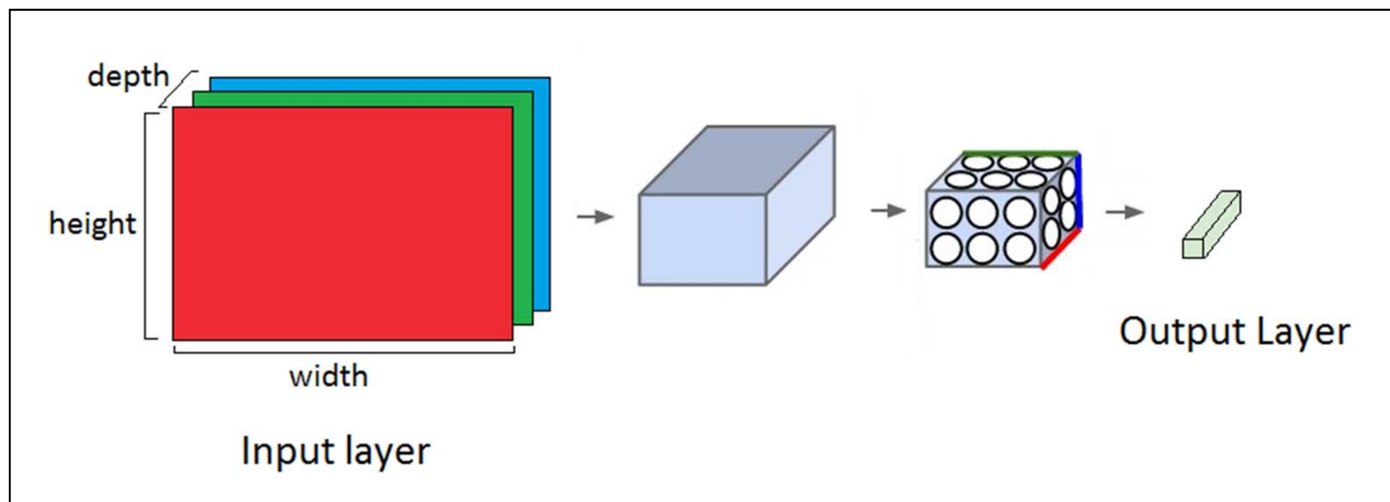
Introduction to Deep Learning

Junzhou Huang, Ph.D.

Department of Computer Science and Engineering

What is CNN?

- Convolutional Neural Network (CNN) is a list of layers that transforms the **image volume** into an **output volume** which will hold the class scores



Example: Input layer holds the image; its width and height is the width and height of the image, and its depth is 3 (RGB channels). Output layer holds the class scores

Basic concepts in CNN

- Explicit assumption in CNN architectures: "**the inputs are images**"
- 3D volumes of neurons: CNN layers have neurons arranged in three dimensions - **width, height and depth**
- Made of neurons that may have: Learnable weights (w) and Biases (b)

- Each neuron receives some inputs, then optionally performs a dot product, add bias, and follows with a non-linearity
- Neurons in a single layer function completely independently, and do not share any connections
- Neurons, Activations, Units – these terms are equivalent to each other

Layers of CNN

- Convolutional Neural Networks have three types of layers:
  - Convolutional layer
  - Pooling layer
  - Fully-connected layer
- Also, sometimes we consider ReLU (Rectified Linear Unit) as a separate layer
- Each of these layers transforms a 3D input volume to a 3D output volume
- We will stack these layers to form a full CNN architecture

Layers of CNN (cont.)

- Hyper-parameters:
  - ReLU does not have, other three have
- Parameters (Model Parameters):
  - Convolutional and Fully Connected layer have, but Pooling and ReLU layer don't have

Convolutional Layer

- Core building block of a CNN architecture
- Consists of a set of learnable **three-dimensional filters**
- Every filter is small spatially (along width and height), but extends through the full depth of the input volume
- Thus, connections are local in space, but always full along the entire depth of input volume
- The spatial extent of this connectivity is called the **filter size** or **receptive field**, which is a hyper-parameter

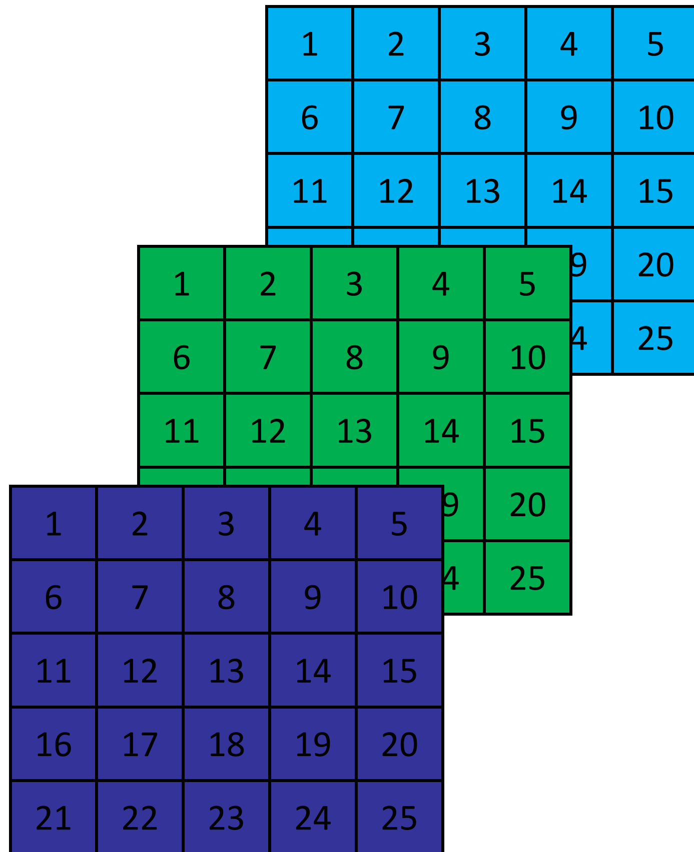
Convolutional Layer (cont.)

- During forward pass:
  - We slide (convolve) each filter across the width and height of the input volume, do element-wise multiplication between the entries of the filter and the input at any position, and finally compute the sum
  - Then, we add the bias terms (b) to it
- Therefore, convolutional layer is just an **image convolution** of the previous layer, where the weights (w) specify the convolution filter

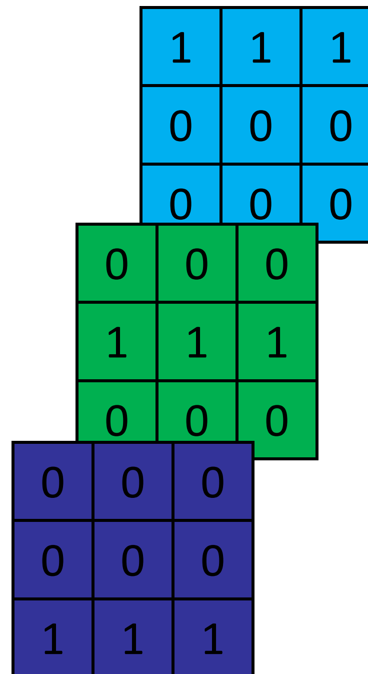
Convolutional Layer (cont.)

- As we slide the filter over the width and height of the input volume, we will produce a two-dimensional activation map
- Thus, applying K different filters, we will get K different two-dimensional activations maps. These activation maps produce our output volume
- Parameter Sharing Scheme:
  - Usually, we use the same filter and same bias variable to get a single activation map
  - This parameter sharing scheme dramatically reduces the numbers of parameters in Convolutional layer
- Many of the times in CNN architecture, a ReLU layer immediately follows a Convolutional layer

Convolutional Layer (cont.)

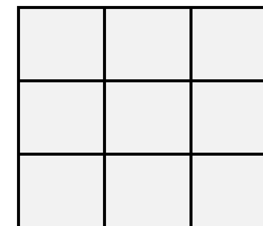


Depth slices of
Input volume



Filter, w0
(3x3x3)

