Convolutional Neural Networks

Laura Graesser

WiMLDS, January 18th, LinkedIn

What we will cover today

- Overview of, and motivation for convolutional neural networks (CNNs)
- 2D and 3D convolutions
- Components of a convolutional layer
  - Kernels
  - Padding
  - Stride
  - Dilation
- Pooling layer
- A simple CNN in pyTorch
- VGG-Net
- Residual Networks

What are Convolutional Neural Networks?

Recap: Multi-layered perceptron

- Encodes little knowledge about the input domain
- Globally connected layers
  - Every node in a layer is connected to every node in the layer below
- Input is a 1-D vector
- Many parameters

Convolutional Neural Network

- Encodes knowledge about the structure of the input domain
- Locally connected layers with shared weights
- Specifically, weights are shared across space
- 2-D, 3-D, or 4-D inputs
- Typically fewer parameters than pure MLPs

Motivation

Biology:

- Hierarchical nature of our visual processing system
- Simple to complex pattern responses
- Local connections are gradually integrated into global connections

Machine Learning:

- Reducing number of free parameters helps generalization
- How to reduce in a smart way?
  Incorporate prior knowledge about images and human visual processing system

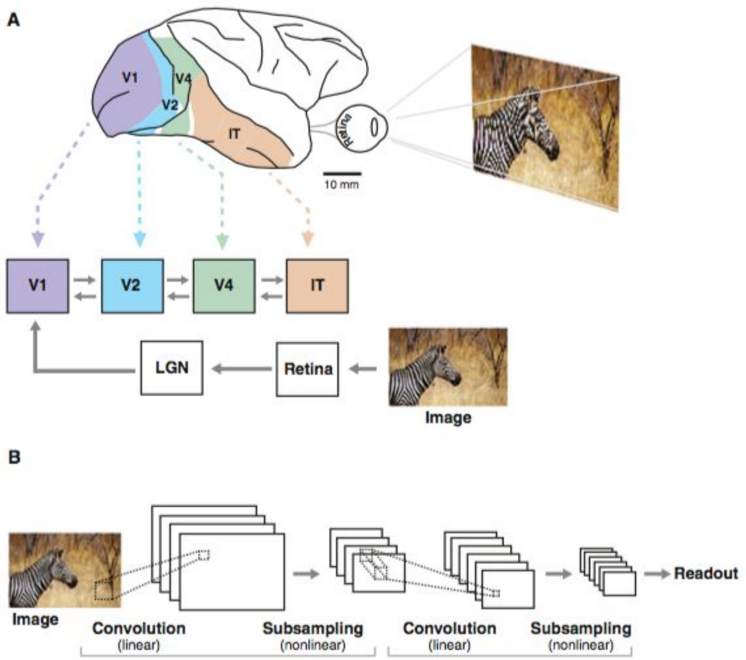


Image Source: Neural Networks and Neuroscience Inspired Computer Vision, Cox & Dean, 2014

2D Convolution

Input Image: 6 x 6

a11	a12	a13	a14	a15	a16
a21	a22	a23	a24	a25	a26
a31	a32	a33	a34	a35	a36
a41	a42	a43	a44	a45	a46
a51	a52	a53	a54	a55	a56
a61	a62	a63	a64	a65	a66

*

Kernel: 3 x 3

k11	k12	k13
k21	k22	k23
k31	k32	k33

=

Output Image: 4 x 4

b11	b12	b13	b14
b21	b22	b23	b24
b31	b32	b33	b34
b41	b42	b43	b44

*b11 = a11 * k11 + a12 * k12 + a13 * k13 +*
*a21 * k21 + a22 * k22 + a23 * k23 +*
*a31 * k31 + a32 * k32 + a33 * k33 +*

2D Convolution

Input Image: 6 x 6

a11	a12	a13	a14	a15	a16
a21	a22	a23	a24	a25	a26
a31	a32	a33	a34	a35	a36
a41	a42	a43	a44	a45	a46
a51	a52	a53	a54	a55	a56
a61	a62	a63	a64	a65	a66

*

Kernel: 3 x 3

k11	k12	k13
k21	k22	k23
k31	k32	k33

=

Output Image: 4 x 4

b11	b12	b13	b14
b21	b22	b23	b24
b31	b32	b33	b34
b41	b42	b43	b44

$$\begin{aligned} b12 = \; & a12 * k11 + a13 * k12 + a14 * k13 + \\ & a22 * k21 + a23 * k22 + a24 * k23 + \\ & a32 * k31 + a33 * k32 + a34 * k33 + \end{aligned}$$

2D Convolution

Input Image: 6 x 6

a11	a12	a13	a14	a15	a16
a21	a22	a23	a24	a25	a26
a31	a32	a33	a34	a35	a36
a41	a42	a43	a44	a45	a46
a51	a52	a53	a54	a55	a56
a61	a62	a63	a64	a65	a66

*

Kernel: 3 x 3

k11	k12	k13
k21	k22	k23
k31	k32	k33

=

Output Image: 4 x 4

b11	b12	b13	b14
b21	b22	b23	b24
b31	b32	b33	b34
b41	b42	b43	b44

$$
\begin{aligned}
b13 = \ & a13 * k11 + a14 * k12 + a15 * k13 + \\
& a23 * k21 + a24 * k22 + a25 * k23 + \\
& a33 * k31 + a34 * k32 + a35 * k33 +
\end{aligned}
$$

2D Convolution

Input Image: 6 x 6

a11	a12	a13	a14	a15	a16
a21	a22	a23	a24	a25	a26
a31	a32	a33	a34	a35	a36
a41	a42	a43	a44	a45	a46
a51	a52	a53	a54	a55	a56
a61	a62	a63	a64	a65	a66

*

Kernel: 3 x 3

k11	k12	k13
k21	k22	k23
k31	k32	k33

=

Output Image: 4 x 4

b11	b12	b13	b14
b21	b22	b23	b24
b31	b32	b33	b34
b41	b42	b43	b44

$$
\begin{aligned}
b14 = \; & a14 * k11 + a15 * k12 + a16 * k13 + \\
& a24 * k21 + a25 * k22 + a26 * k23 + \\
& a34 * k31 + a35 * k32 + a36 * k33 +
\end{aligned}
$$

2D Convolution

Input Image: 6 x 6

a11	a12	a13	a14	a15	a16
a21	a22	a23	a24	a25	a26
a31	a32	a33	a34	a35	a36
a41	a42	a43	a44	a45	a46
a51	a52	a53	a54	a55	a56
a61	a62	a63	a64	a65	a66

