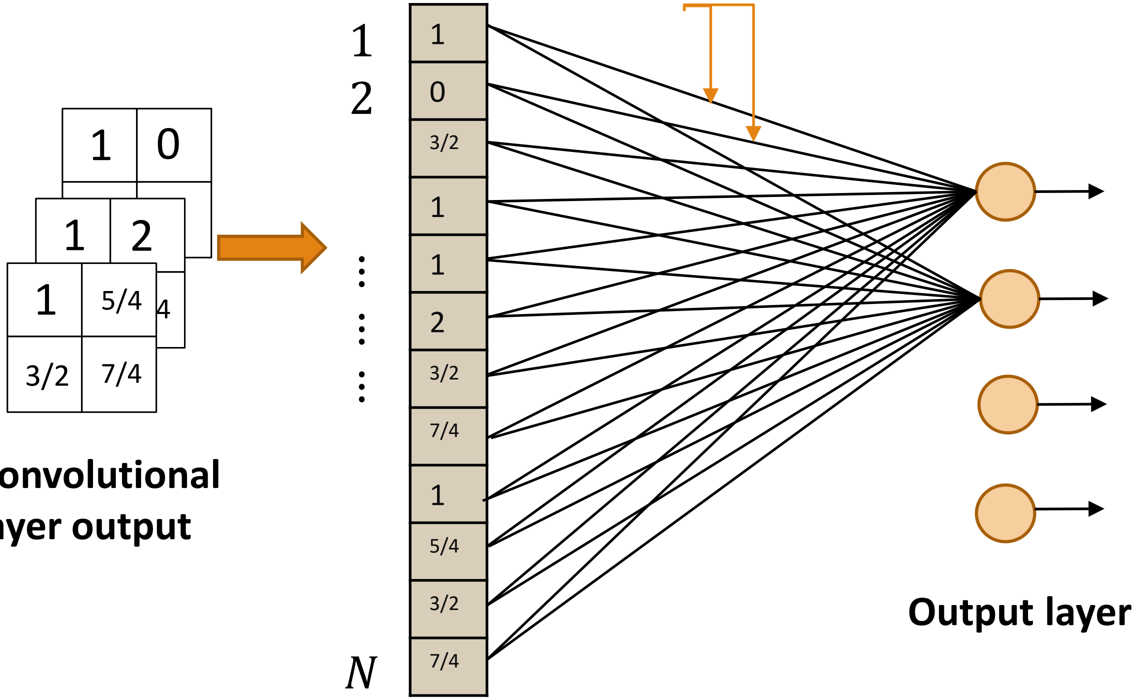
Note: The choice of subsampling method depends on the applications

Fully connected layers (Classifier)

- Fully connected layers at the end of a typical CNN will classify the inputs depending upon the features extracted by the convolutional layers

Weights

$\theta_{i,j}$ – weight between i**th pre neuron and jth post neuron**



Convolutional layer output

Output layer

$$z_j = \sum_{i=1}^{N} a_i \theta_{i,j}$$

Training

- The weights in the convolutional kernels must be changed according to the input training data set
- Typical training method is the gradient descent based backward propagation
- Learning rule

$$\boxed{\theta = \theta + \Delta\theta\,, \qquad \Delta\theta = -\alpha\frac{\partial J(\theta)}{\partial\theta}}$$

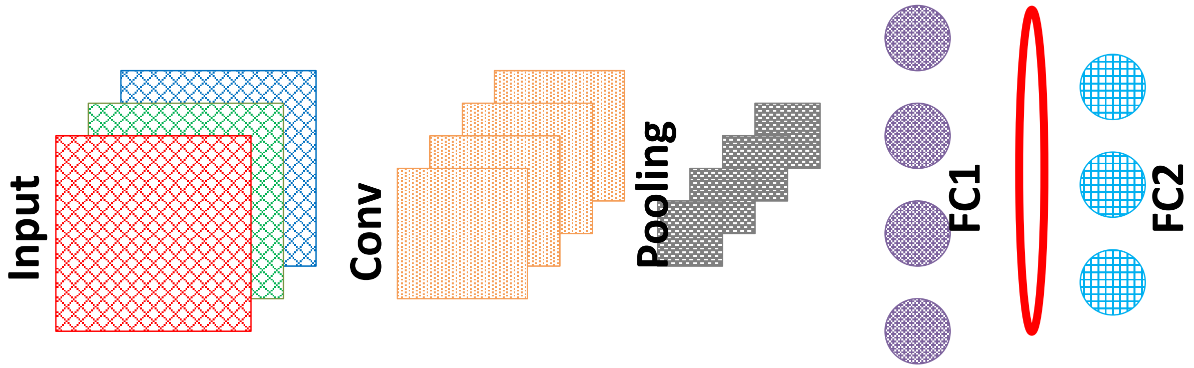
$J(\theta)$ is the error/cost function at the output and α is the learning rate. The error function can be the Euclidean distance between the expected and the actual output

$$\boxed{J = \frac{1}{2}||h - y||^2}$$

y is the expected output, and h is the actual output (hypothesis)

Training - Example

Consider a CNN with two final fully connected (FC) layers, one convolutional layer and an average pooling layer. The activation function is a sigmoid $\left(g(z) = \frac{1}{1+e^{-z}}\right)$



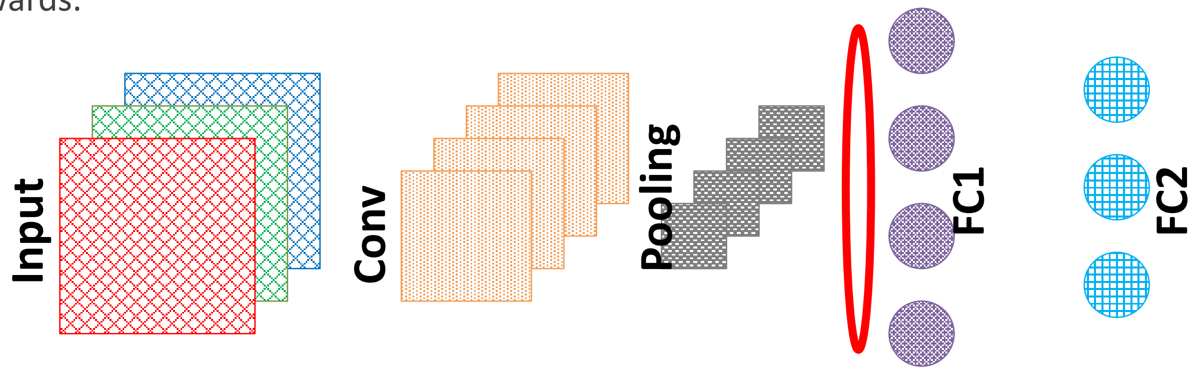
Updating the weights of the final fully connected layers can be done as follows

$$\boxed{\frac{\partial J}{\partial \theta_{i,j}^{[FC2]}} = \frac{\partial J}{\partial h_j}\frac{\partial h_j}{\partial z_j}\frac{\partial z_j}{\partial \theta_{i,j}^{[FC2]}} = (h_j - y_j)h_j(1-h_j)a_i^{[FC1]} = a_i^{[FC1]}\delta_j^{[FC2]}}$$