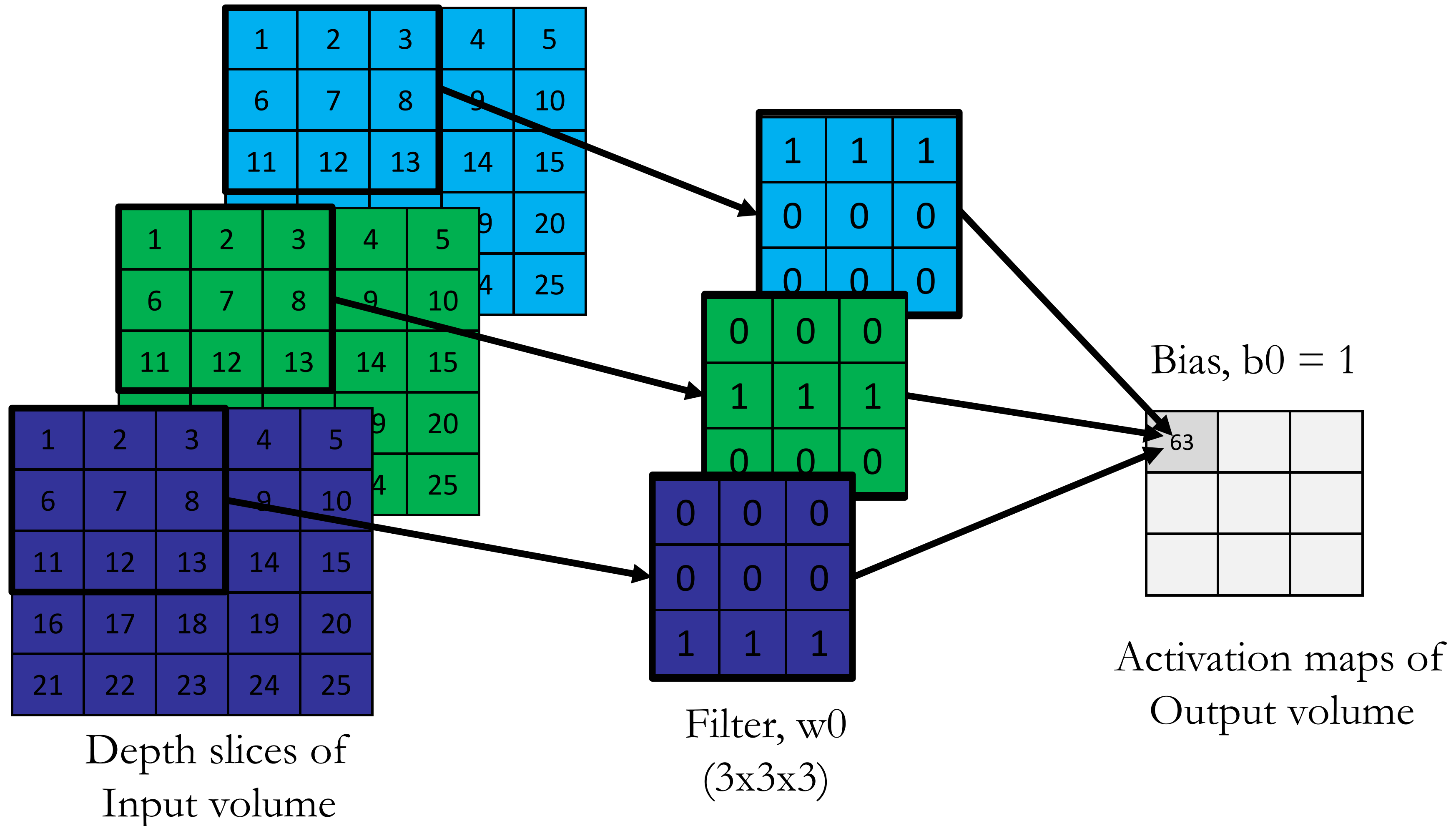
Bias, b0 = 1



Activation maps of
Output volume

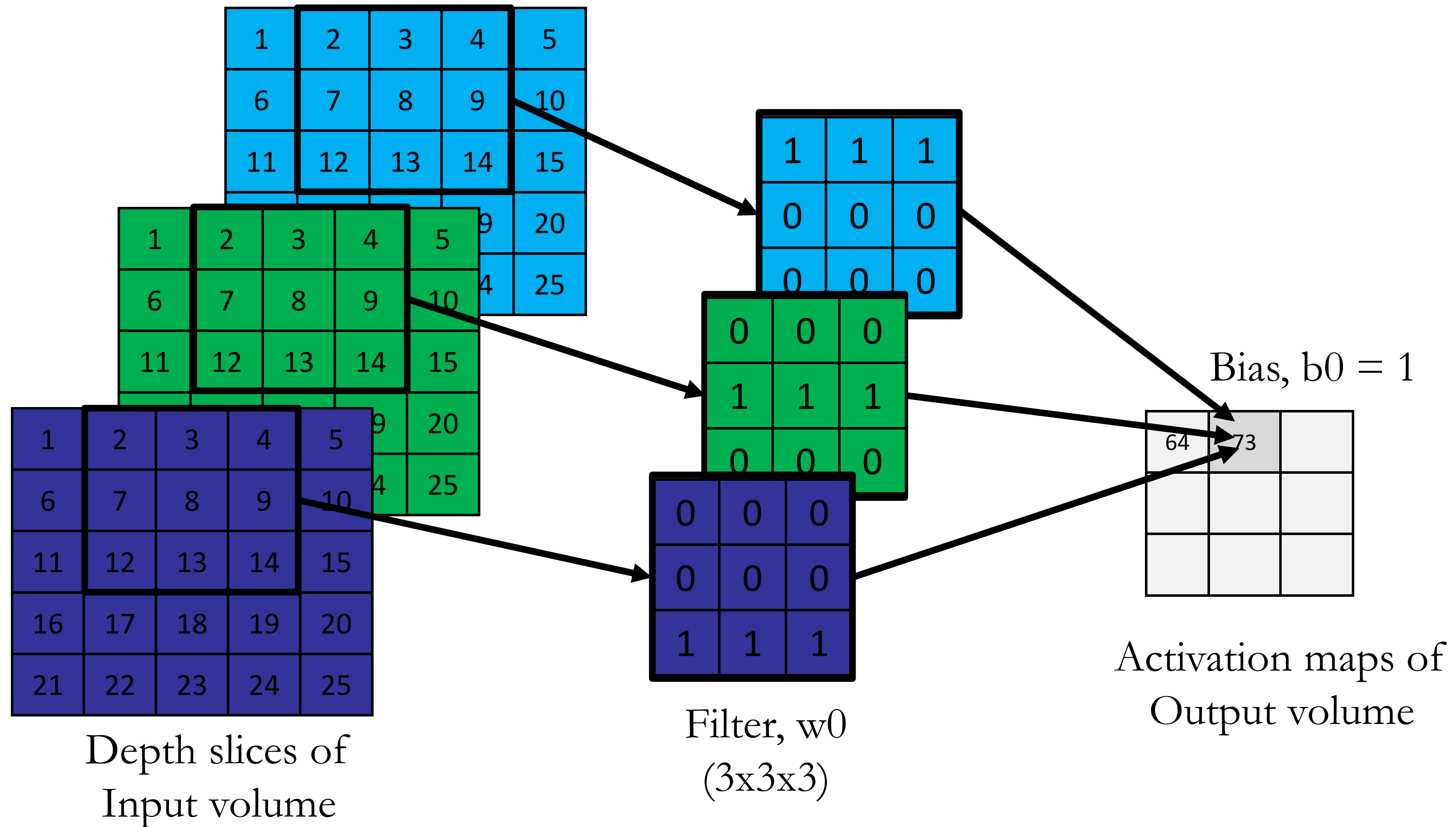
Example: Convolutional Layer with two 3x3x3 filters and Stride 1, Zero-padding 0

Convolutional Layer (cont.)



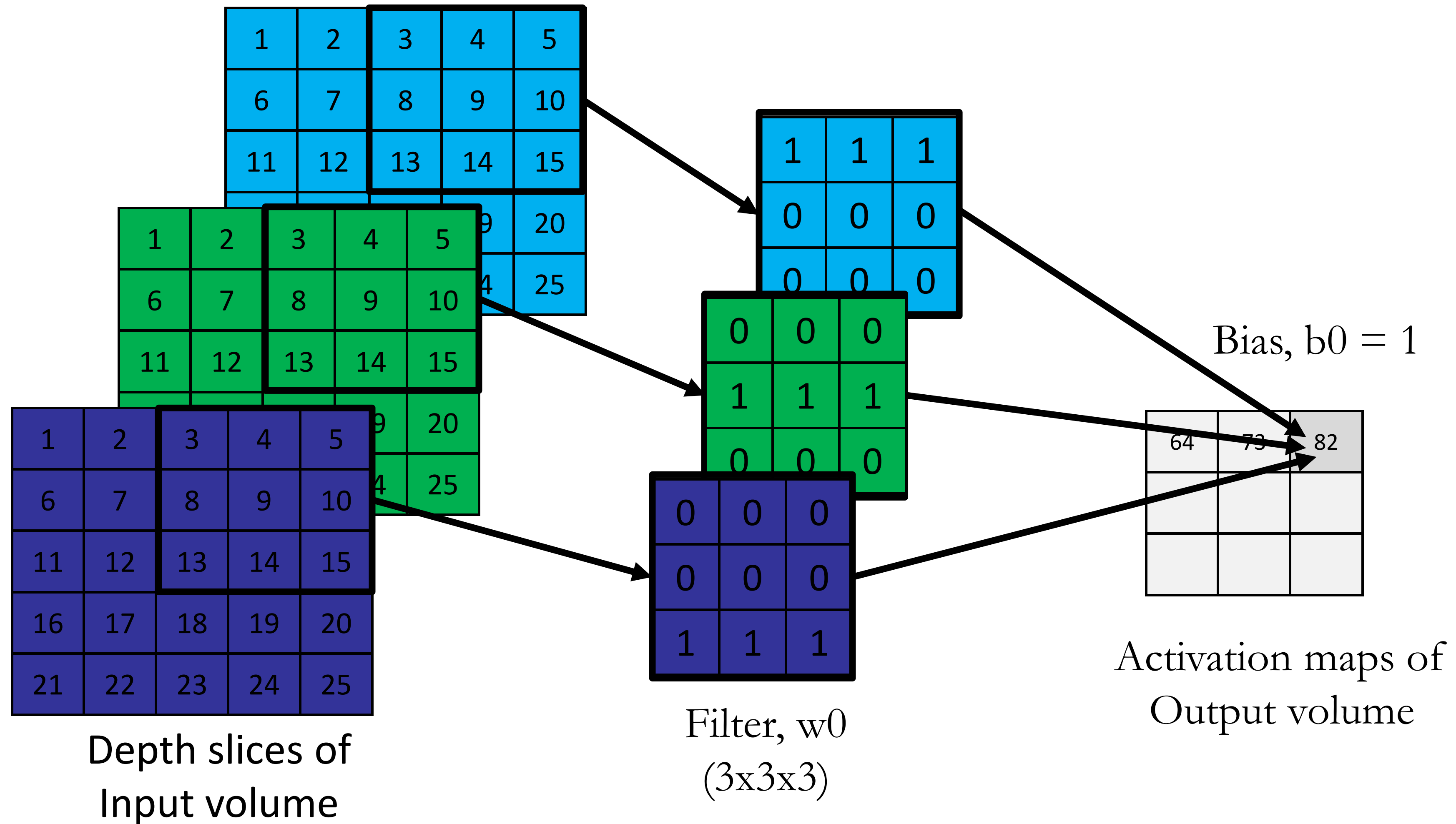
Example: Convolutional Layer with two 3x3x3 filters and Stride 1, Zero-padding 0

Convolutional Layer (cont.)



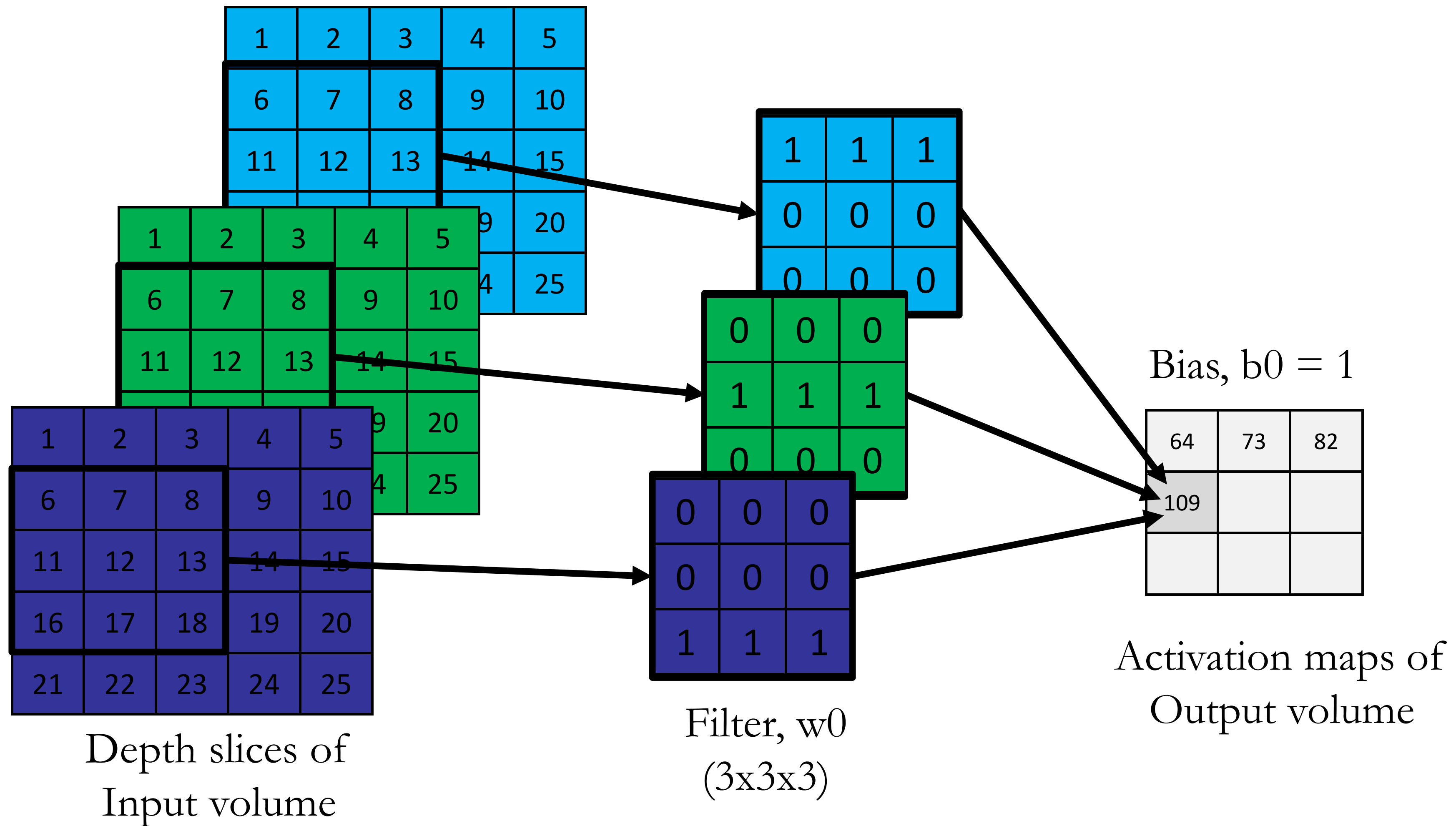
Example: Convolutional Layer with two 3x3x3 filters and Stride 1, Zero-padding 0

Convolutional Layer (cont.)



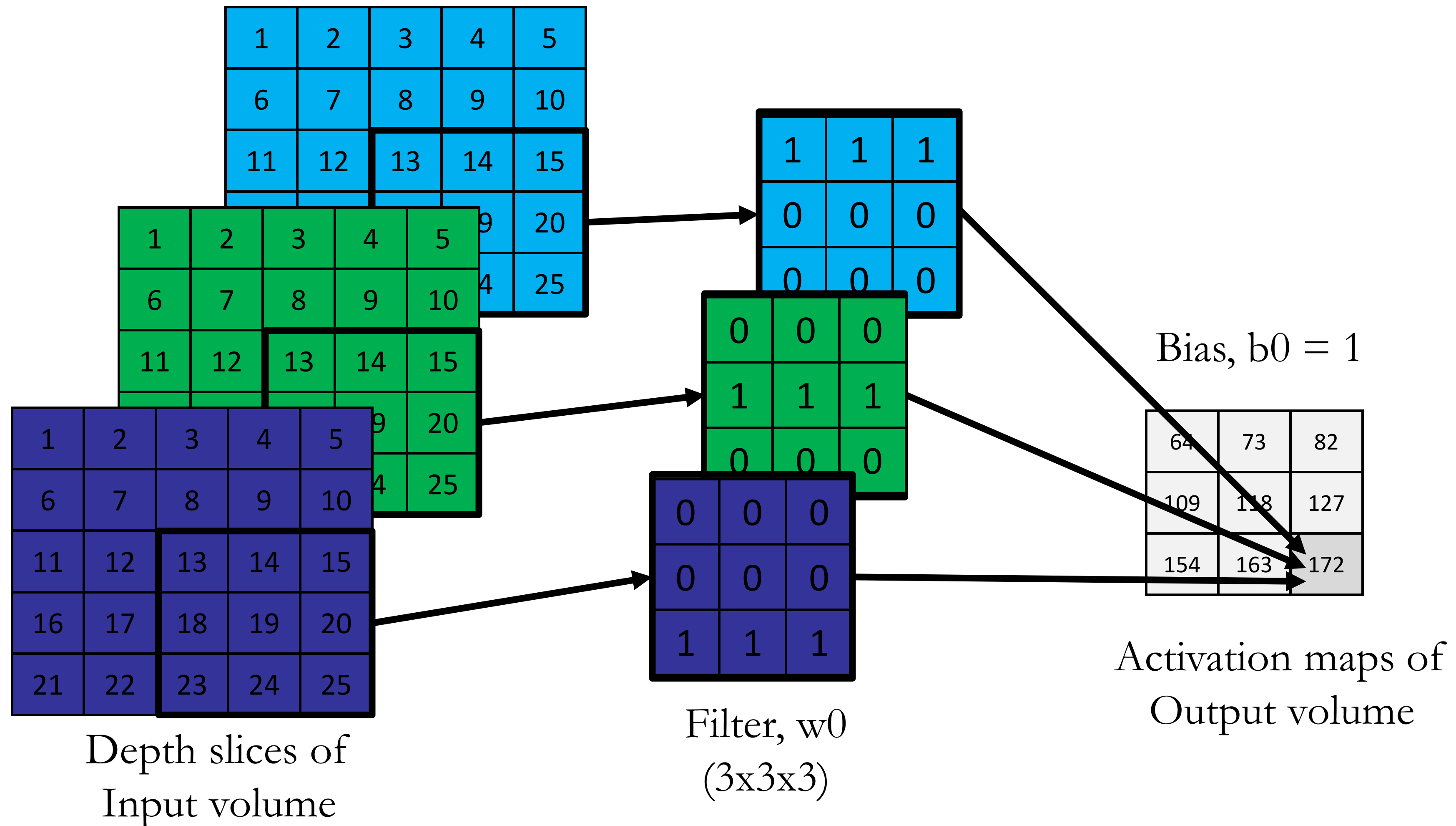
Example: Convolutional Layer with two 3x3x3 filters and Stride 1, Zero-padding 0

Convolutional Layer (cont.)