*

Kernel: 3 x 3

k11	k12	k13
k21	k22	k23
k31	k32	k33

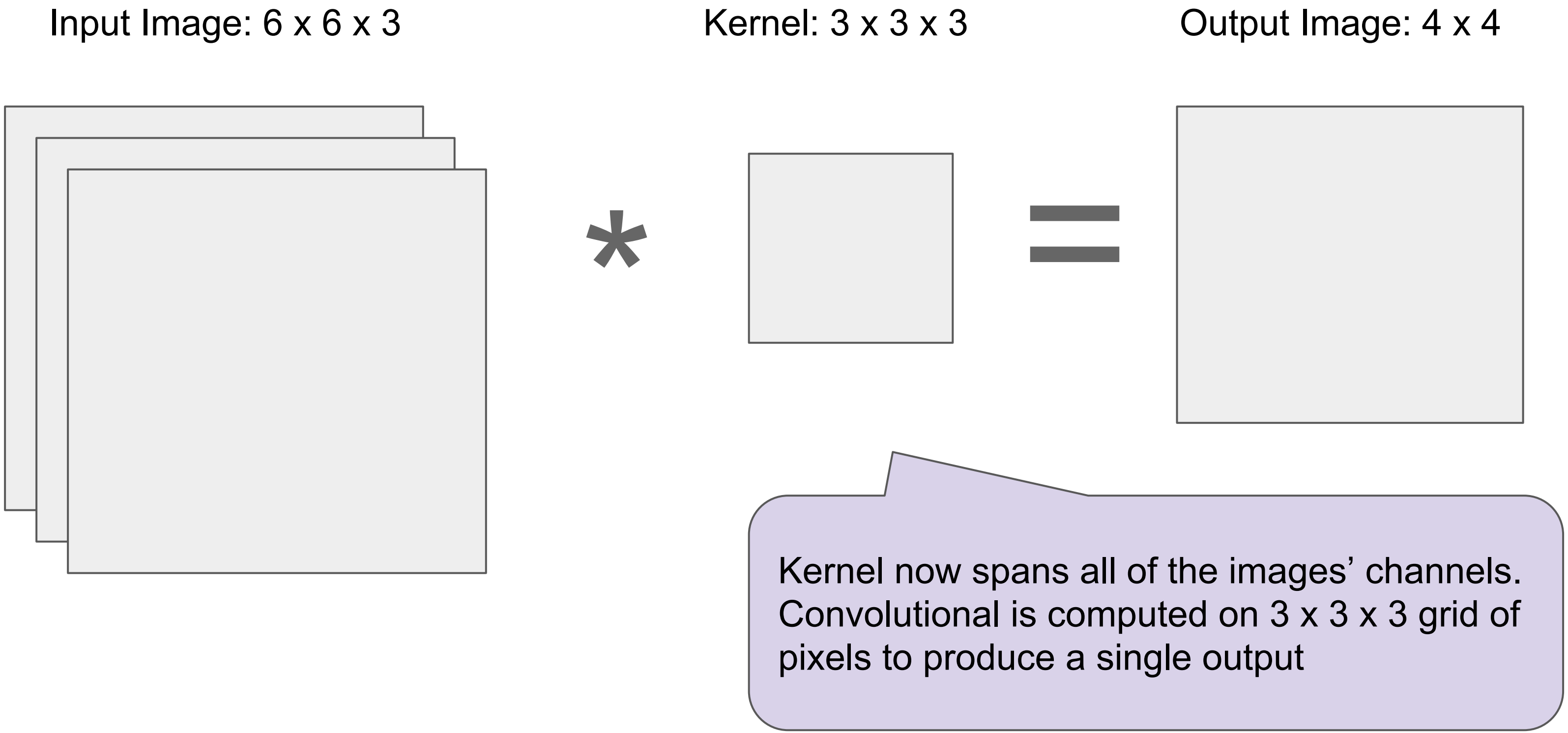
=

Output Image: 4 x 4

b11	b12	b13	b14
b21	b22	b23	b24
b31	b32	b33	b34
b41	b42	b43	b44

*b21 = a21 * k11 + a22 * k12 + a23 * k13 +*
*a31 * k21 + a32 * k22 + a33 * k23 +*
*a41 * k31 + a42 * k32 + a43 * k33 +*

