USC Viterbi School of Engineering

CONVOLUTIONAL NEURAL NETWORKS

EE 541 – UNIT 8

DR. BRANDON FRANZKE

Fall 2025

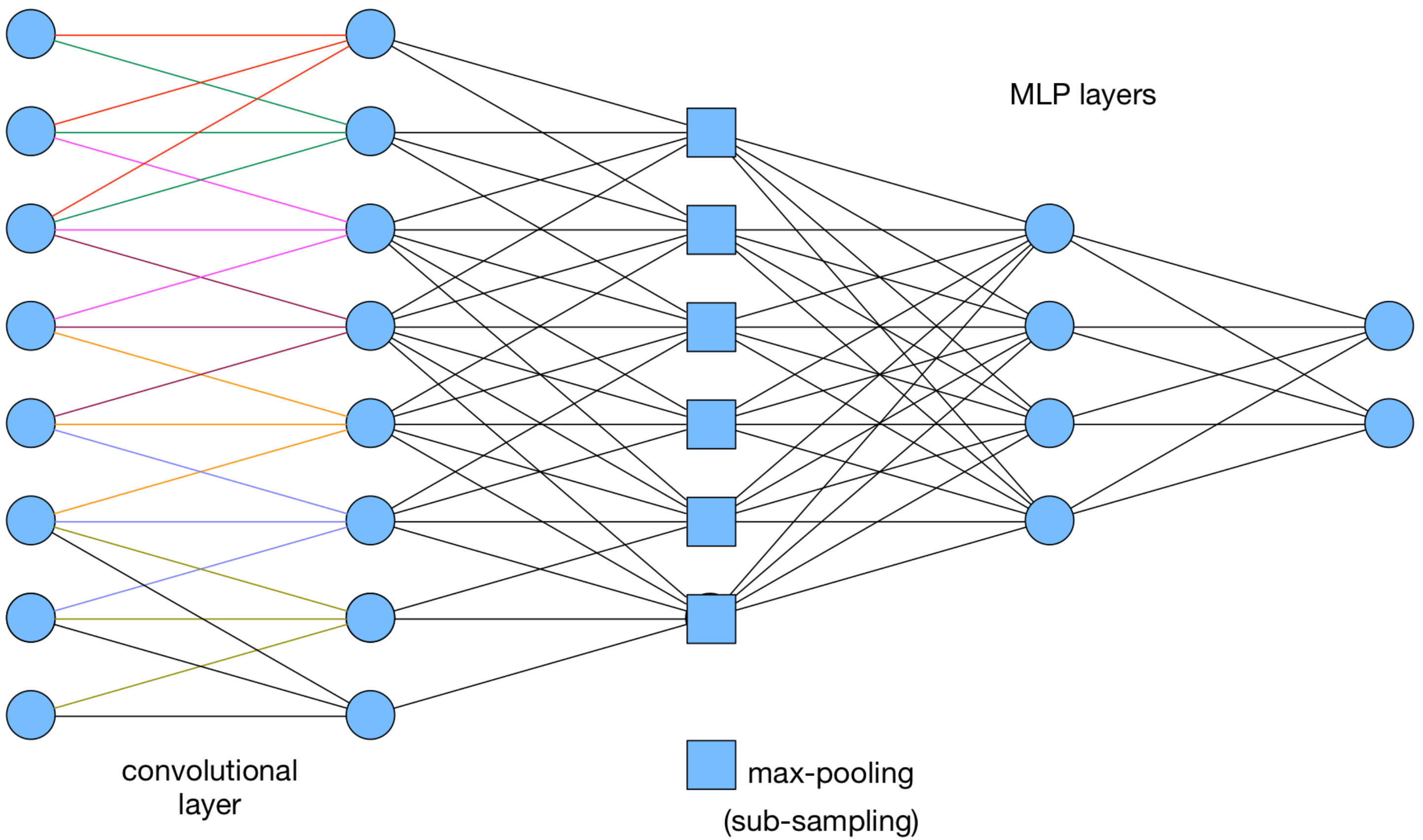
OUTLINE FOR SLIDES

- Motivation, applications
- Basic 2D convolution operations
  - PyTorch 2Dconv layer
- Pooling and stride
- Fashion MNIST example
- Visualization methods
- Some common CNN structures
- Reduced complexity CNN architectures
- Outline of Back-propagation for CNNs