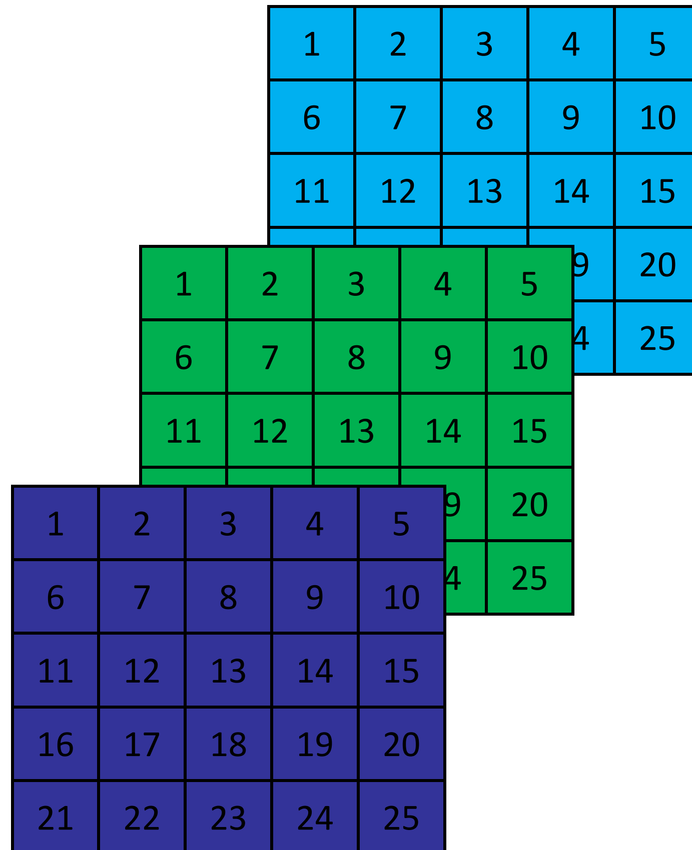
Example: Convolutional Layer with two 3x3x3 filters and Stride 1, Zero-padding 0

Convolutional Layer (cont.)



Example: Convolutional Layer with two 3x3x3 filters and Stride 1, Zero-padding 0

Convolutional Layer (cont.)



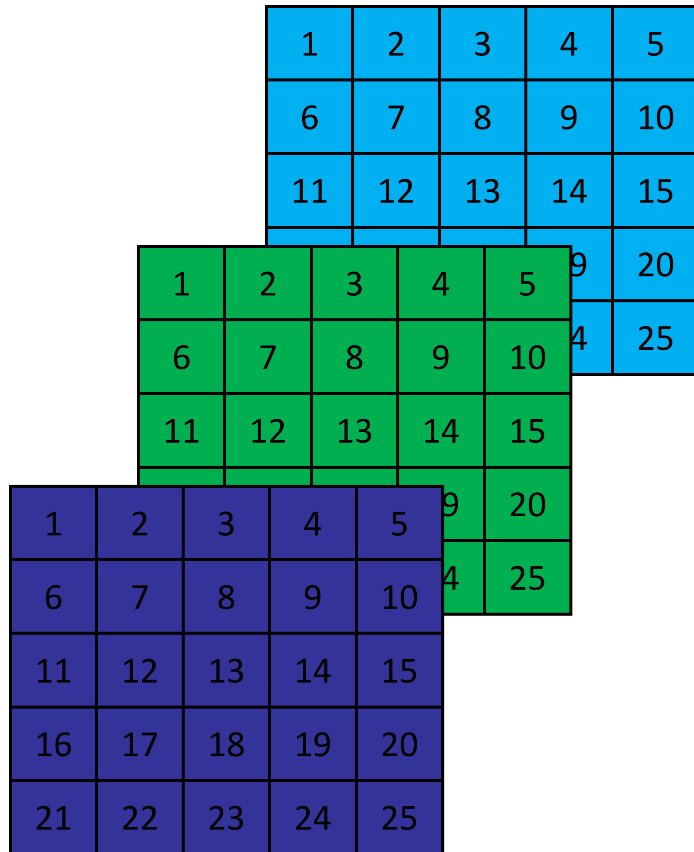
Depth slices of
Input volume

64	73	82
109	118	127
154	163	172

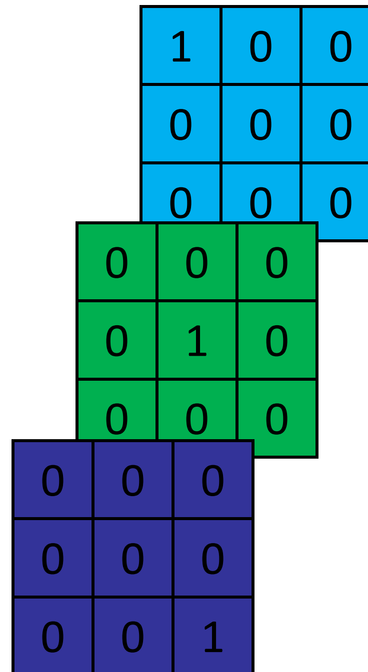
Activation maps of
Output volume

Example: Convolutional Layer with two 3x3x3 filters and Stride 1, Zero-padding 0

Convolutional Layer (cont.)

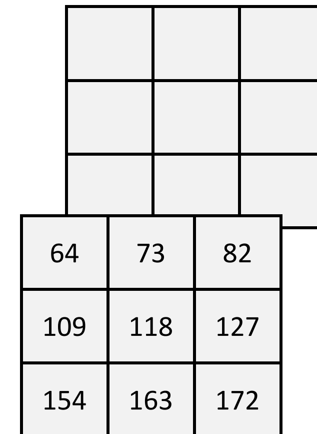


Depth slices of Input volume



Filter, w1 (3x3x3)