3D Convolution



Convolutional Layers

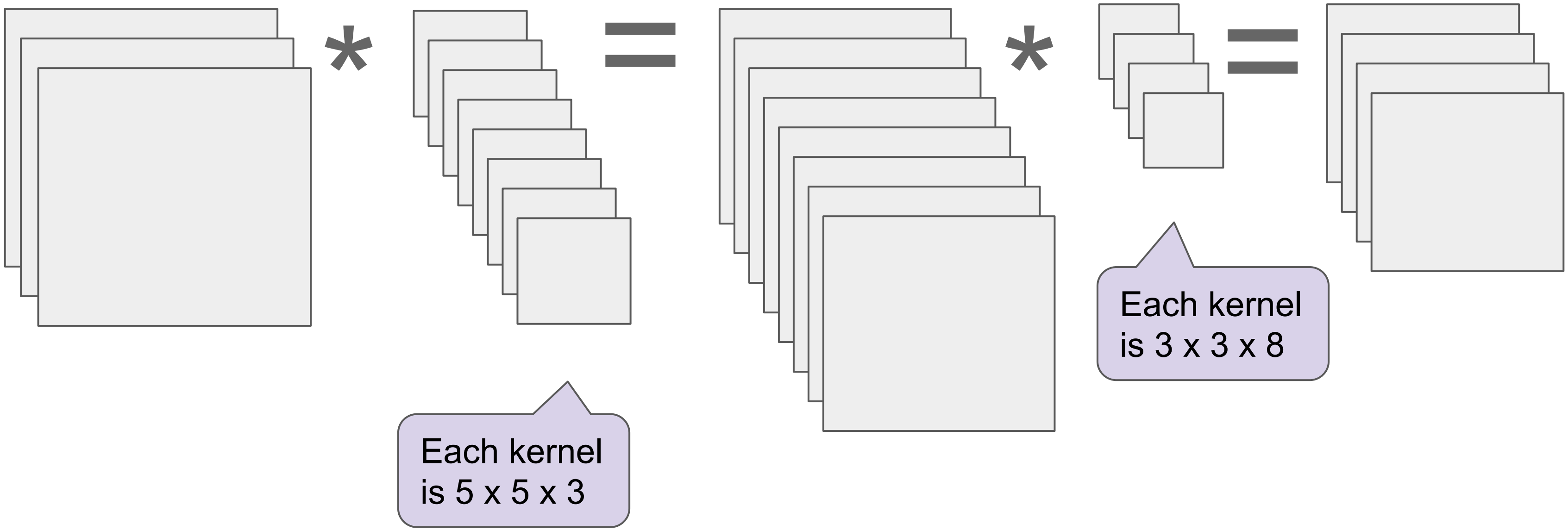
Input Image:
32 x 32 x 3

Layer 1:
8 kernels,
5 x 5 x 3

Output Layer 1:
28 x 28 x 8

Layer 2:
4 kernels,
3 x 3 x 8

Output Layer 2:
26 x 26 x 4



Kernels compute features

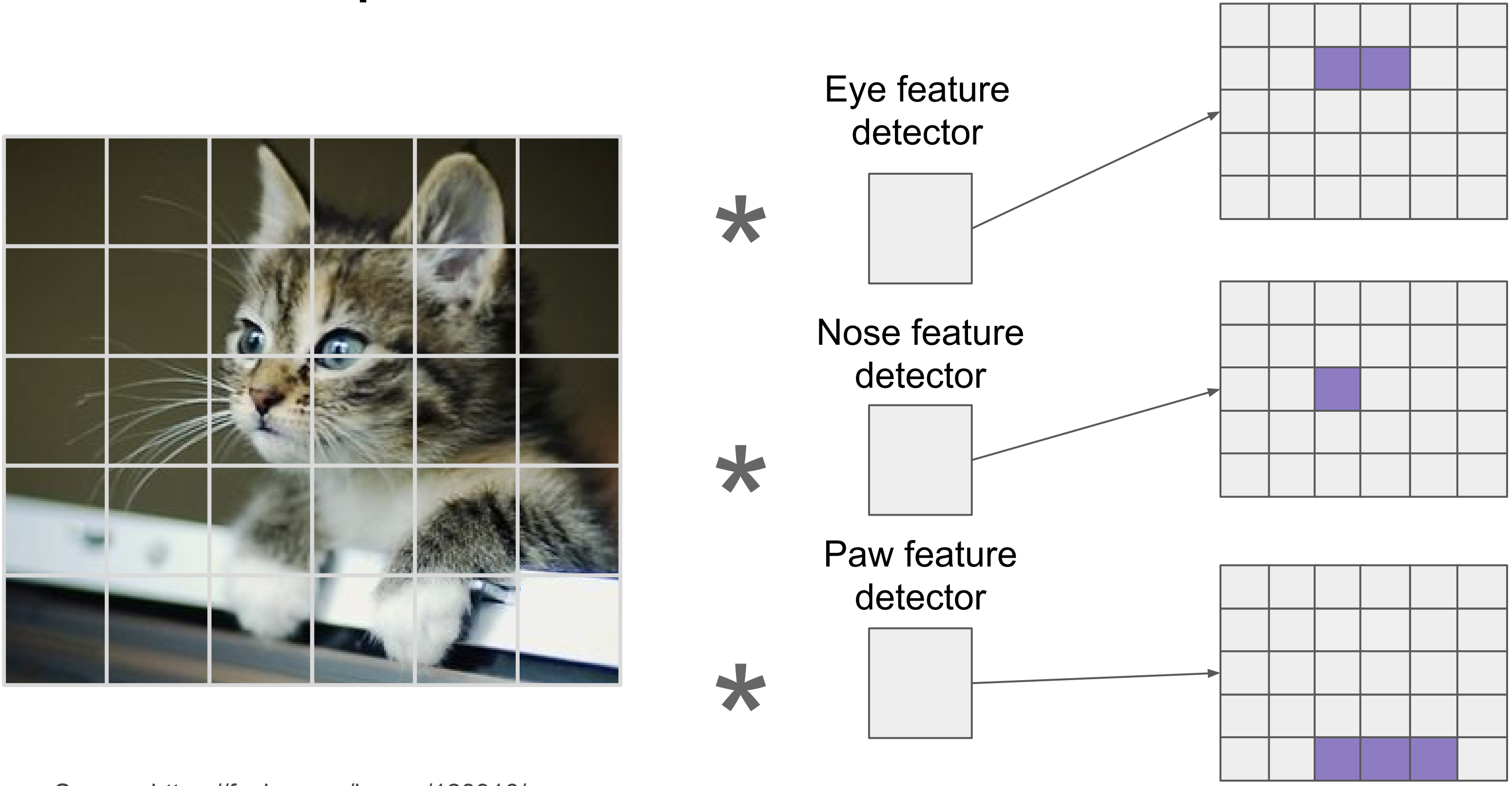
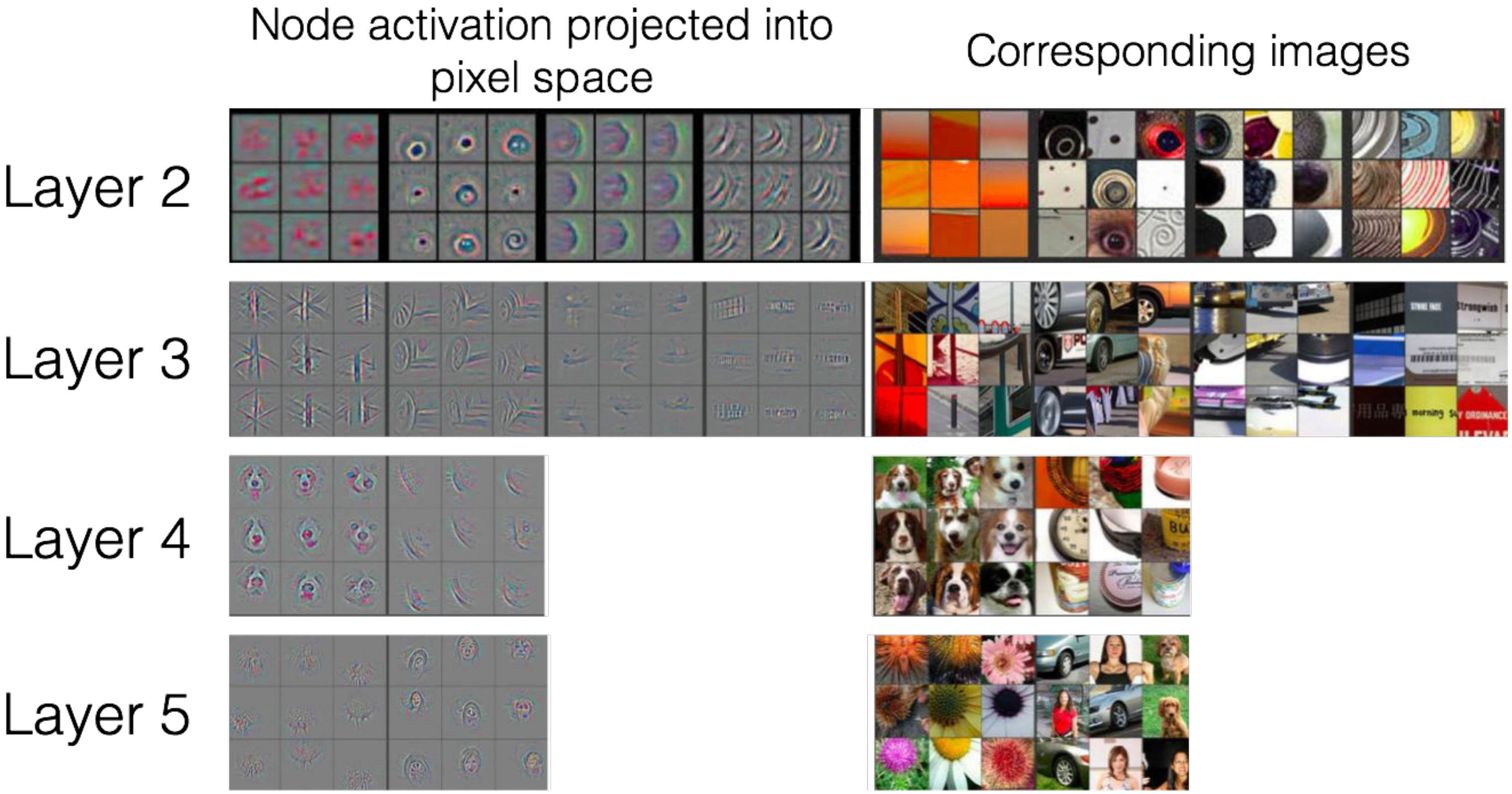