Here $\theta_{i,j}^{[FC2]}$is the weight between the output of the pre layer ($a_i^{[FC1]}$) and the output of the final FC layer $\left(h_j = g(z_j)\right)$. z_j is the weighted summation that is fed to the output neurons ($z_j = \sum_i a_i^{[FC1]}\theta_{i,j}^{[FC2]}$)

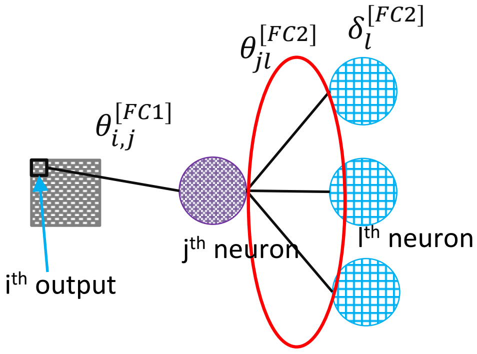
Training - Example

Updating the weights of the second FC layer is slightly different since the error at the output must be propagated backwards.



$$\frac{\partial J}{\partial a_j^{[FC1]}} = \sum_{l\in L}\left(\frac{\partial J}{\partial a_l^{[FC2]}}\frac{\partial a_l^{[FC2]}}{\partial z_l^{[FC2]}}\frac{\partial z_l^{[FC2]}}{\partial a_j^{[FC1]}}\right)$$

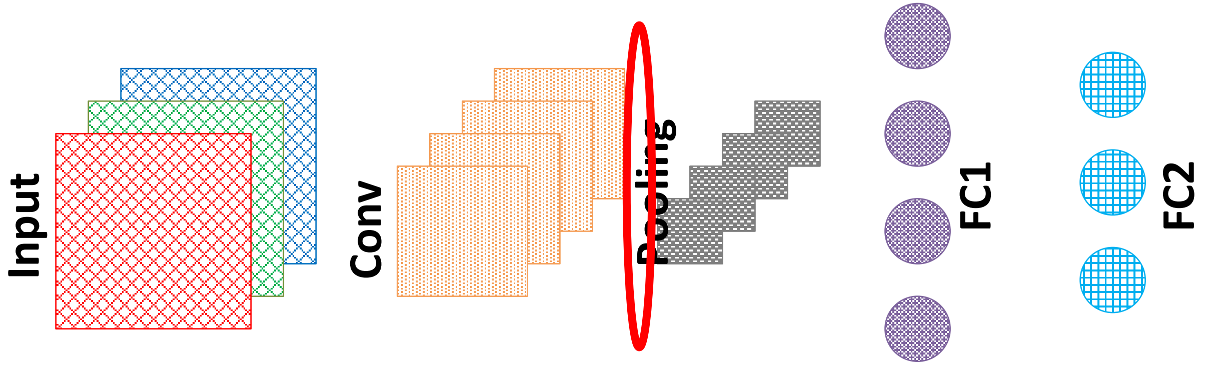
$$\frac{\partial J}{\partial \theta_{i,j}^{[FC1]}} = \frac{\partial J}{\partial a_j^{[FC1]}}\frac{\partial a_j^{[FC1]}}{\partial z_j^{[FC1]}}\frac{\partial z_j^{[FC1]}}{\partial \theta_{ij}^{[FC1]}} = \sum_{l\in L}\delta_l^{[FC2]}\theta_{jl}^{[FC2]}\, a_j^{[FC1]}\left(1 - a_j^{[FC1]}\right) a_i^{[p]}$$
$$= a_i^{[p]}\delta_j^{[FC1]}$$



Here $\theta_{i,j}^{[FC1]}$ is the weight between the output of the pooling layer ($a_i^{[p]}$) and the output of the FC1 layer ($a_j^{[FC1]} = g(z_j^{[FC1]})$). The neuron output $a_j^{[FC1]}$ is connected to all the $l \in L$ neurons in front. $\delta_l^{[FC2]}$ is the $\frac{\partial J}{\partial z_l^{[FC2]}}$ of the l^{th} neuron in front.

Training - Example

For all the hidden layers, the aforementioned $\frac{\partial J}{\partial a_j} = \sum_{l \in L} \delta_l \theta_{jl}$ rule applies. There is no weight update to the pooling layer (since it is only a scaling operation). However, the associated δ values must be calculated to update the conv layer weights



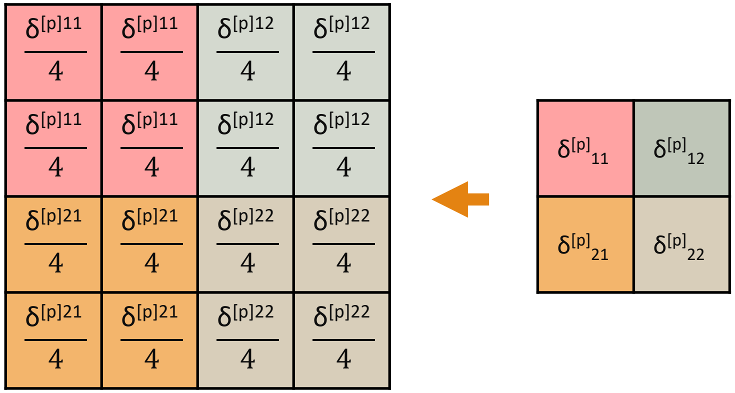
$$\delta_j^{[p]} = \frac{\partial J}{\partial a_j^{[p]}} \frac{\partial a_j^{[p]}}{\partial z_j^{[p]}} = \sum_{l \in L} \delta_l^{[FC1]} \theta_{jl}^{[FC1]}$$

Here $\theta_{jl}^{[FC1]}$is a weight between the output of the pooling layer $\left(a_j^{[p]}\right)$ and the output of the FC1 layer $\left(a_l^{[FC1]} = g\left(z_l^{[FC1]}\right)\right)$. The neuron output $a_j^{[p]}$is connected to all the $l \in L$ neurons in front. Here the $a_j^{[p]}(1 - a_j^{[p]})$ term is not used since the pooling layer output does not have a sigmoid function associated with it.