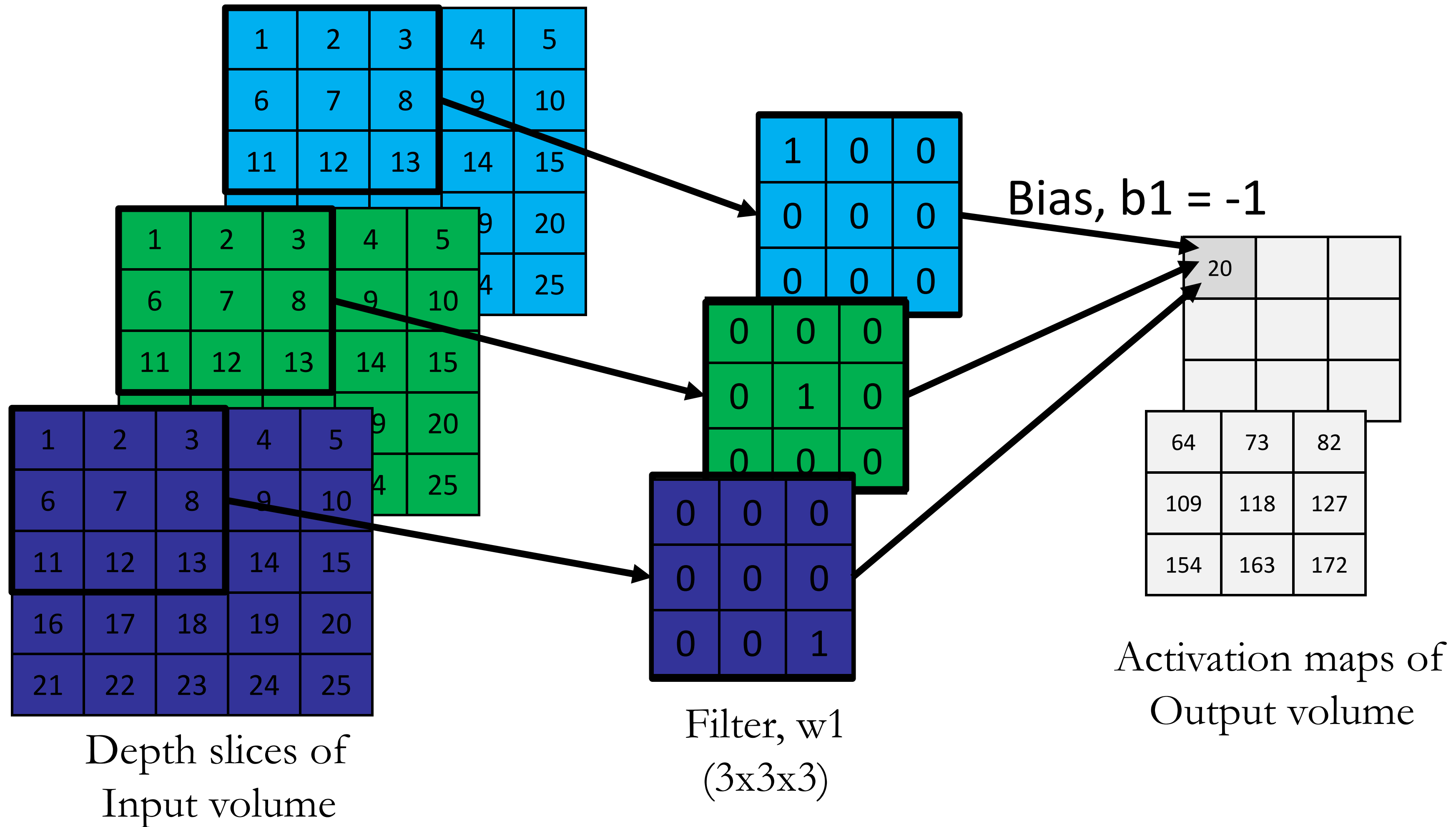
Bias, b1 = -1



Activation maps of Output volume

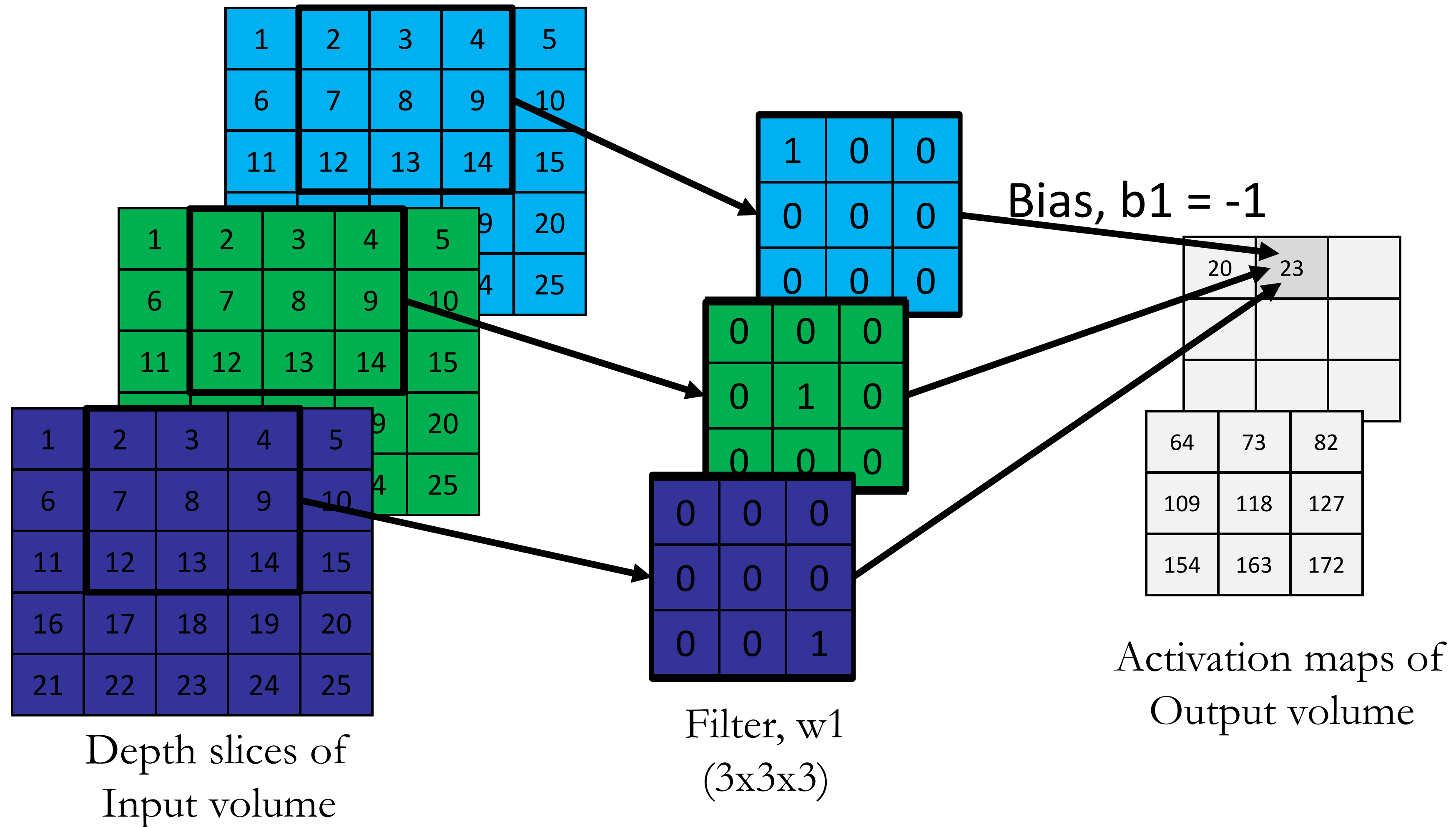
Example: Convolutional Layer with two 3x3x3 filters and Stride 1, Zero-padding 0

Convolutional Layer (cont.)



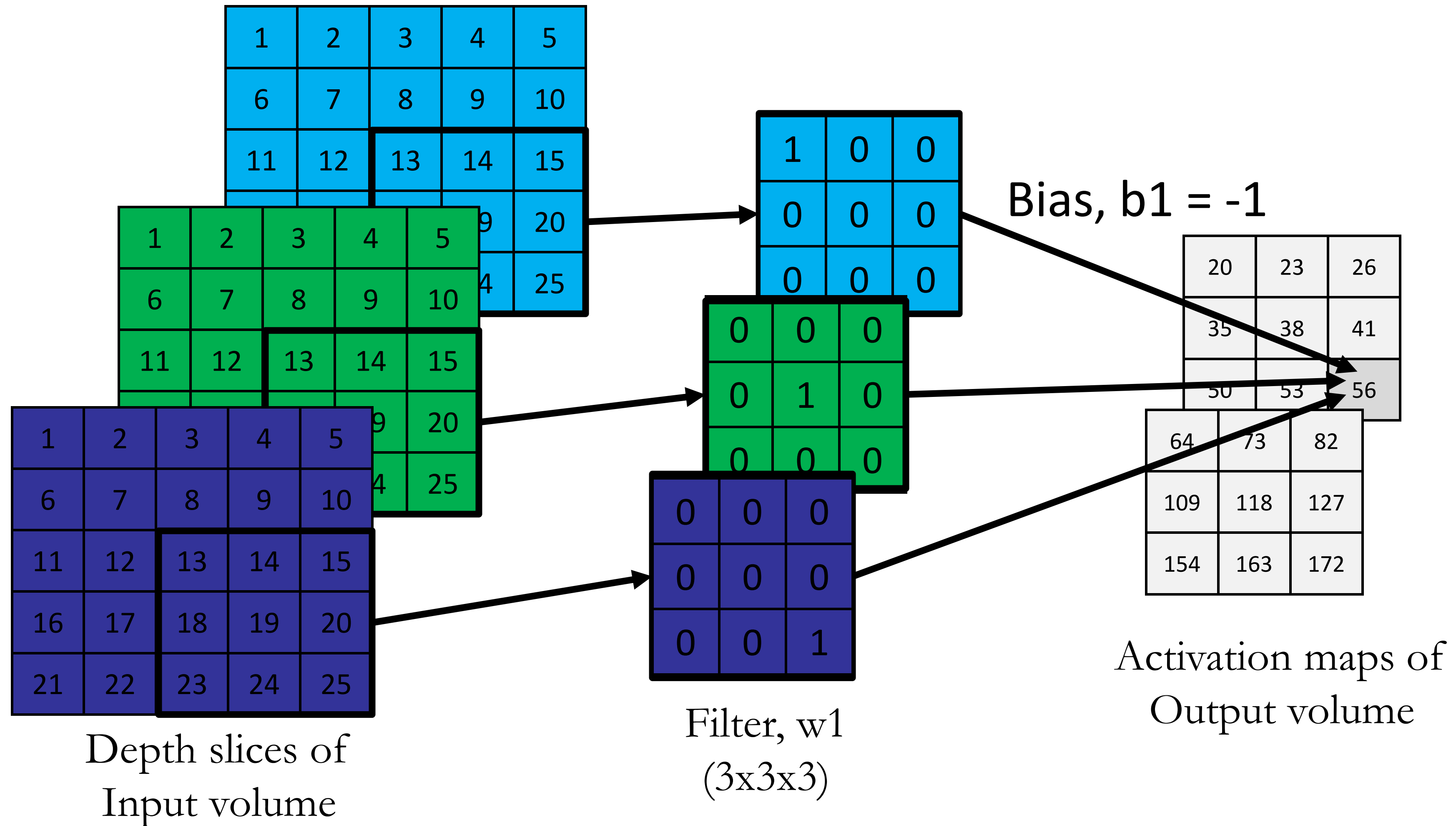
Example: Convolutional Layer with two 3x3x3 filters and Stride 1, Zero-padding 0

Convolutional Layer (cont.)



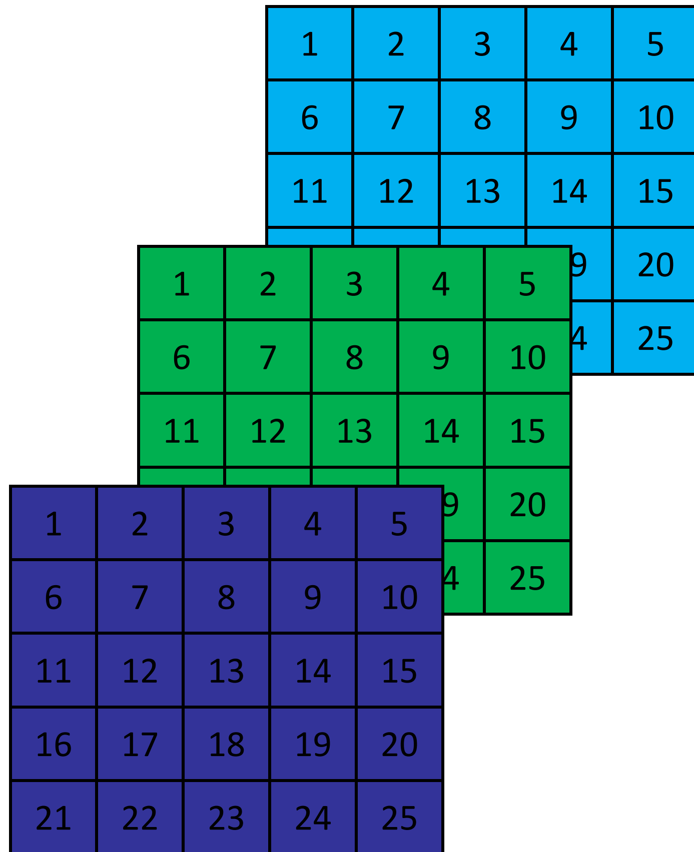
Example: Convolutional Layer with two 3x3x3 filters and Stride 1, Zero-padding 0

Convolutional Layer (cont.)

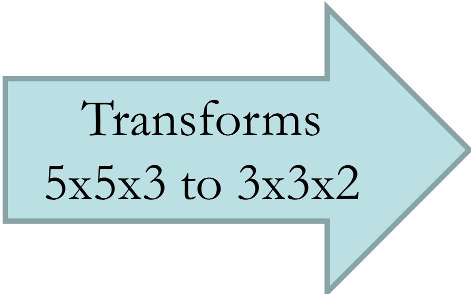


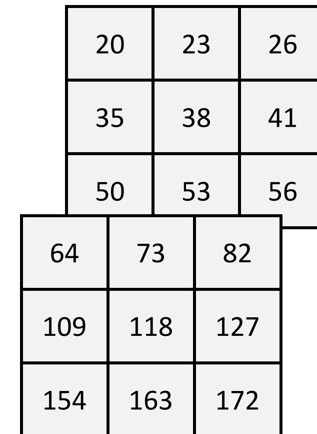
Example: Convolutional Layer with two 3x3x3 filters and Stride 1, Zero-padding 0

Convolutional Layer (cont.)



Depth slices of
Input volume





Activation maps of
Output volume

Example: Convolutional Layer with two 3x3x3 filters and Stride 1, Zero-padding 0

Convolutional Layer (cont.)

- From previous example, notice that:
  - The extent of the connectivity along the depth axis is 3, since this is the depth of the input volume
- Hyper-parameters:
  - Number of filters (K)
    - ➢How many different filters we want to use?
    - ➢Each of these filters will look for something different in the input
  - Spatial extend of filters (F)
    - ➢What is the width and height of filter?

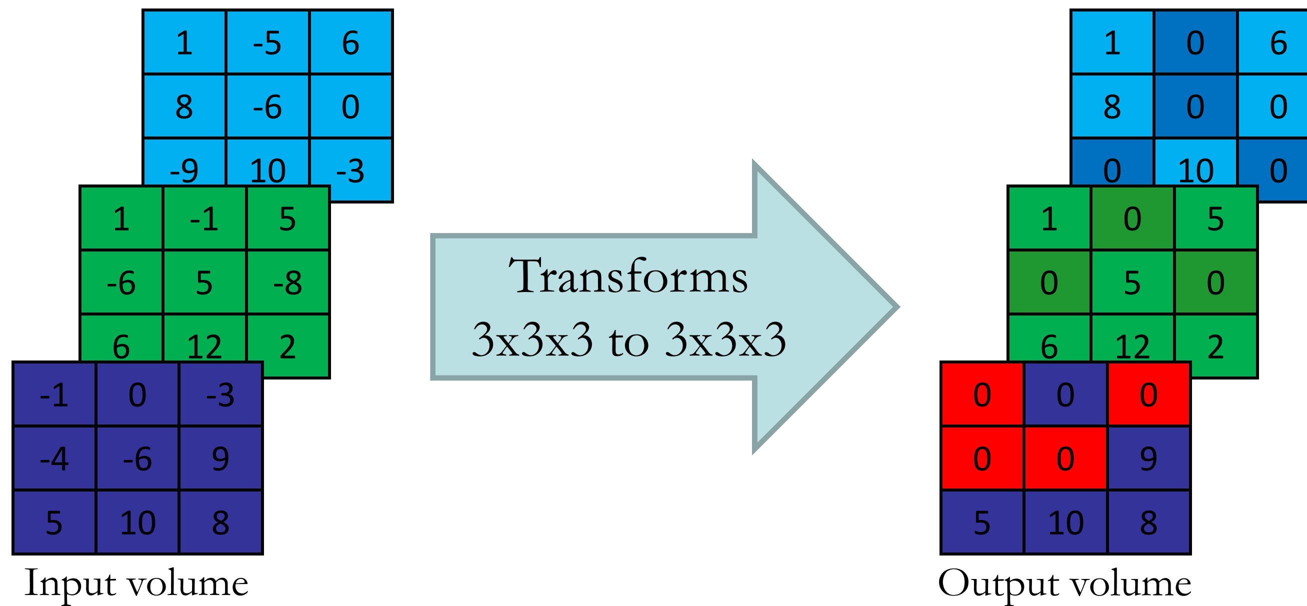
Convolutional Layer (cont.)

- Hyper-parameters:
  - Stride (S)
    - ➢How far we should slide the filter?
    - ➢When S=1, then we move the filters one pixel at a time
  - Amount of Zero-padding (P)
    - ➢Do we want to pad the input volume with zeros around the spatial border? If yes, how much?
- Parameters:
  - Weights of the filters (w)
  - Biases (b)

ReLU Layer

- We use ReLU (Rectified Linear Units) for non-linearity
- ReLU layer applies an element-wise activation function
  - max(0, x): thresholding at zero
- This leaves the dimensions of the input volume unchanged
- ReLU layer does not have any hyper-parameter and parameter

ReLU Layer



Example: ReLU Layer

Pooling Layer

- Pooling Layer operates independently on every depth slice of input volume
- Thus, spatial size (width, height) is reduced while the depth remains unchanged
- There are several types of Pooling layers:
  - Max pooling
    - Takes the maximum value from receptive fields
    - Works better in practice
  - Average pooling
    - Computes average value from receptive fields
    - Was often used historically

Pooling Layer (cont.)