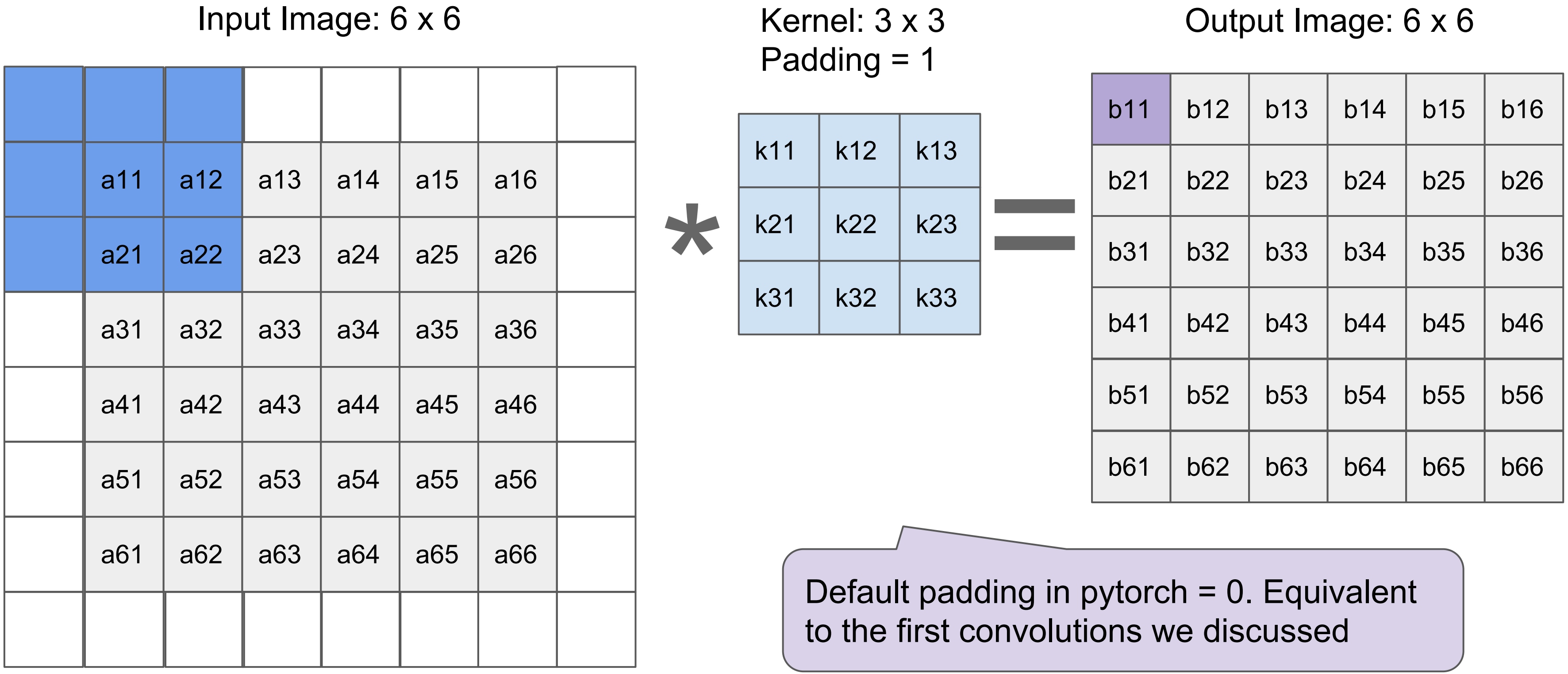
Image Source: https://favim.com/image/128916/

Features start simple and become more complex



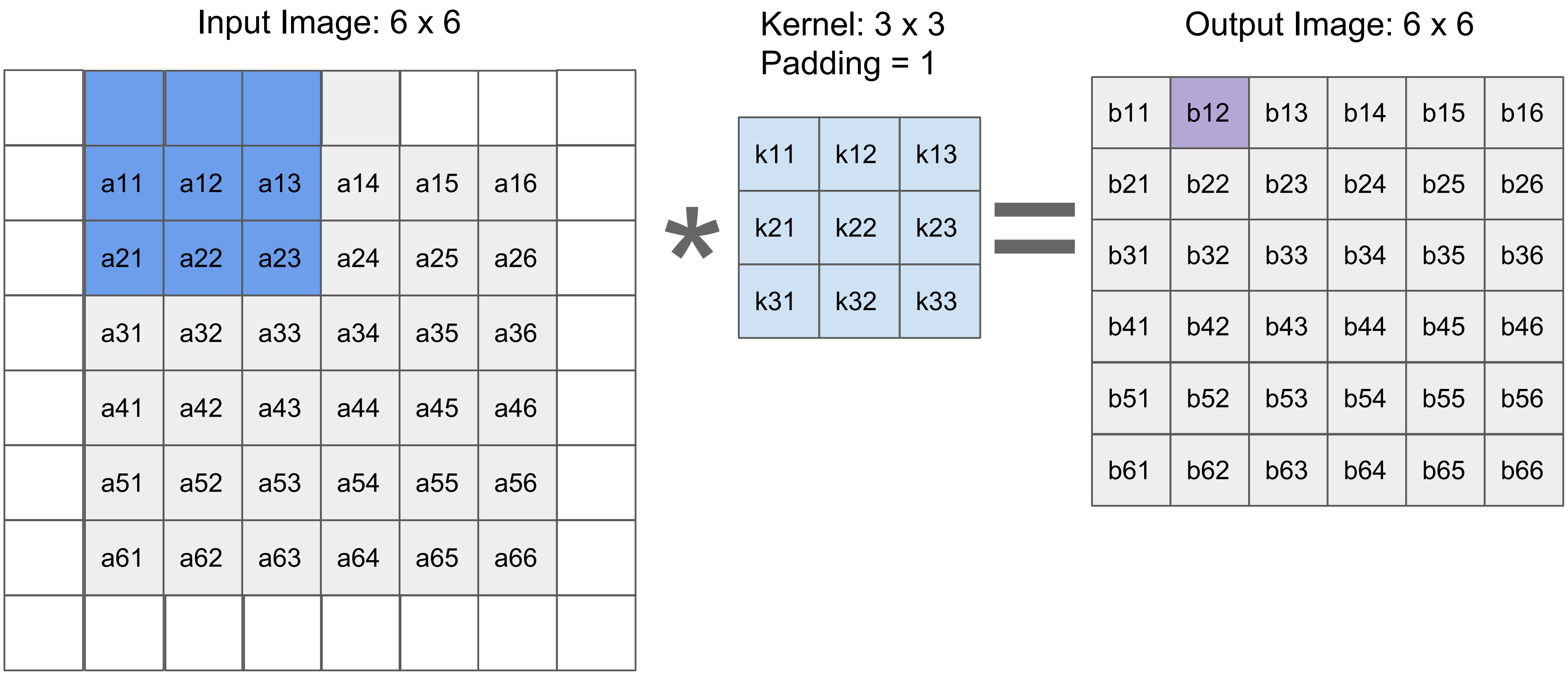
Source: Visualizing and Understanding Convolutional Networks, Zeiler & Fergus, 2014