Training - Example

In order to calculate the δ values associated with the conv. Layer $\left(\delta^{[c]}\right)$, the pooling layer's δ values $\left(\delta^{[p]}\right)$ can be used as follows.

Average pooling

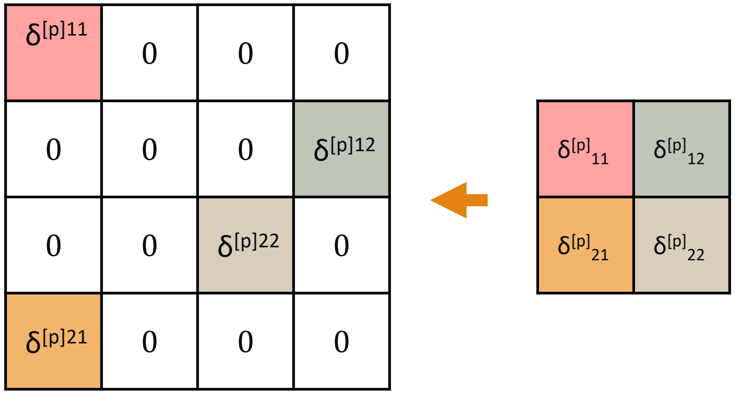


$\frac{\delta^{[p]11}}{4}$	$\frac{\delta^{[p]11}}{4}$	$\frac{\delta^{[p]12}}{4}$	$\frac{\delta^{[p]12}}{4}$
$\frac{\delta^{[p]11}}{4}$	$\frac{\delta^{[p]11}}{4}$	$\frac{\delta^{[p]12}}{4}$	$\frac{\delta^{[p]12}}{4}$
$\frac{\delta^{[p]21}}{4}$	$\frac{\delta^{[p]21}}{4}$	$\frac{\delta^{[p]22}}{4}$	$\frac{\delta^{[p]22}}{4}$
$\frac{\delta^{[p]21}}{4}$	$\frac{\delta^{[p]21}}{4}$	$\frac{\delta^{[p]22}}{4}$	$\frac{\delta^{[p]22}}{4}$

$\delta^{[p]}_{11}$	$\delta^{[p]}_{12}$
$\delta^{[p]}_{21}$	$\delta^{[p]}_{22}$

$$\delta^{[c]}_{i,j} = \frac{\delta^{[p]}_{m,n}}{N \times N} a^{[c]}_{i,j}\left(1 - a^{[c]}_{i,j}\right)$$

Max pooling



$\delta^{[p]11}$	0	0	0
0	0	0	$\delta^{[p]12}$
0	0	$\delta^{[p]22}$	0
$\delta^{[p]21}$	0	0	0

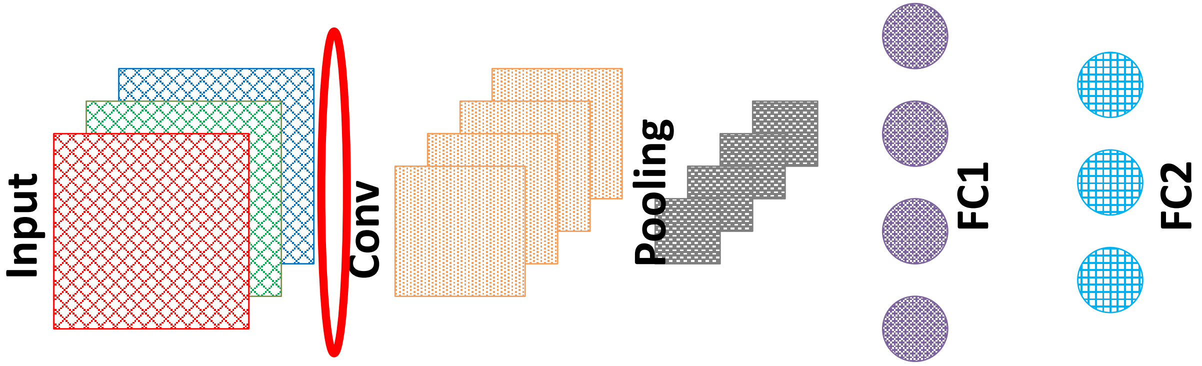
$\delta^{[p]}_{11}$	$\delta^{[p]}_{12}$
$\delta^{[p]}_{21}$	$\delta^{[p]}_{22}$

$$\delta^{[c]}_{i,j} = \begin{cases} \delta^{[p]}_{m,n} a^{[c]}_{i,j}\left(1 - a^{[c]}_{i,j}\right) & \text{if it is the max val} \\ 0 & \text{if it is not the max val} \end{cases}$$

The (i,j) element in the conv layer output will be mapped to the (m,n) element in the pooling layer output. N is the scaling factor during pooling. In above example, $N = 2$. $a^{[c]}_{i,j}$ is the output of the conv layer.