1	5	6	1
8	6	8	2
9	10	3	4
6	1	5	9

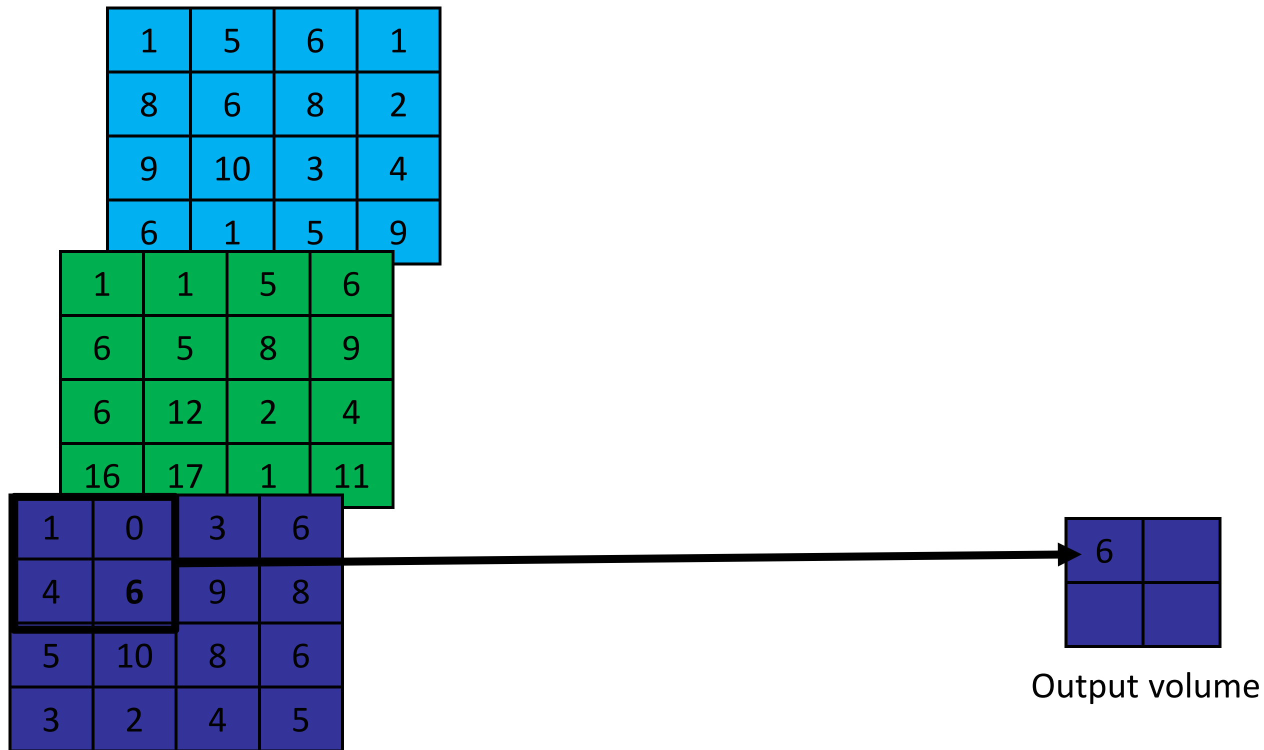
1	1	5	6
6	5	8	9
6	12	2	4
16	17	1	11

1	0	3	6
4	6	9	8
5	10	8	6
3	2	4	5

Depth slices of Input volume

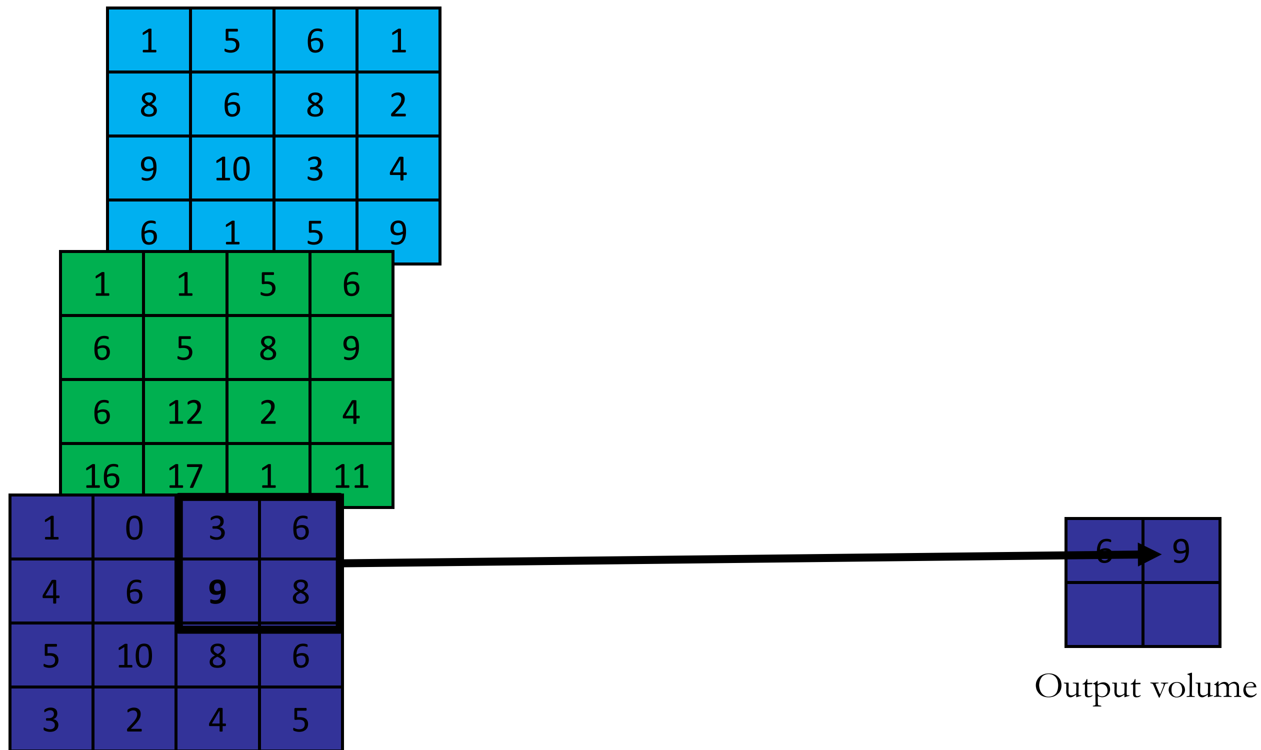
Example: 2x2 Max Pooling with Stride 2

Pooling Layer (cont.)



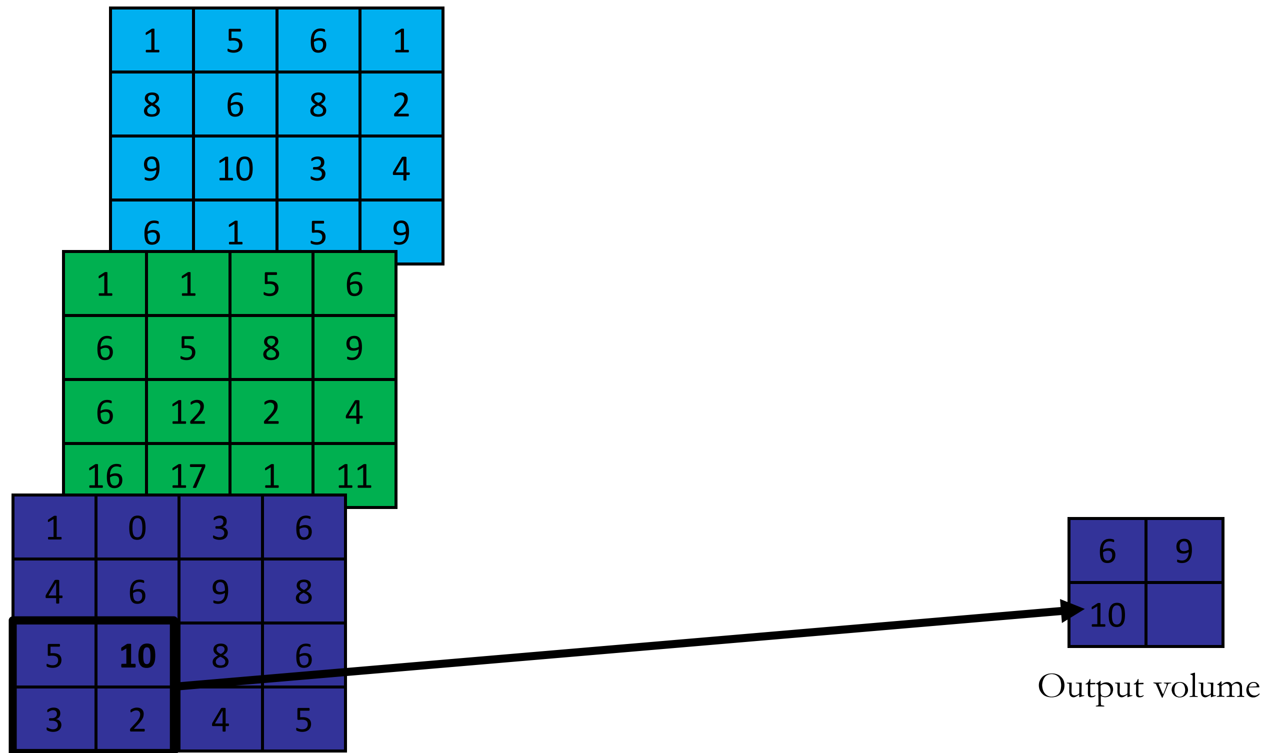
Example: 2x2 Max Pooling with Stride 2

Pooling Layer (cont.)



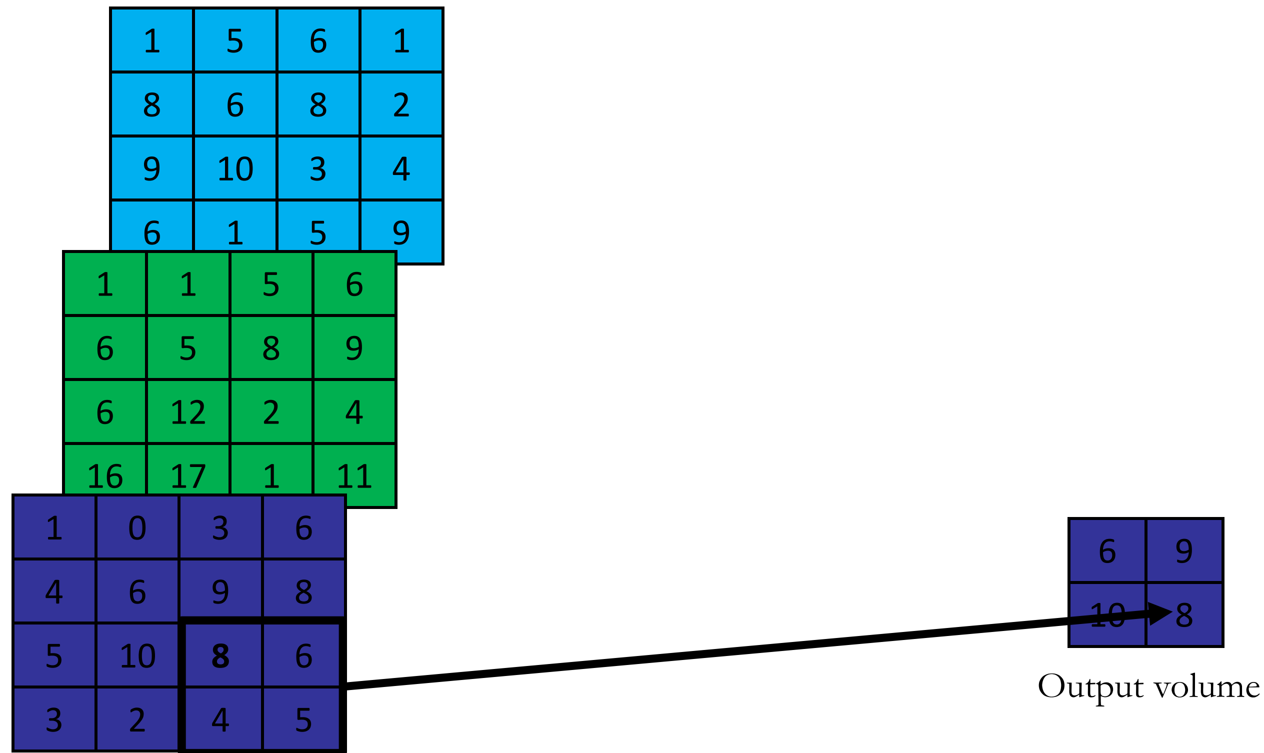
Example: 2x2 Max Pooling with Stride 2

Pooling Layer (cont.)