Kernel options: Padding

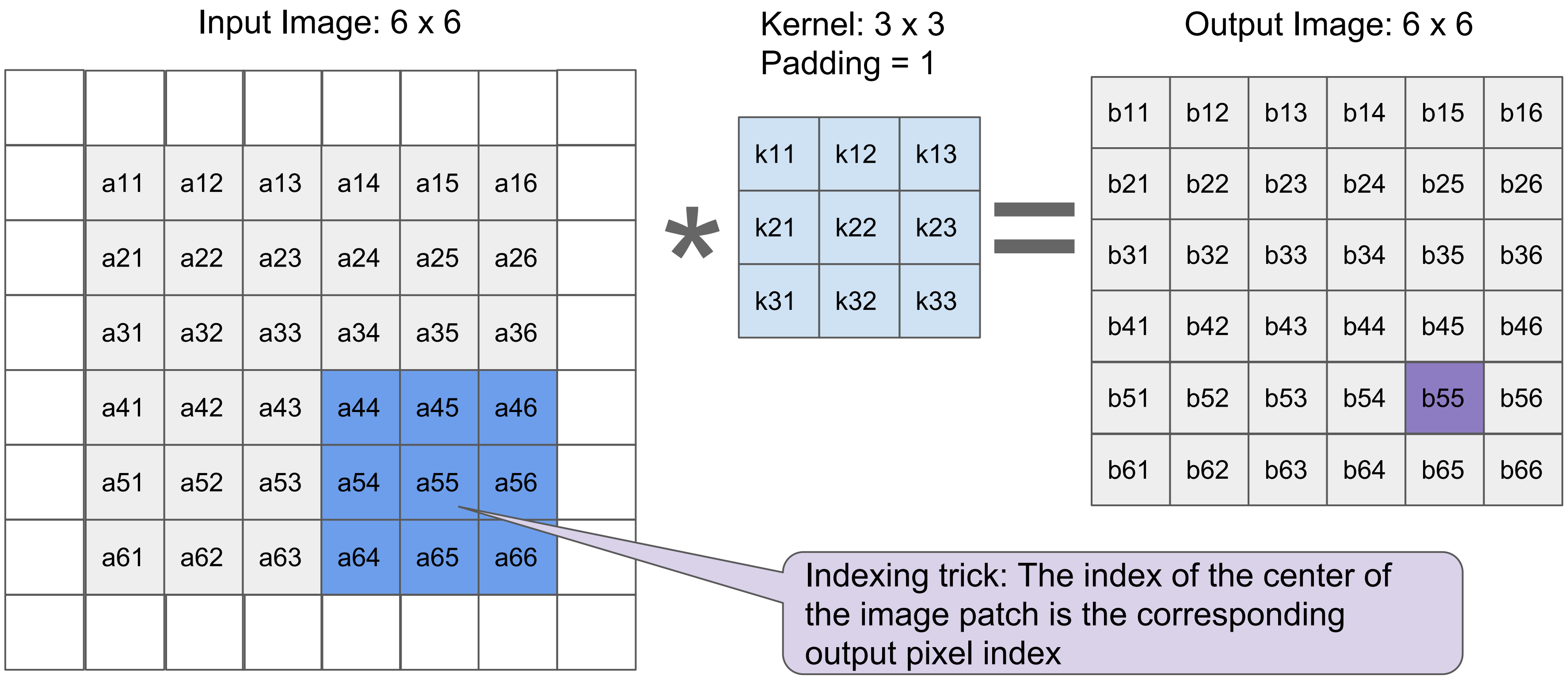


Default padding in pytorch = 0. Equivalent to the first convolutions we discussed

Kernel options: Padding



Kernel options: Padding



Kernel options: Stride

Input Image: 6 x 6

a11	a12	a13	a14	a15	a16
a21	a22	a23	a24	a25	a26
a31	a32	a33	a34	a35	a36
a41	a42	a43	a44	a45	a46
a51	a52	a53	a54	a55	a56
a61	a62	a63	a64	a65	a66

*

Kernel: 3 x 3
Stride = 2

k11	k12	k13
k21	k22	k23
k31	k32	k33

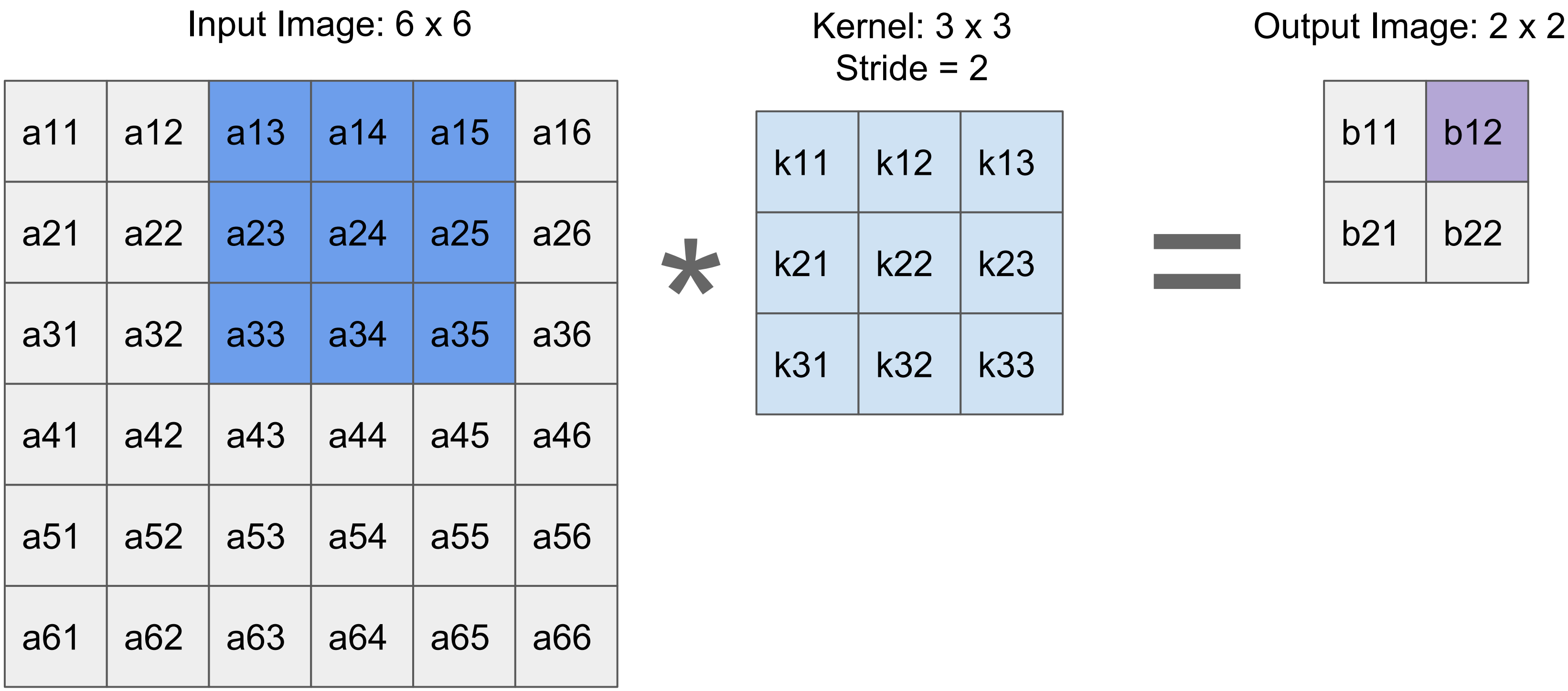
=

Output Image: 2 x 2

b11	b12
b21	b22

Default stride in pytorch = 1. Equivalent to the first convolutions we discussed

Kernel options: Stride



Kernel options: Stride

Input Image: 6 x 6

a11	a12	a13	a14	a15	a16
a21	a22	a23	a24	a25	a26
a31	a32	a33	a34	a35	a36
a41	a42	a43	a44	a45	a46
a51	a52	a53	a54	a55	a56
a61	a62	a63	a64	a65	a66

*

Kernel: 3 x 3
Stride = 2

k11	k12	k13
k21	k22	k23
k31	k32	k33

=

Output Image: 2 x 2

b11	b12
b21	b22

Kernel options: Dilation

Input Image: 6 x 6

a11	a12	a13	a14	a15	a16
a21	a22	a23	a24	a25	a26
a31	a32	a33	a34	a35	a36
a41	a42	a43	a44	a45	a46
a51	a52	a53	a54	a55	a56
a61	a62	a63	a64	a65	a66