Training - Example

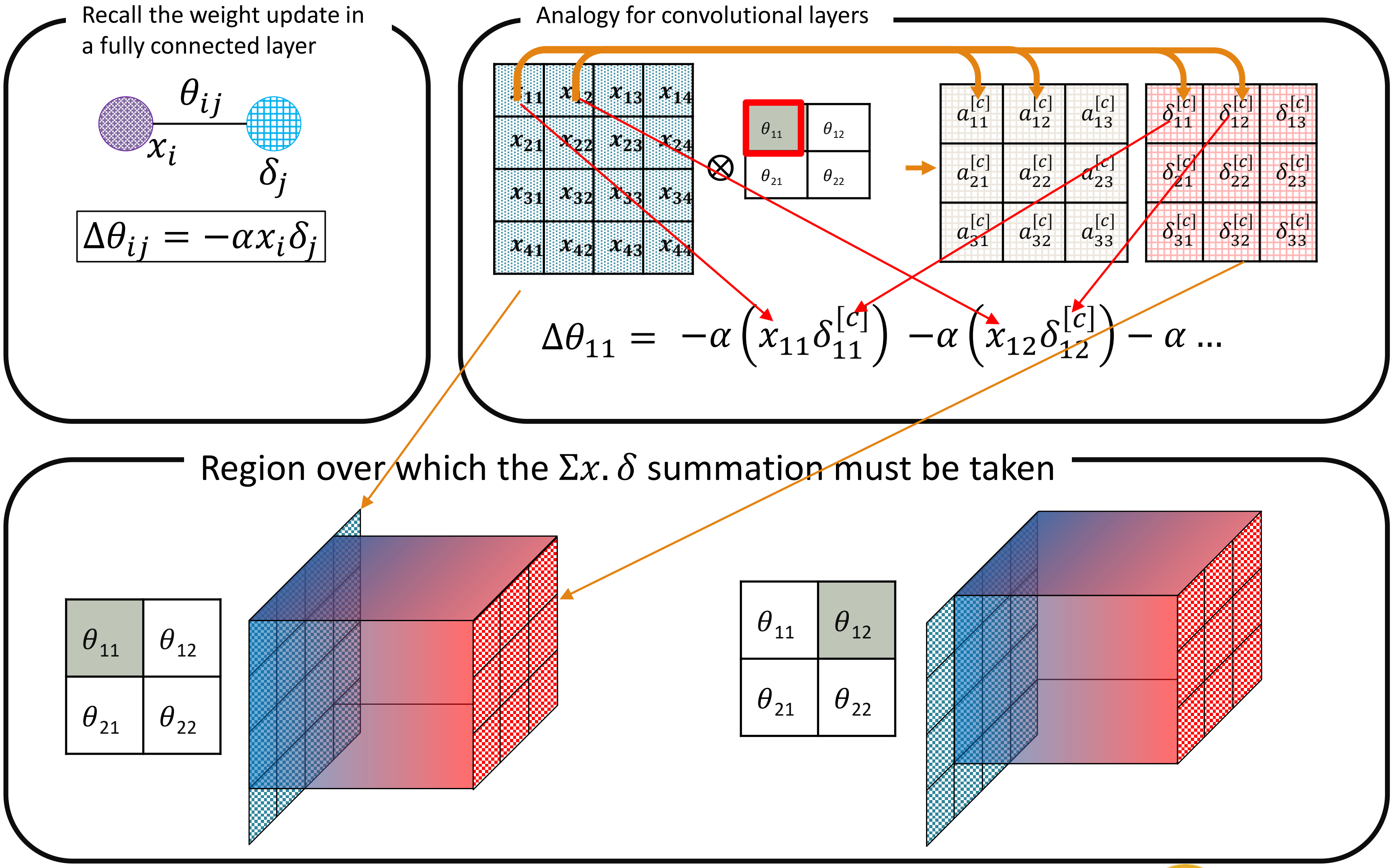
Once the δ values are calculated, the weights between the input and the conv layer can be calculated as follows.



$$\frac{\partial J}{\partial \theta_{i,j}^{[c]}} = \sum x_i \delta_j^{[c]}$$

$\theta_{i,j}^{[c]}$ is the weight between the input layer output x_i and convolution layer output $a_j^{[c]}$. Unlike a weight in a fully connected layer, a weight in a convolutional layer is connected to multiple inputs and outputs. Therefore, the summation over the entire space must be taken.

Training - Example



Recall the weight update in a fully connected layer

θ_{ij}
x_i
δ_j

$$\boxed{\Delta\theta_{ij} = -\alpha x_i \delta_j}$$

Analogy for convolutional layers

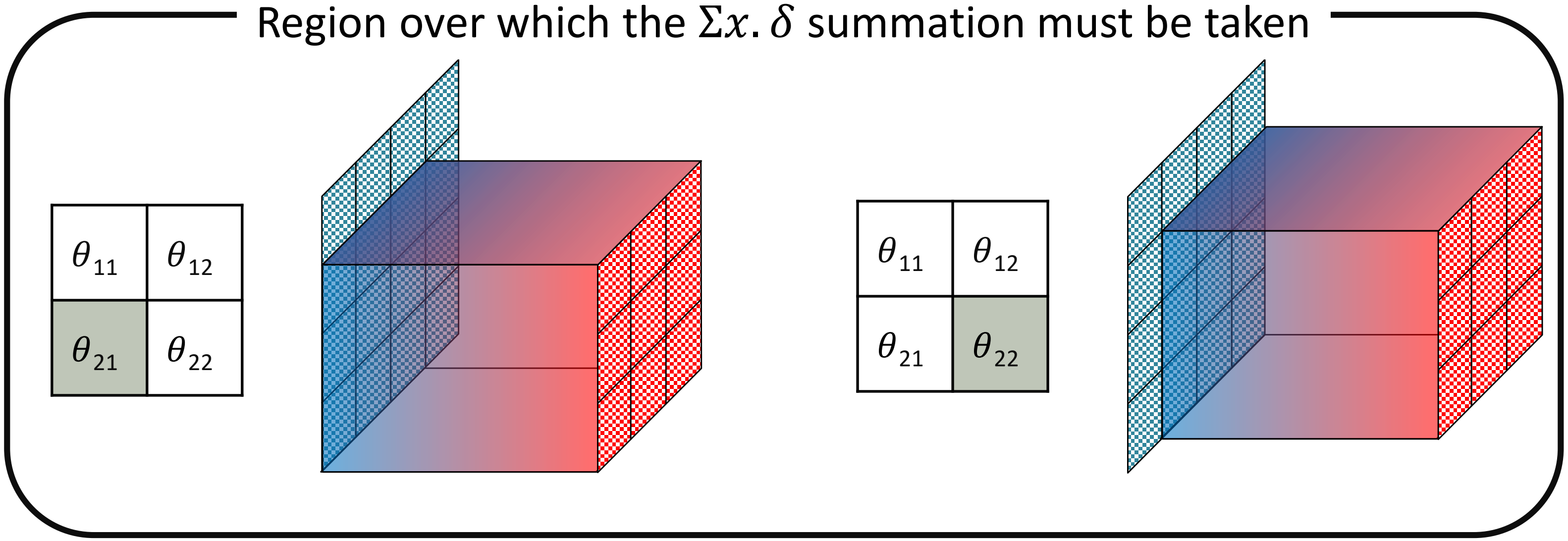
x_{11} x_{12} x_{13} x_{14}
x_{21} x_{22} x_{23} x_{24}
x_{31} x_{32} x_{33} x_{34}
x_{41} x_{42} x_{43} x_{44}
$\otimes$
θ_{11} θ_{12}
θ_{21} θ_{22}
$a_{11}^{[c]}$ $a_{12}^{[c]}$ $a_{13}^{[c]}$
$a_{21}^{[c]}$ $a_{22}^{[c]}$ $a_{23}^{[c]}$
$a_{31}^{[c]}$ $a_{32}^{[c]}$ $a_{33}^{[c]}$
$\delta_{11}^{[c]}$ $\delta_{12}^{[c]}$ $\delta_{13}^{[c]}$
$\delta_{21}^{[c]}$ $\delta_{22}^{[c]}$ $\delta_{23}^{[c]}$
$\delta_{31}^{[c]}$ $\delta_{32}^{[c]}$ $\delta_{33}^{[c]}$

$$\Delta\theta_{11} = -\alpha\left(x_{11}\delta_{11}^{[c]}\right) - \alpha\left(x_{12}\delta_{12}^{[c]}\right) - \alpha \ldots$$