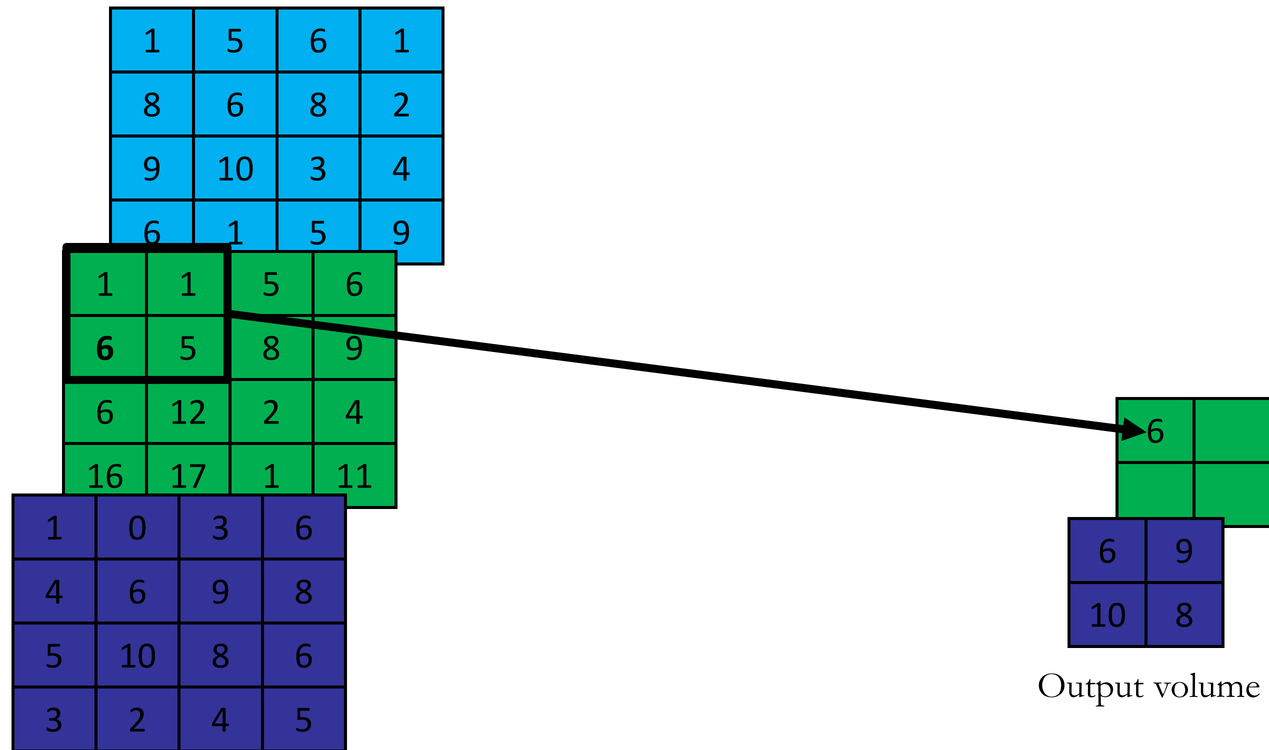
Example: 2x2 Max Pooling with Stride 2

Pooling Layer (cont.)



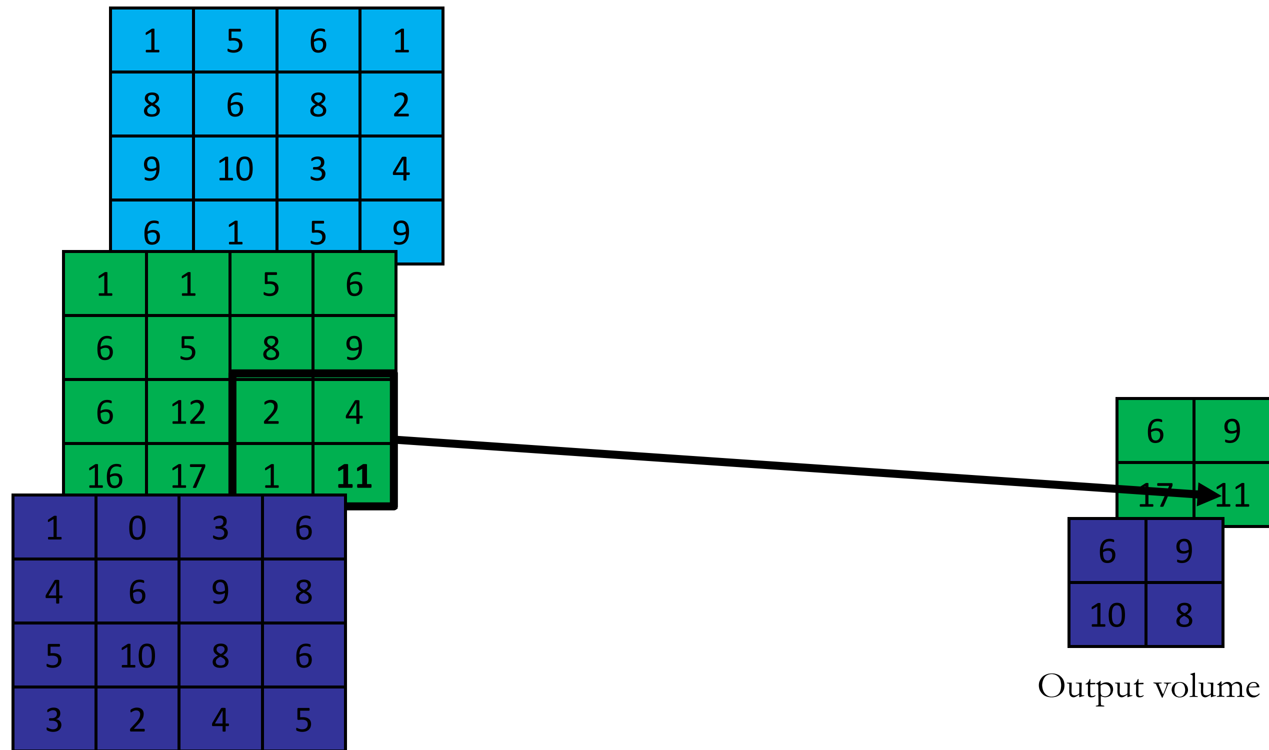
Example: 2x2 Max Pooling with Stride 2

Pooling Layer (cont.)



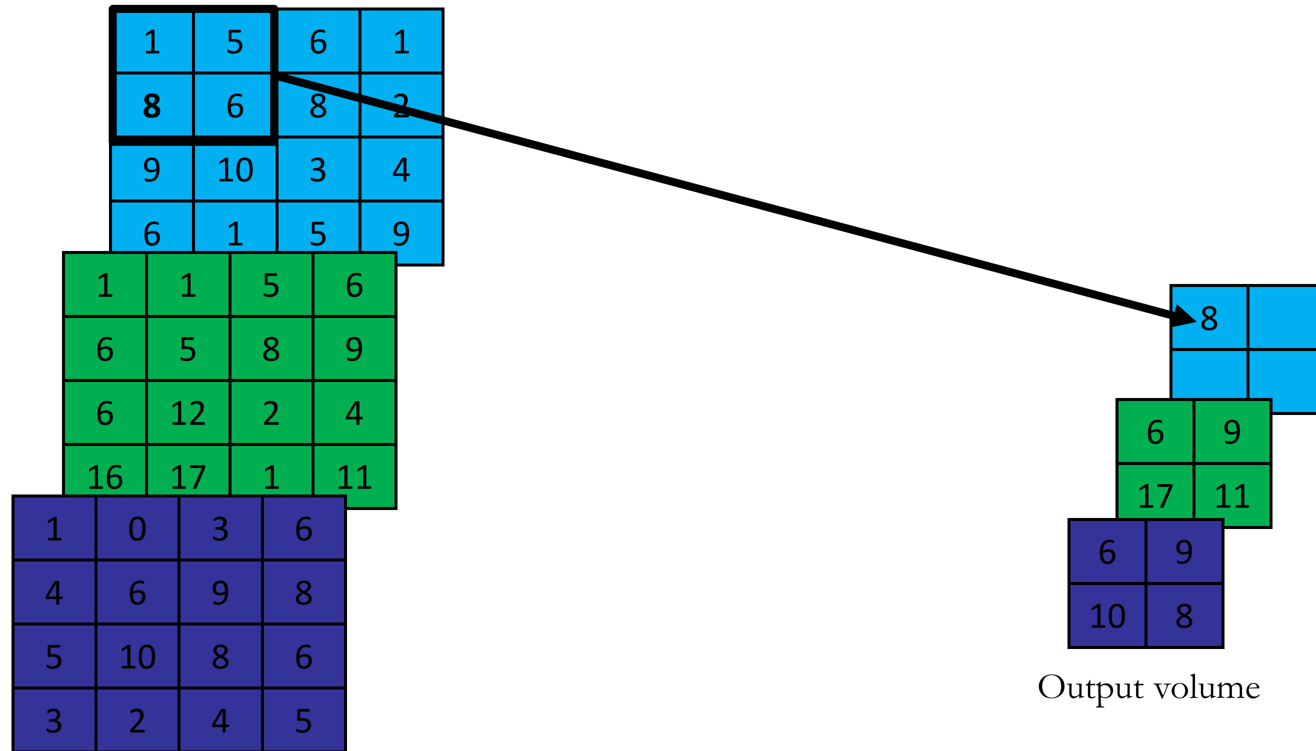
Example: 2x2 Max Pooling with Stride 2

Pooling Layer (cont.)



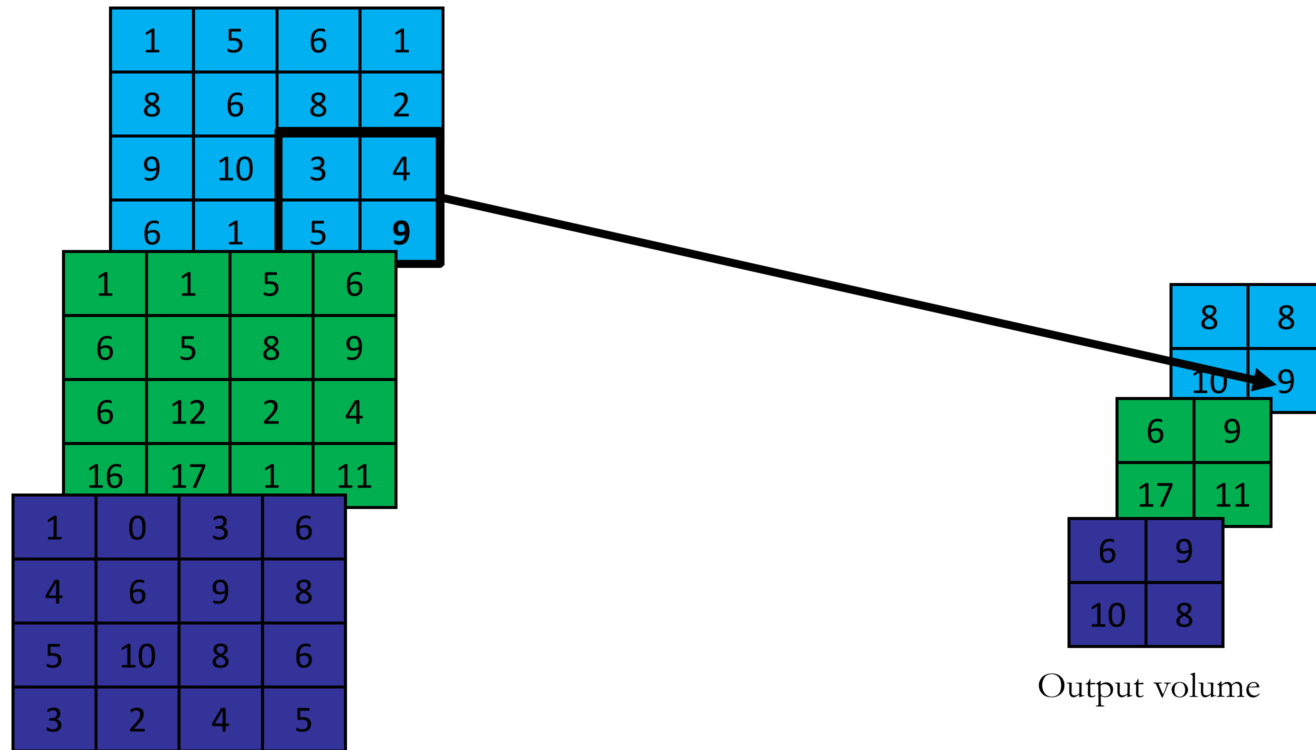
Example: 2x2 Max Pooling with Stride 2

Pooling Layer (cont.)