*

Kernel: 3 x 3
Dilation = 2

k11	k12	k13
k21	k22	k23
k31	k32	k33

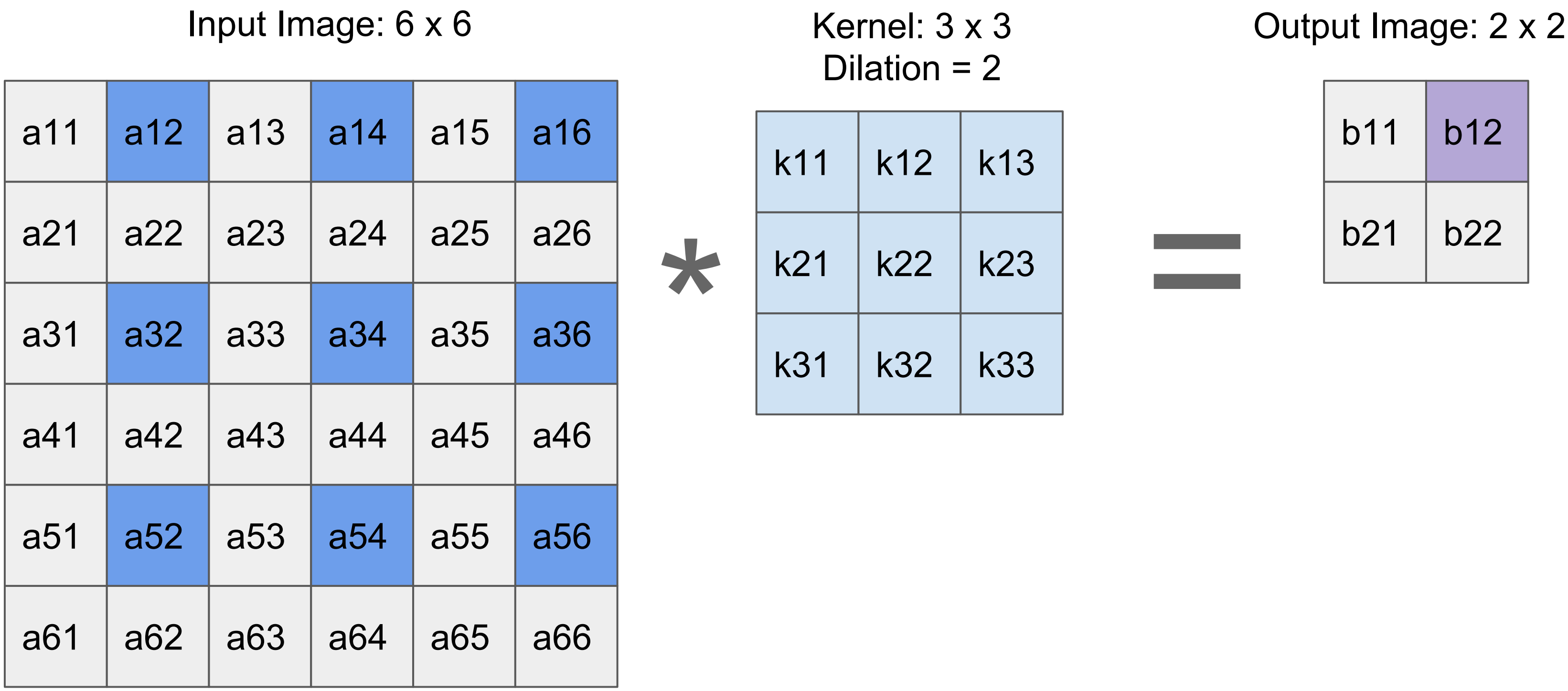
=

Output Image: 2 x 2

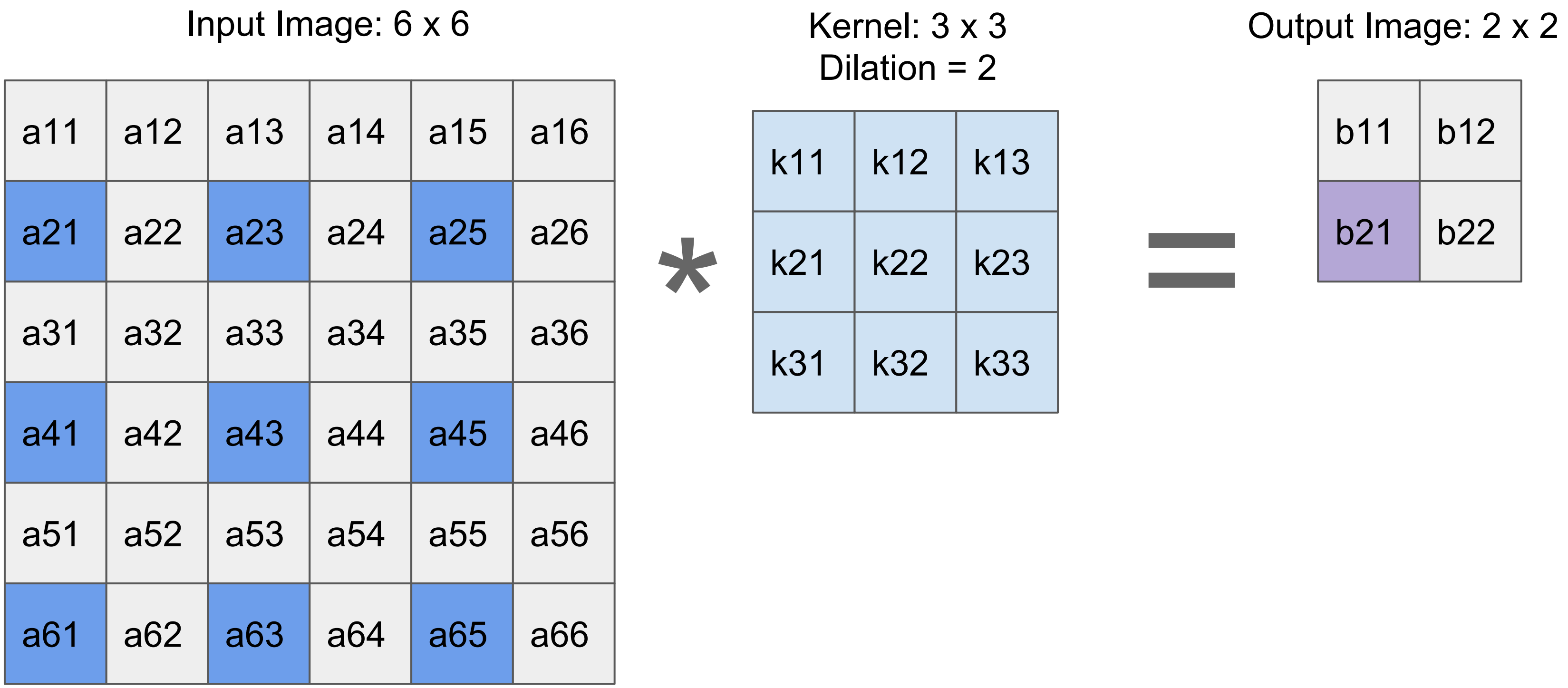
b11	b12
b21	b22

Default dilation in pytorch = 1. Equivalent to the first convolutions we discussed