Region over which the $\Sigma x.\delta$ summation must be taken

θ_{11} θ_{12}
θ_{21} θ_{22}
θ_{11} θ_{12}
θ_{21} θ_{22}

Training - Example

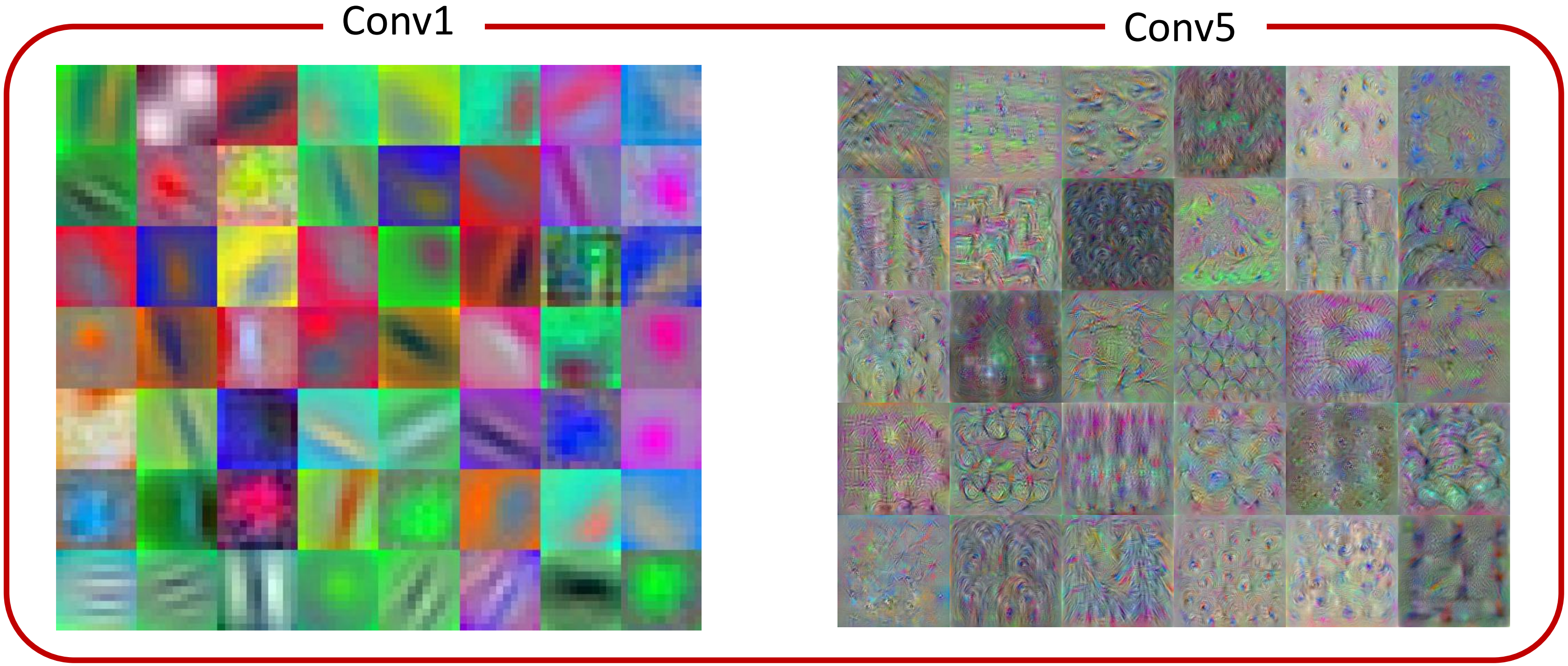


$$-\alpha\left(\begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \\ x_{41} & x_{42} & x_{43} & x_{44} \end{bmatrix} \otimes \begin{bmatrix} \delta^{[c]}_{11} & \delta^{[c]}_{12} & \delta^{[c]}_{13} \\ \delta^{[c]}_{21} & \delta^{[c]}_{22} & \delta^{[c]}_{23} \\ \delta^{[c]}_{31} & \delta^{[c]}_{32} & \delta^{[c]}_{33} \end{bmatrix}\right) = \begin{bmatrix} \Delta\theta_{11} & \Delta\theta_{12} \\ \Delta\theta_{21} & \Delta\theta_{22} \end{bmatrix}$$

$$\boxed{\Delta\theta = -\alpha(x\otimes\delta\,)}$$

Trained networks

- A trained network is capable to classifying certain inputs which were not available during the training
- The convolutional kernels closer to the images will extract basic features whereas, deeper kernels will extract more subtle features of inputs



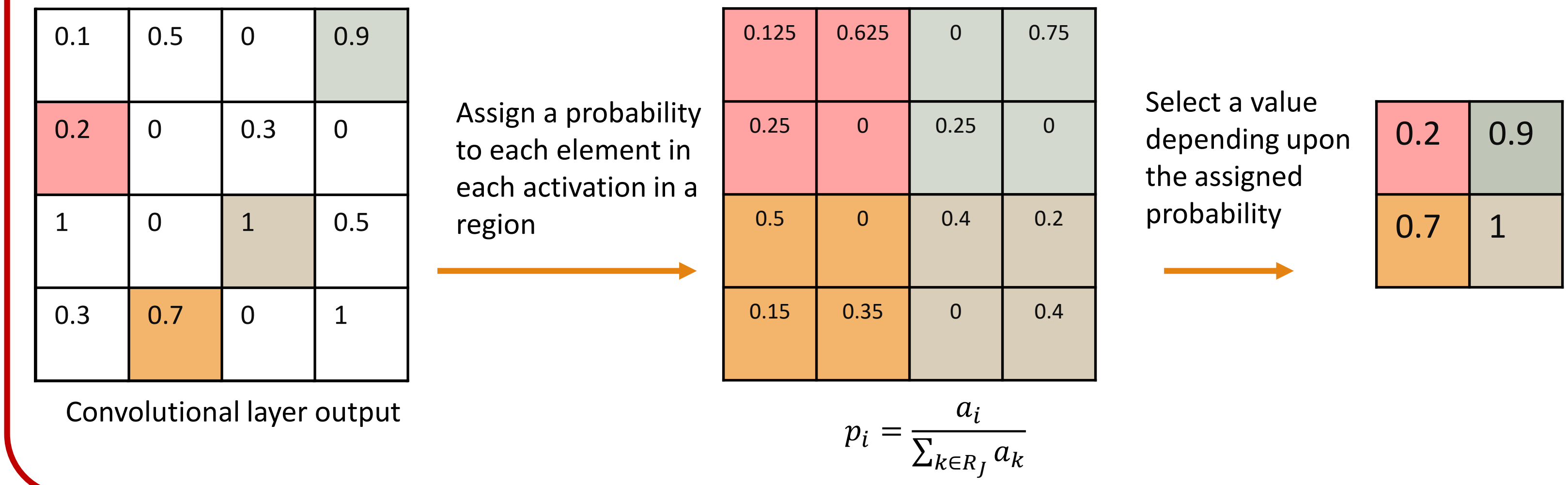
Features activated by the 1st and 5th convolutional layer kernels in AlexNet