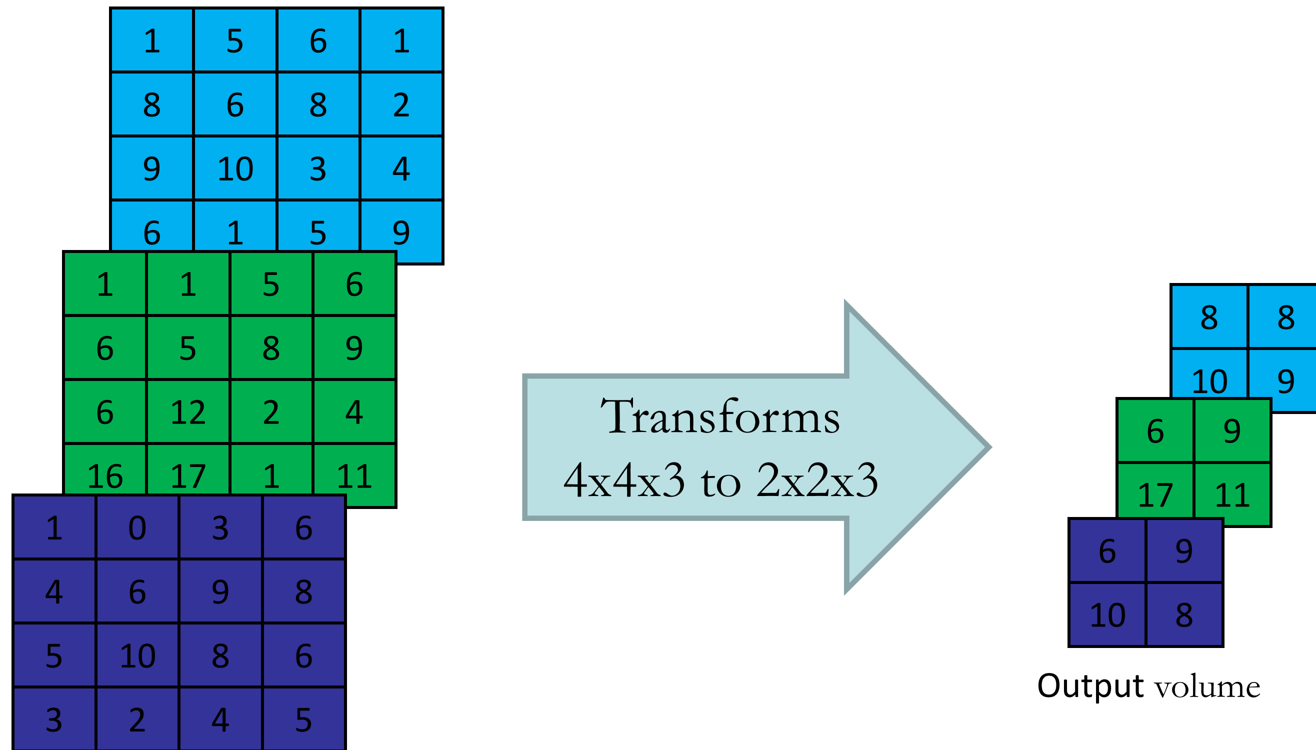
Example: 2x2 Max Pooling with Stride 2

Pooling Layer (cont.)



Example: 2x2 Max Pooling with Stride 2

Pooling Layer (cont.)



Example: 2x2 Max Pooling with Stride 2

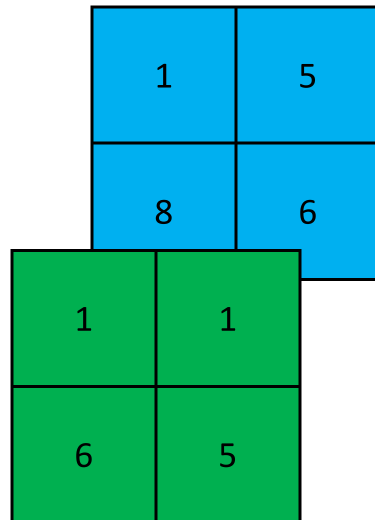
Pooling Layer (cont.)

- Hyper-parameters:
  - Spatial extent (F)
    - width and height of receptive field
  - Stride (S)
    - how far should we go while sliding?
- Parameters:
  - No parameters, since it computes a fixed function of the input (max, avg)
- In CNN architecture, we usually insert a Pooling Layer in-between successive Convolutional Layers

Fully-connected Layer

- A Fully-connected layer takes all neurons from the previous layer, and connects it to every single neuron it has
- Therefore, they are similar to the traditional/regular neural network layers
- They are placed at that end of CNN architecture after several convolutional and pooling layers
- Usually, in fully-connected layers, width = 1 and height = 1. So, they are only expanded along the depth
- The purpose of the fully-connected layer is to use high-level features (outputs from convolutional and pooling layers) for classifying the input image into various classes

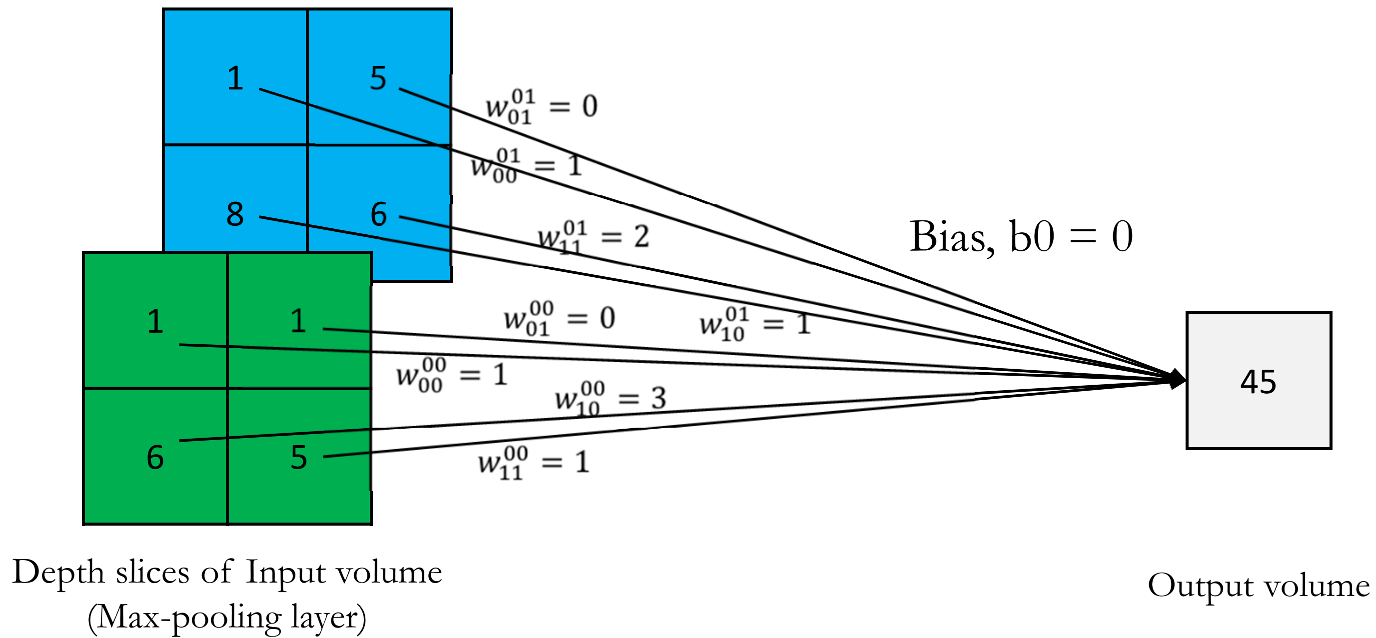
Fully-connected Layer (cont.)



Depth slices of Input volume
(Max-pooling layer)

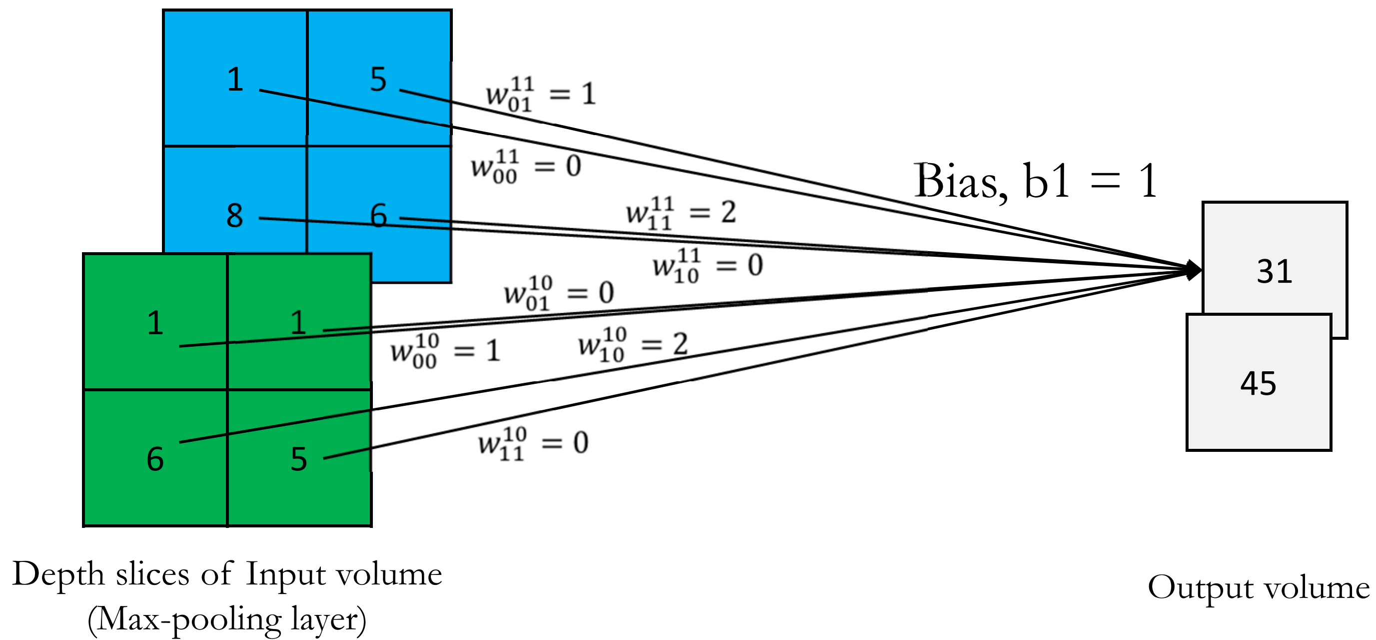
Example: Fully-connected Layer

Fully-connected Layer (cont.)



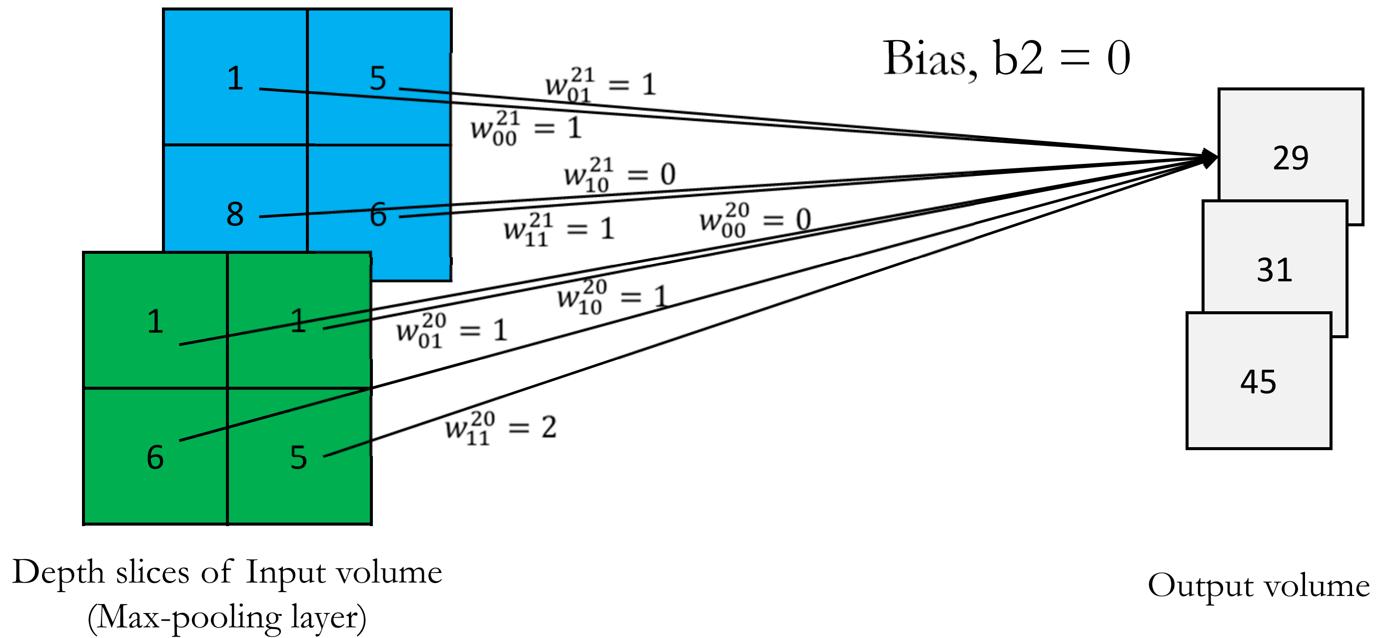
Example: Fully-connected Layer

Fully-connected Layer (cont.)