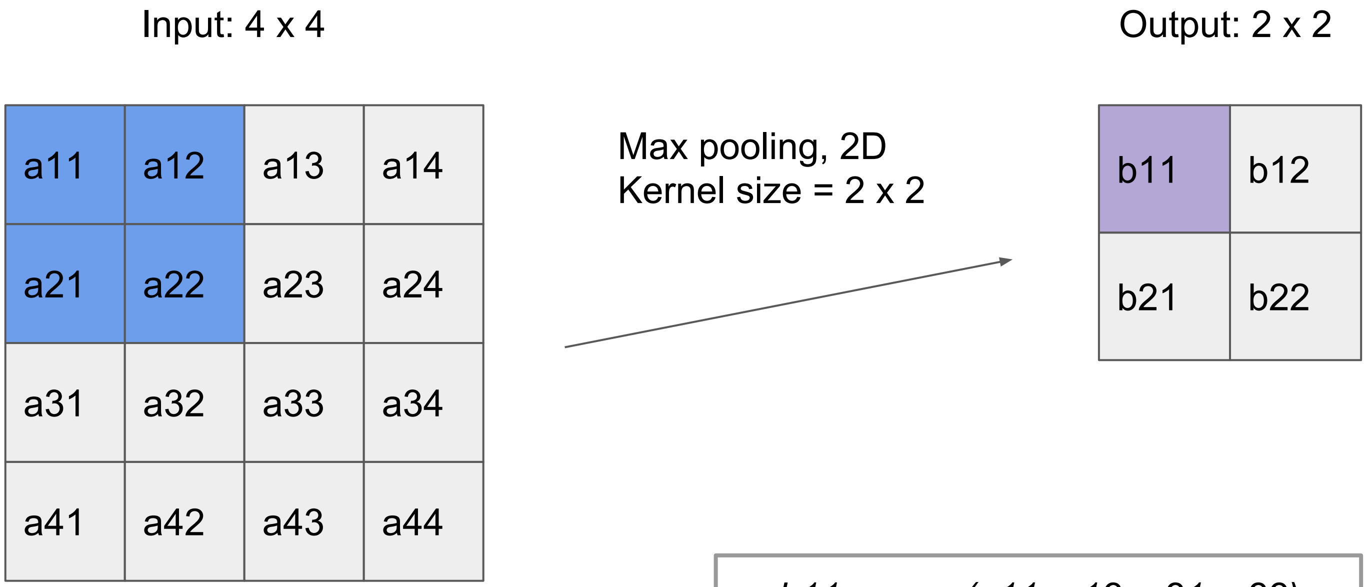
Kernel options: Dilation



Kernel options: Dilation

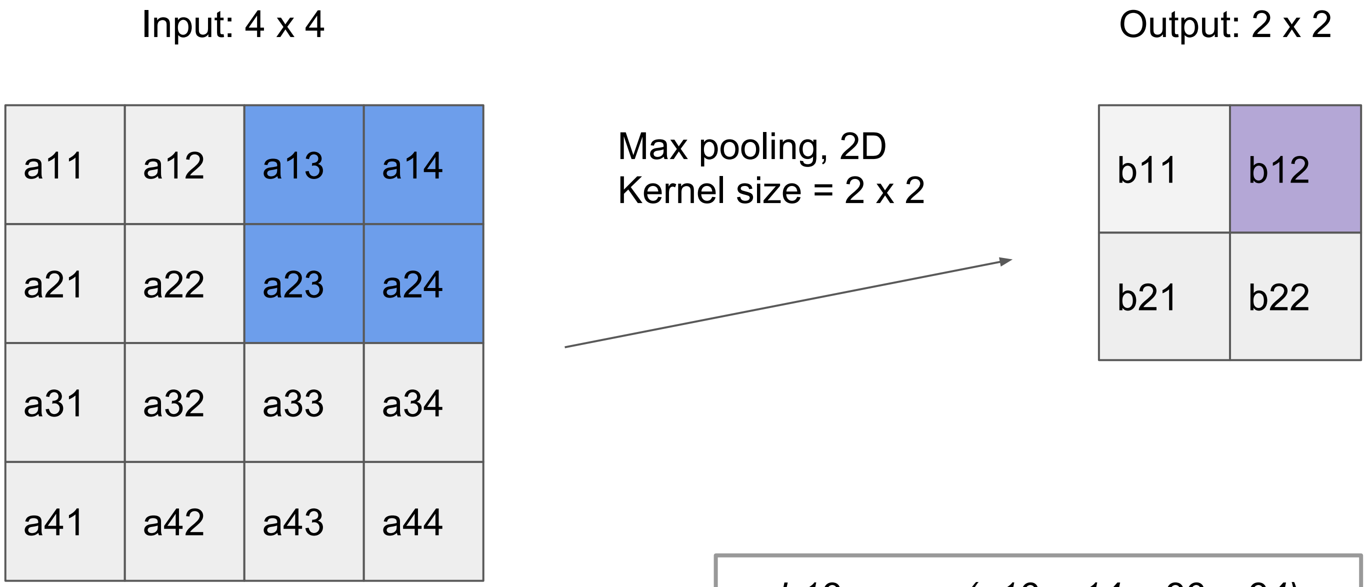


Pooling layer



$$b11 = max(a11, a12, a21, a22)$$

Pooling layer



b12 = max(a13, a14, a23, a24)

Convolution summary

- Similar to a dot product but with some structure
- Applied on a subset (patch) of an image
- Applied repeatedly
- Multiple kernels make up a convolutional layer
- Convolutional layer options
    - Number of kernels (feature detectors), often referred to as filters
    - Kernel size
    - Stride
    - Padding
    - Dilation
- Pooling layers can be used to downsample output

Simple convolutional network in pyTorch

Code available in CNN-Tutorial/model.py

```
Sequential (
  (0): Conv2d(3, 16, kernel_size=(5, 5), stride=(2, 2))
  (1): ReLU ()
  (2): Conv2d(16, 32, kernel_size=(5, 5), stride=(2, 2))
  (3): ReLU ()
  (4): Dropout2d (p=0.3)
)
Sequential (
  (0): Linear (800 -> 10)
)
```

Simple convolutional network in pyTorch

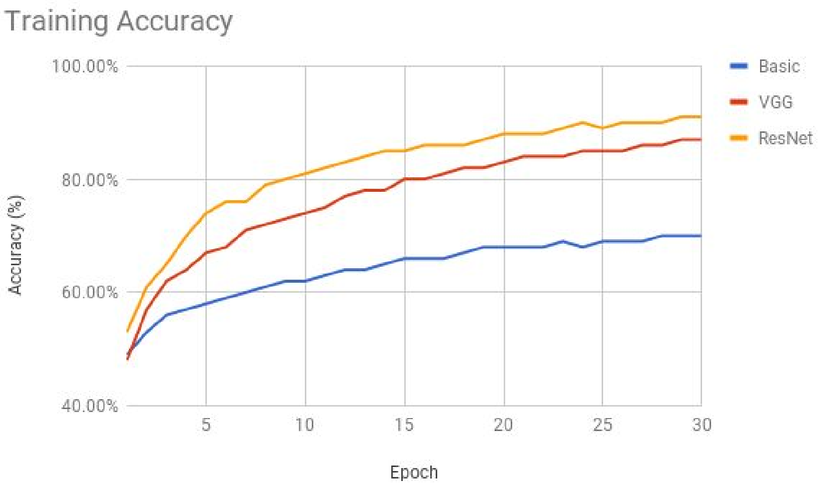
```
class CNN(nn.Module):
    '''Simple convolutional neural network, with 2 conv layers and one fully connected layer'''
    def __init__(self, dropout, nclasses):
        super(CNN, self).__init__()
        layers = []
        layers += [nn.Conv2d(in_channels=3, out_channels=16, kernel_size=5, stride=2, padding=0, dilation=1)]
        layers += [nn.ReLU()]
        layers += [nn.Conv2d(in_channels=16, out_channels=32, kernel_size=5, stride=2, padding=0, dilation=1)]
        layers += [nn.ReLU()]
        layers += [nn.Dropout2d(dropout)]
        self.conv_model = nn.Sequential(*layers)
        layers = []
        layers += [nn.Linear(in_features=800, out_features=nclasses)]
        self.flat_model = nn.Sequential(*layers)
        self.params = list(self.conv_model.parameters()) + list(self.flat_model.parameters())
        init_layers(self.params, 'Conv')
        init_layers(self.params, 'Linear')

    def forward(self, x):
        x = self.conv_model(x)
        x = x.view(-1, 800)
        x = self.flat_model(x)
        return x
```