Trained networks

- Convolutional neural networks also face overfitting problem.
- The generalization techniques explained previously are applicable here as well.
- Apart from those methods, stochastic pooling is another generalization technique used to avoid overfitting specifically in CNNs

Stochastic Pooling

Involves stochastically activating certain outputs in the pooling layer



0.1	0.5	0	0.9
0.2	0	0.3	0
1	0	1	0.5
0.3	0.7	0	1

Convolutional layer output

Assign a probability to each element in each activation in a region

0.125	0.625	0	0.75
0.25	0	0.25	0
0.5	0	0.4	0.2
0.15	0.35	0	0.4

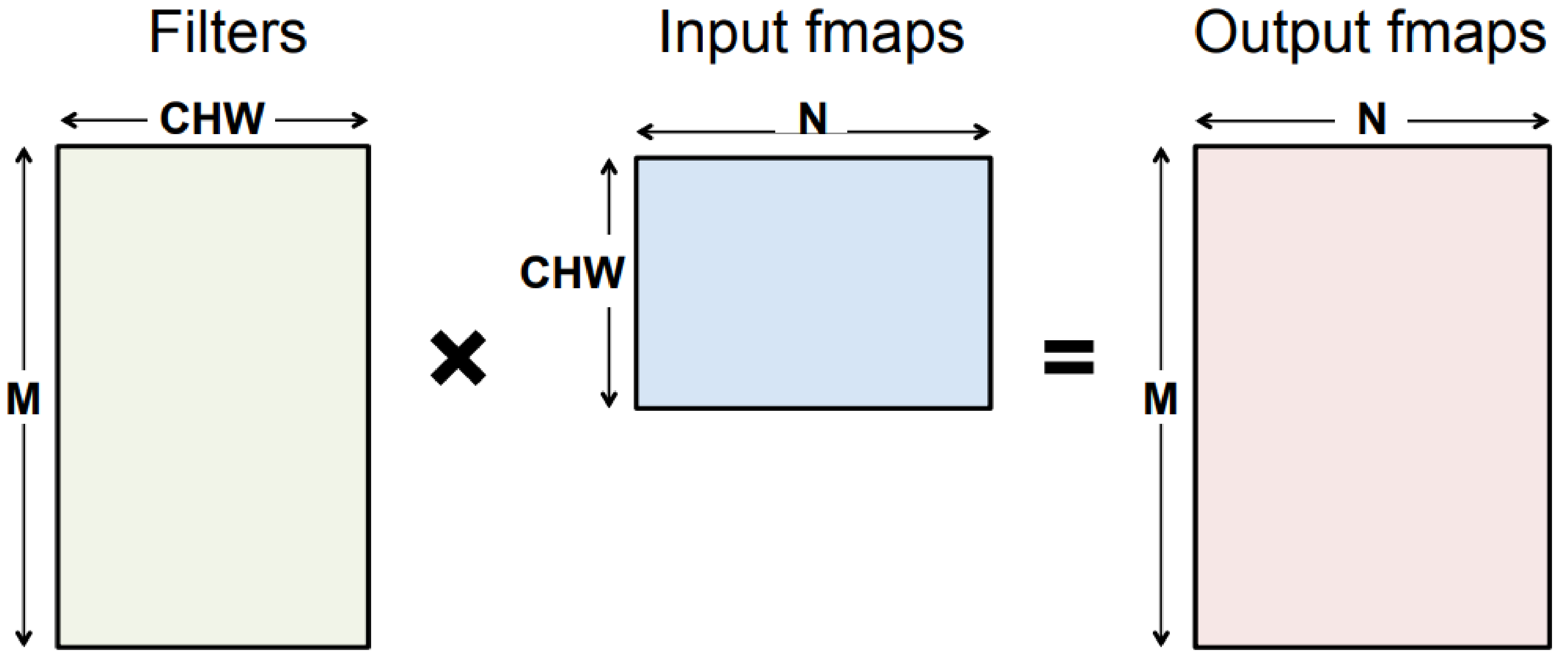
$$p_i = \frac{a_i}{\sum_{k \in R_J} a_k}$$