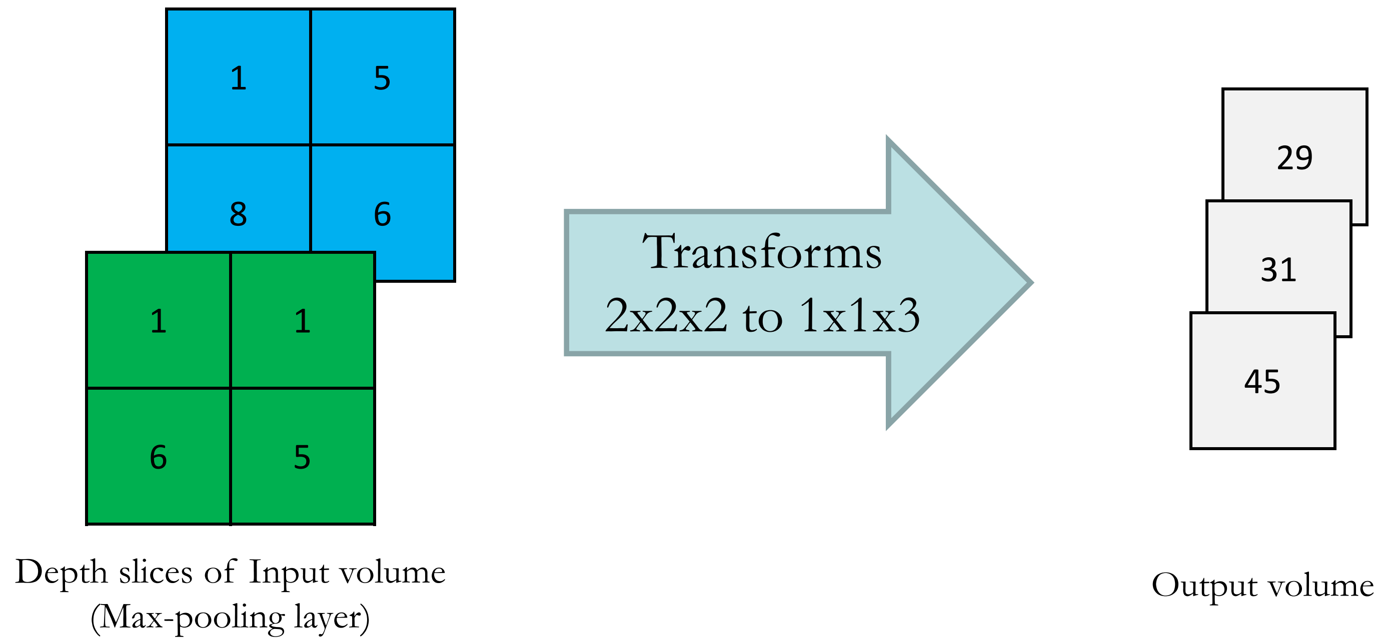
Example: Fully-connected Layer

Fully-connected Layer (cont.)



Example: Fully-connected Layer

Fully-connected Layer (cont.)



Example: Fully-connected Layer

Fully-connected Layer (cont.)

- A non-linear activity function (ReLU) or a Softmax activation may follow a fully-connected layer
  - We should use Softmax as the activation function, when a fully-connected layer is being used as the output layer. This ensures that the sum of output probabilities is 1
- Hyper-parameters:
  - Depth: number of neurons along the depth axis
- Parameters:
  - Weights (w)
  - Biases (b)

CNN Layer Patterns

- The most common CNN architectures follow the pattern:

INPUT --> [[CONV -> RELU]*N --> POOL?]*M --> [FC --> RELU]*K --> FC

where,

* = repetition,

? = optional layer,

N >= 0,

N <= 3 (usually),

M >= 0,

K >= 0, and

K < 3 (usually)

Some popular CNN architectures

- **LeNet**
  - Developed by Yann LeCun in 1990
  - Was used to read zip codes, digits, etc.
- **AlexNet**
  - Winner, ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2012
  - The first work that popularized CNN in Computer Vision
- **ZF Net**
  - Winner, ILSVRC 2013
  - Expanded the size of the middle convolutional layers and made the stride and filter size on the first layer smaller

Some popular CNN architectures (cont.)

- **GoogLeNet**
  - Winner, ILSVRC 2014
  - Dramatically reduced the number of parameters in the network
- **VGGNet**
  - Runner-up, ILSVRC 2014
  - Showed that, the depth of the network is a critical component for good performance
- **ResNet**
  - Winner, ILSVRC 2015
  - By May,2016, this was the state of the art Convolutional Neural Network model

A closer look at VGGNet

- VGGNet is composed of:
  - Convolutional layers that perform 3x3 convolutions with stride 1 and zero-padding 1
  - Pooling layers that perform 2x2 max pooling with stride 2
  - Fully-connected layers
- Now, we will take a closer look at VGGNet's:
  - Layers and their hyper-parameters
  - Dimensions (size of the representation), number of neurons, and total number of weights at each layer

A closer look at VGGNet (cont.)

Layers	Dimension	Number for neurons	Number of weight-parameters
INPUT	224 x 224 x 3	224*224*3 = 150K	0
CONV3-64	224 x 224 x 64	3.2M	(3*3*3)*64 = 1,728
CONV3-64	224 x 224 x 64	3.2M	(3*3*64)*64 = 36,864
POOL2	112 x 112 x 64	800K	0
CONV3-128	112 x 112 x 128	1.6M	(3*3*64)*128 = 73,728
CONV3-128	112 x 112 x 128	1.6M	(3*3*128)*128 = 147,456
POOL2	56 x 56 x 128	400K	0
CONV3-256	56 x 56 x 256	800K	(3*3*128)*256 = 294,912
CONV3-256	56 x 56 x 256	800K	(3*3*256)*256 = 589,824
CONV3-256	56 x 56 x 256	800K	(3*3*256)*256 = 589,824
POOL2	28 x 28 x 256	200K	0
CONV3-512	28 x 28 x 512	400K	(3*3*256)*512 = 1,179,648
CONV3-512	28 x 28 x 512	400K	(3*3*512)*512 = 2,359,296
CONV3-512	28 x 28 x 512	400K	(3*3*512)*512 = 2,359,296
POOL2	14 x 14 x 512	100K	0
CONV3-512	14 x 14 x 512	100K	(3*3*512)*512 = 2,359,296

A closer look at VGGNet (cont.)

Layers	Dimension	Number for neurons	Number of weight-parameters
CONV3-512	14 x 14 x 512	100K	(3*3*512)*512 = 2,359,296
CONV3-512	14 x 14 x 512	100K	(3*3*512)*512 = 2,359,296
POOL2	7 x 7 x 512	25K	0
FC	1 x 1 x 4096	4096	7*7*512*4096 = 102,760,448
FC	1 x 1 x 4096	4096	4096*4096 = 16,777,216
FC	1 x 1 x 1000	1000	4096*1000 = 4,096,000

- To calculate the actual memory (in bytes) for neurons in a layer, we need to multiply “Number of neurons” by 4 or 8
  - We save each of the neurons in float or double data type, which usually takes 4 or 8 bytes
- Notice that, most of the memory is used in the early convolutional layers. And, most of the parameters are in the last Fully-connected layers

Training

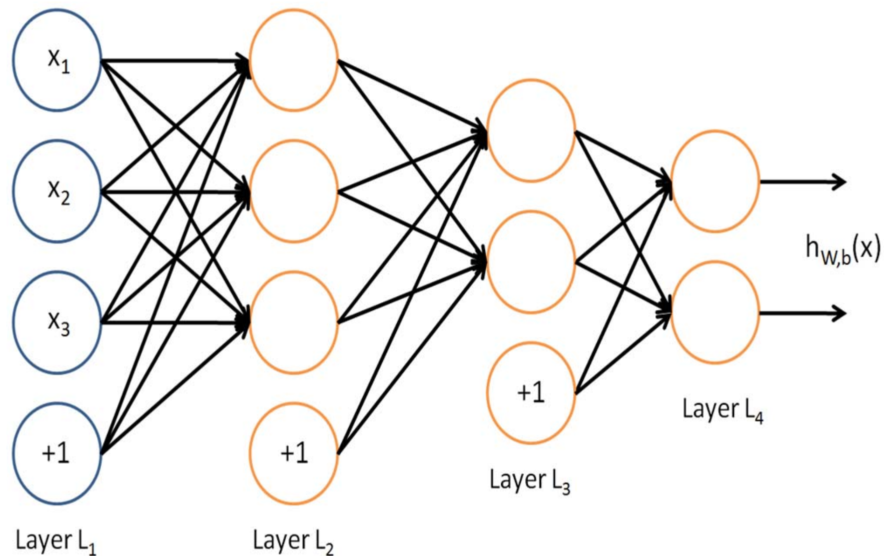
How to train it: how to train the weights (W) and biases (b) (use forward, backward propagation)

Initialize W and b randomly

Iter=1: all_epochs (each is called an epoch)

- Forward propagation for each output neuron:
  - Use training samples: X_{class_t} : feed forward to find y.
  - Err=error_function(y-t)
- Backward propagation:
  - Find ΔW and Δb to reduce Err.
  - $W_{new}=W_{old}+\Delta W$; $b_{new}=b_{old}+\Delta b$

BPNN



- **forward propagation:**

$$z^{(l+1)} = W^{(l)} a^{(l)} + b^{(l)}$$
$$a^{(l+1)} = f(z^{(l+1)})$$

- **cost function:**

$$J(W, b; x, y) = \frac{1}{2} \left\| h_{W,b}(x) - y \right\|^2.$$

- **Update the W and b:**

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b)$$
$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W, b)$$

BPNN

BP to compute $\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x,y)$ $\frac{\partial}{\partial b_{i}^{(l)}} J(W,b;x,y)$

Set $\delta_i^l = -\frac{\partial E}{\partial z_i^l}$

– Output layer: $\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \left\|y - h_{W,b}(x)\right\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$

– Other layers: $\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$

BPNN

BP to compute $\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x,y)$ $\frac{\partial}{\partial b_i^{(l)}} J(W,b;x,y)$

- We can compute $\nabla_{W^{(l)}} J$ and $\nabla_{b^{(l)}} J$ as follows:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x,y) = a_j^{(l)} \delta_i^{(l+1)}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W,b;x,y) = \delta_i^{(l+1)}.$$

CNN – Backpropagation

Compute the ∇w and ∇b of:

- Fully connected layer
- Convolutional layer
- Pooling layer

CNN – Backpropagation- Fully Connected Layer

Use the same BP algorithm as Neural networks.

- The delta of output layer:
- $\delta^L = \frac{\partial E}{\partial z^L}$, E is the loss function.
- Delta of fully connected layer:
- $\delta^{L-1} = \frac{\partial E}{\partial z^{L-1}} = \left(\left(w^{(L-1)}\right)^T \delta^L\right) \cdot f'(z^{L-1})$

CNN – Backpropagation- Pooling Layer

As there is no non-linearity function in this layer, delta is only calculated:

$$\delta_k^{(l)} = \text{upsample}\left((W_k^{(l)})^T \delta_k^{(l+1)}\right) \bullet f'(z_k^{(l)})$$

Use **convn** function:

convn(net.layers{l + 1}.d{j}, rot180(net.layers{l + 1}.k{i}{j}), 'full')

```
for i = 1 : numel(net.layers{l}.a)
        z = zeros(size(net.layers{l}.a{1}));
        for j = 1 : numel(net.layers{l + 1}.a)
            z = z + convn(net.layers{l + 1}.d{j},- rot180(net.layers{l + 1}.k{i}{j}), 'full');
        end
        net.layers{l}.d{i} = z;
end
```

CNN – Backpropagation- Convolutional Layer

2 steps:

- To calculate delta: $\delta^l = upsample(\delta^{l+1}) \cdot f'(z^l)$, $\cdot$ is dot product
- To calculate $\nabla w, \nabla b$
  - $\nabla w^{l-1} = \delta^l (a^{l-1})^T$
  - $\nabla b^{l-1} = \delta^l$

CNN – Backpropagation- Convolutional Layer

- Discussion about upsample function: $\delta^{l} = upsample(\delta^{l+1}) \cdot f'(z^{l})$,
- Should know the map locations between 2 layers.
  - Max pooling: record the location from the forward pass.
  - Mean pooling:

References

- http://cs231n.github.io/convolutional-networks
- https://www.tensorflow.org/
- https://github.com/dmlc/mxnet
- http://deeplearning.net/software/theano/
- http://caffe.berkeleyvision.org/
- http://pytorch.org/