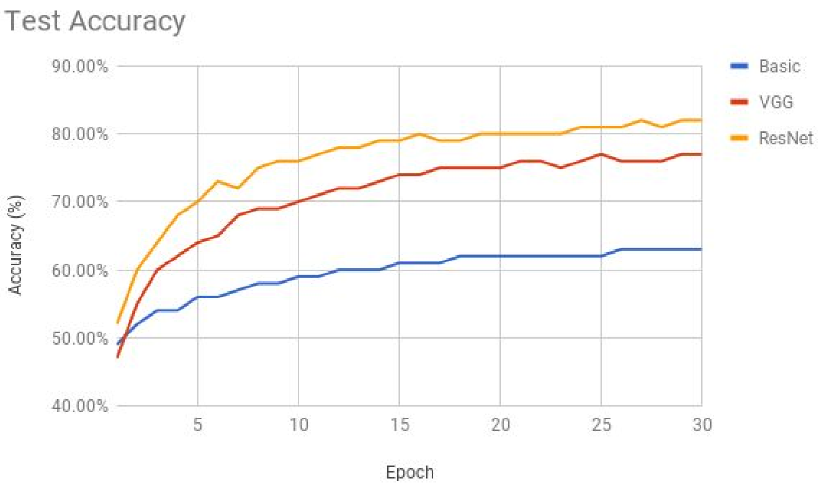
Only need to specify one dimension for square kernels.

pyTorch automatically infers how deep the kernel needs to be using number of in_channels

Some network designs are more powerful than others

Classification accuracy on CIFAR 10





VGG-net

- Developed by Simonyan and Zisserman in the Visual Geometry Group at Oxford in 2014
- Small kernels, many layers (for 2014)
- Simple architecture, repeated blocks
- RELU activation
- Won ImageNet challenge in 2014

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Note: RELU activations omitted from table for simplicity. Table source: Very Deep Convolutional Networks for Large Scale Image Recognition, Simonyan & Zisserman, 2014 https://arxiv.org/pdf/1409.1556.pdf

Residual Network

- Developed by He, Zhang, Ren and Sun at Microsoft Research in 2015
- Composed of many residual blocks
- Each block computes
  - H(x) = σ(x + F(x))
- Significantly helps with the vanishing / exploding gradient problem
- Makes it possible to train ***very*** deep (100+ layers) networks
- Make learning easier by introducing shortcut connections

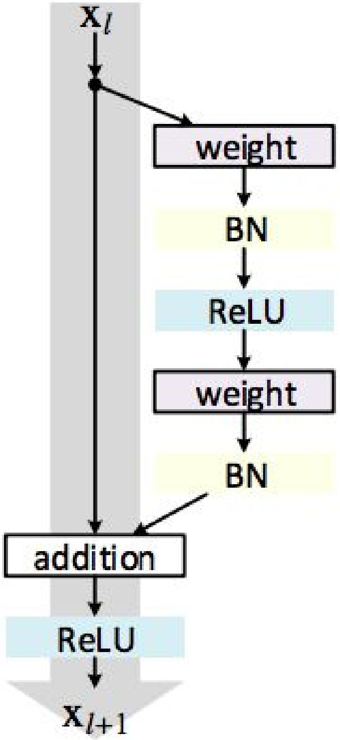


Image source: Identity Mappings in Deep Residual Networks, He. et. al, 2016 via. Professor Rob Fergus' lecture slides.
https://cs.nyu.edu/~fergus/teaching/vision/8_resnets.pdf

Residual Block

Implementation available in CNN-Tutorial/model.py

```
class ResidualBlock(nn.Module):
    def __init__(self, filt):
        super(ResidualBlock, self).__init__()
        model = []
        model += [nn.Conv2d(filt, filt, kernel_size=3, stride=1, padding=1)]
        model += [nn.BatchNorm2d(filt)]
        model += [nn.ReLU(inplace=True)]
        model += [nn.Conv2d(filt, filt, kernel_size=3, stride=1, padding=1)]
        model += [nn.BatchNorm2d(filt)]
        self.model = nn.Sequential(*model)

    def forward(self, x):
        out = self.model(x)
        out += x
        return F.relu(out)
```

Residual Network

```python
class ResidualNet(nn.Module):
    def __init__(self, in_dim, channels, filt, num_res_blocks, drop_p, nclasses):
        super(ResidualNet, self).__init__()
        self.in_dim = in_dim
        model = []
        # Large kernel and stride of 2 so that image shrunk to filt * 14 * 14 dim
        model += [nn.Conv2d(channels, filt, kernel_size=5, stride=2)]
        model += [nn.BatchNorm2d(filt)]
        model += [nn.ReLU(inplace=True)]
        model += [nn.Dropout(drop_p)]
        # Add residual blocks. Downsample size and double number of filters every 2 blocks
        for i in range(num_res_blocks):
            if (i + 1) % 2 == 0:
                model += [ResidualBlock(filt)]
                model += [nn.Conv2d(filt, filt * 2,
                                    kernel_size=3, stride=2, padding=1)]
                filt = filt * 2
                model += [nn.BatchNorm2d(filt)]
                model += [nn.ReLU(inplace=True)]
                model += [nn.Dropout(drop_p)]
            else:
                model += [ResidualBlock(filt)]
                model += [nn.Dropout(drop_p)]
        self.res_blocks = nn.Sequential(*model)
        self.flat_weights = self.get_conv_output_size()
        print("Number of flat weights: {}".format(self.flat_weights))
        self.fc1 = nn.Linear(self.flat_weights, nclasses)
        self.params = list(self.res_blocks.parameters())
        # Initialize model parameters
        init_layers(self.params, 'Linear')
        init_layers(self.params, 'Conv')
        init_layers(self.params, 'BatchNorm')
        torch.nn.init.xavier_uniform(self.fc1.weight.data)
        self.fc1.bias.data.fill_(0.01)
```

```python
def get_conv_output_size(self):
    x = Variable(torch.ones(1, *self.in_dim))
    x = self.res_blocks(x)
    return x.numel()

def forward(self, x):
    x = self.res_blocks(x)
    x = x.view(-1, self.flat_weights)
    x = self.fc1(x)
    return x
```

What next?

- Try out these models yourself using the CNN-Tutorial repo
    - https://github.com/lgraesser/CNN-Tutorial
    - Change the hyperparameters and model structure using the command line argument
- Modify the implementations provided to create new models
- Read the VGG and Residual network papers
    - Very Deep Convolutional Networks for Large-Scale Image Recognition, Simonyan & Zisserman, 2014
    - Deep Residual Learning for Image Recognition, He et. al, 2015
- Further reading
    - Generalization and Network Design Strategies, LeCun, 1989
    - Neural Networks and Neuroscience-Inspired Computer Vision, Dean & Cox, 2014

Thank You & Questions?