Select a value depending upon the assigned probability

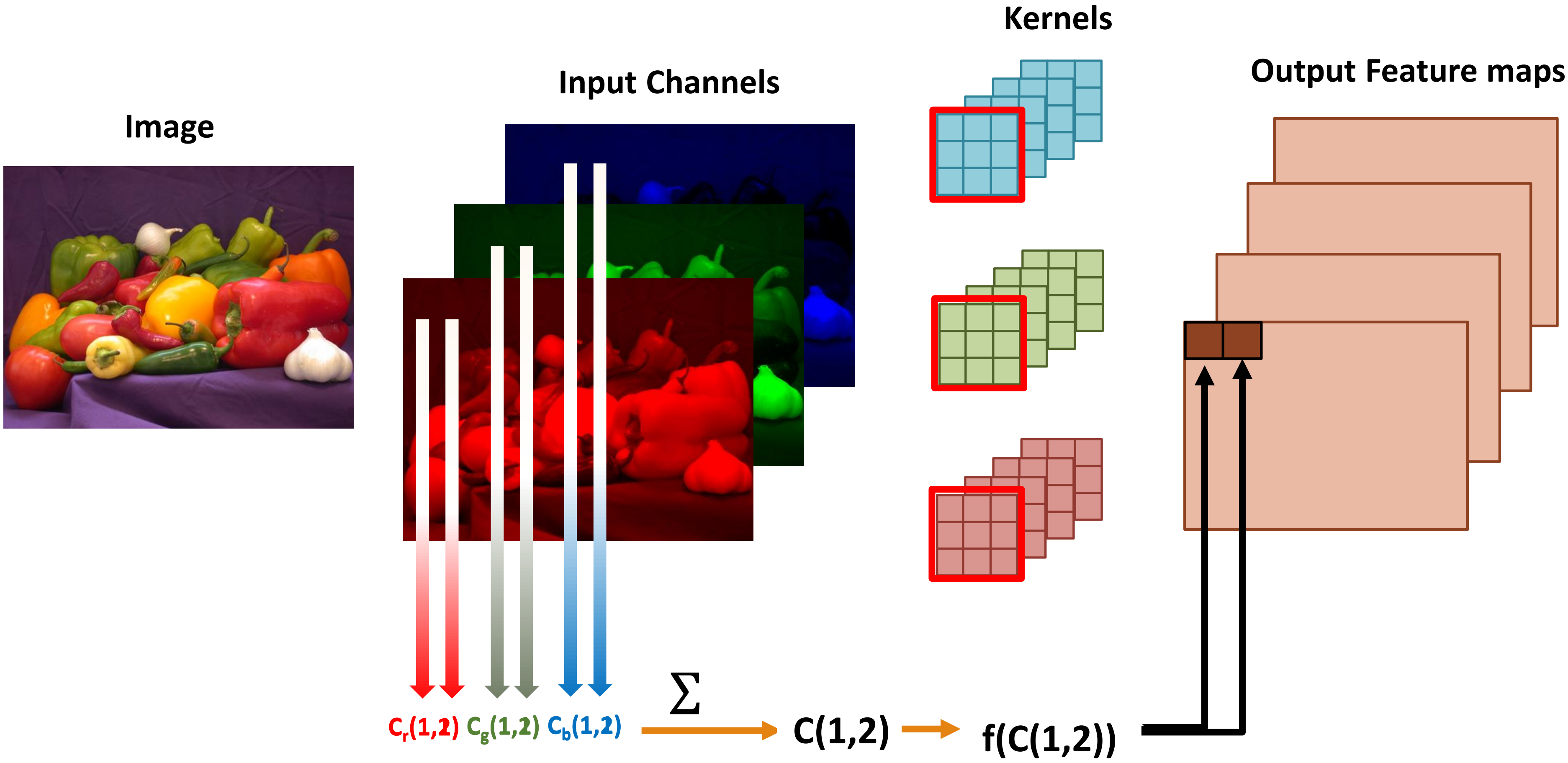
0.2	0.9
0.7	1

GPU Implementation: Fully Connected Layer Matrix-Matrix Operation

- Batching (N) turns operation into a Matrix-Matrix multiply



Convolutional Layer Structure - Revisited



Image

Input Channels

Kernels

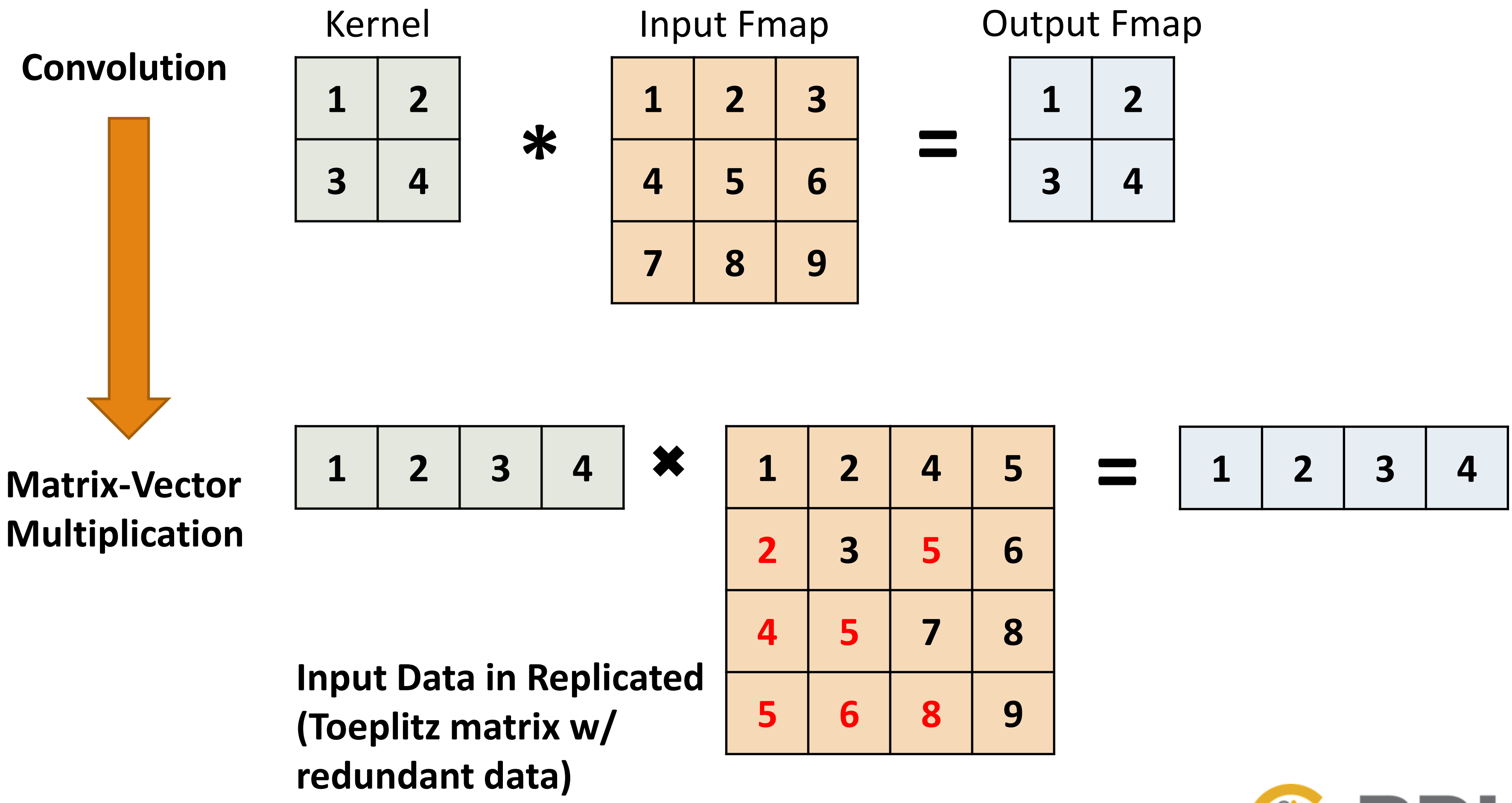
Output Feature maps

$C_r(1,2)$ $C_g(1,2)$ $C_b(1,2)$ $\xrightarrow{\Sigma}$ C(1,2) → f(C(1,2))

f is the activation function. Typical functions include Sigmoid, ReLU, tanh

Conv Layer as Matrix-Matrix Operation - 1

Convert to matrix-matrix operation using **Toeplitz Matrix**



Convolution

Kernel

1	2
3	4

*

Input Fmap

1	2	3
4	5	6
7	8	9

=

Output Fmap

1	2
3	4

Matrix-Vector Multiplication

1	2	3	4

✖

1	2	4	5
2	3	5	6
4	5	7	8
5	6	8	9

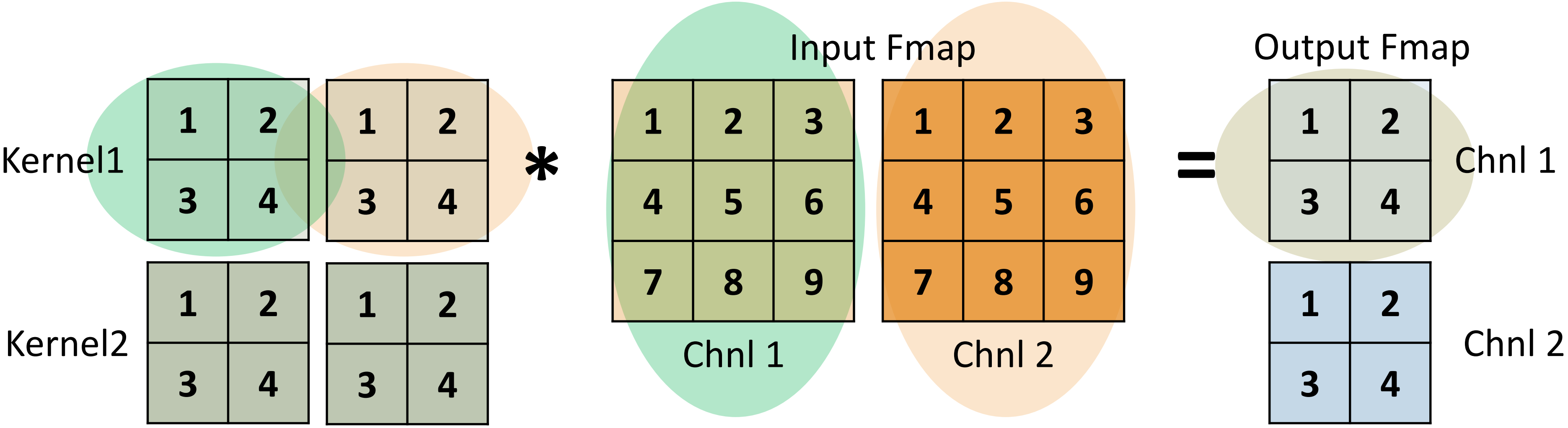
=

1	2	3	4

Input Data in Replicated (Toeplitz matrix w/ redundant data)

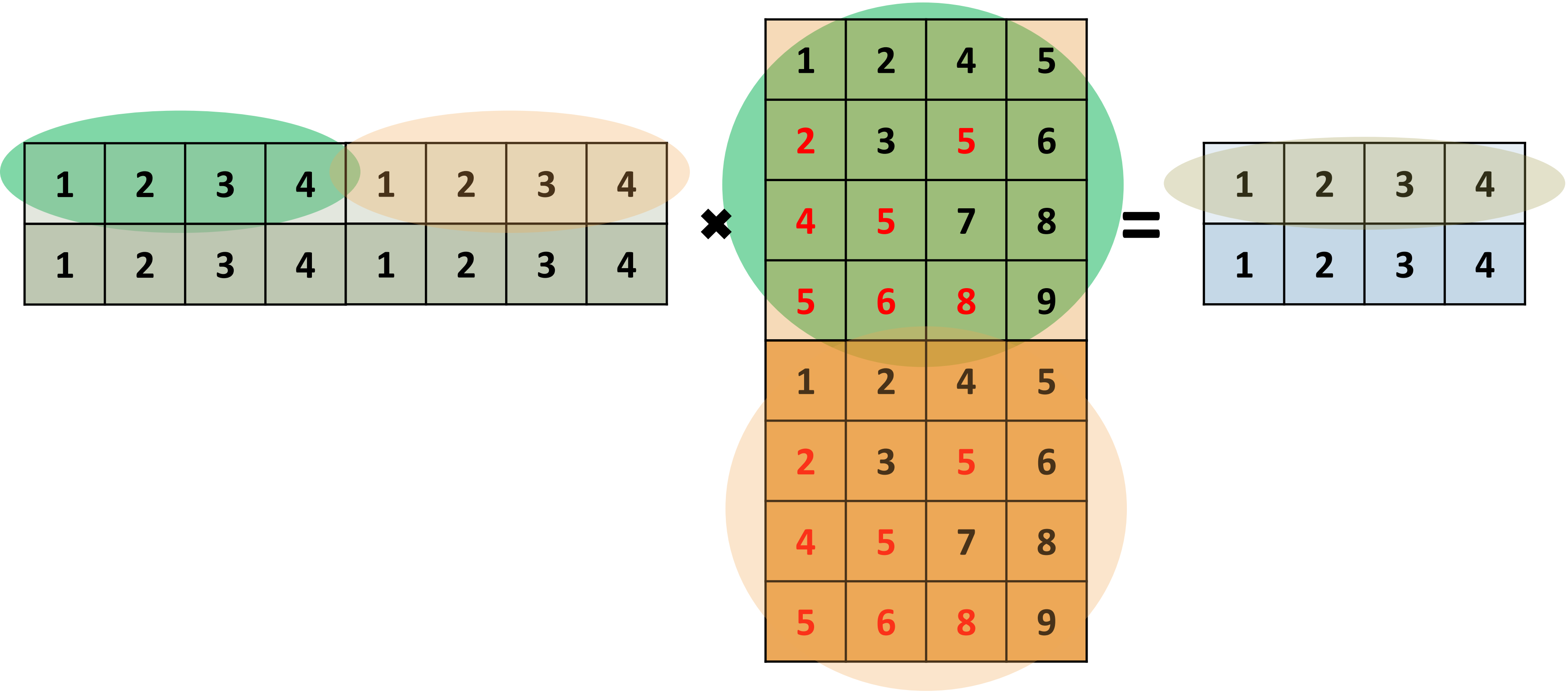
Conv Layer as Matrix-Matrix Operation - 2

Convolution layer with **multiple** kernels and channels



Conv Layer as Matrix-Matrix Operation - 3

Multiple kernels and channels as **Matrix-Matrix operation**



$$\begin{bmatrix} 1 & 2 & 3 & 4 & 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 & 1 & 2 & 3 & 4 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 4 & 5 \\ 2 & 3 & 5 & 6 \\ 4 & 5 & 7 & 8 \\ 5 & 6 & 8 & 9 \\ 1 & 2 & 4 & 5 \\ 2 & 3 & 5 & 6 \\ 4 & 5 & 7 & 8 \\ 5 & 6 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

Input Data in Replicated (Toeplitz matrix w/ redundant data)