CONVNETS

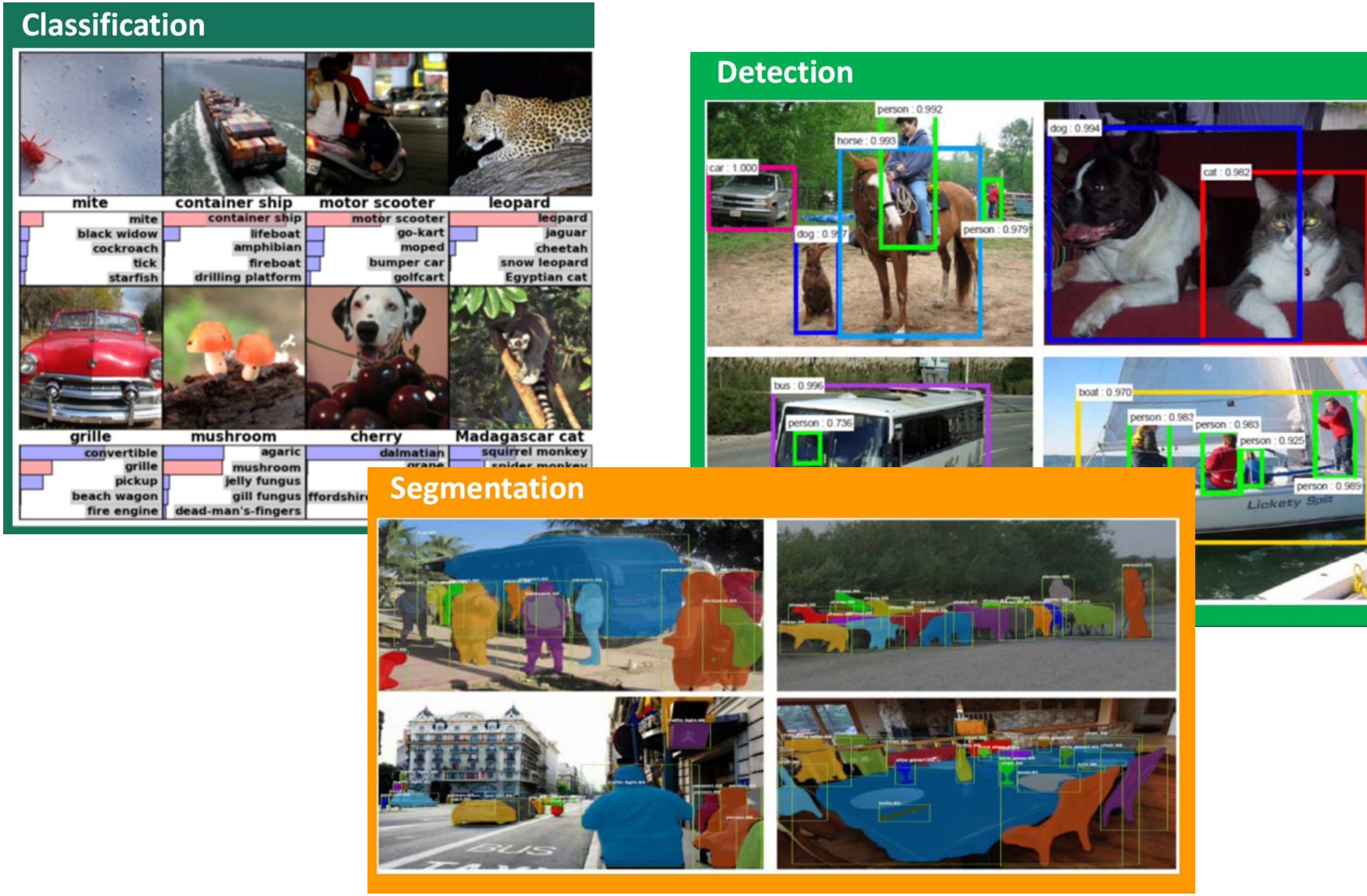
(TYPES OF NEURAL NETWORKS)

Convolutional NNets

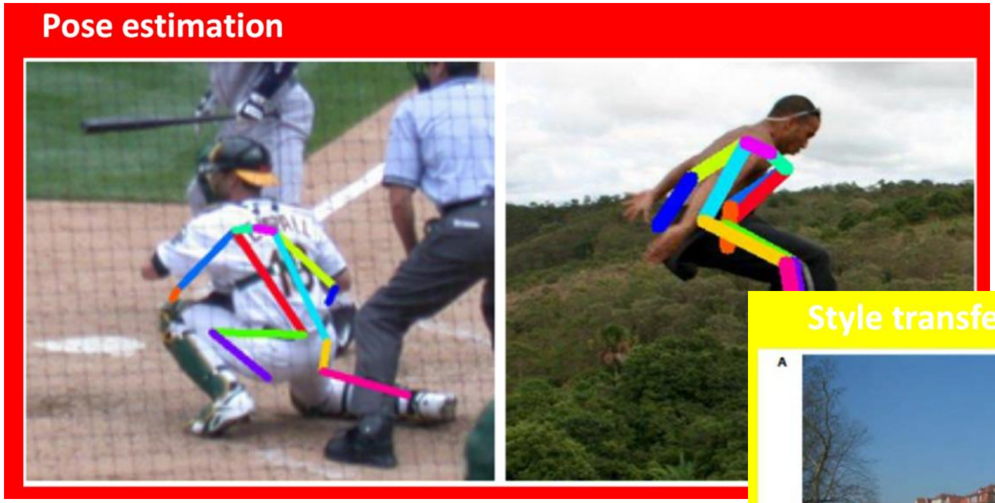


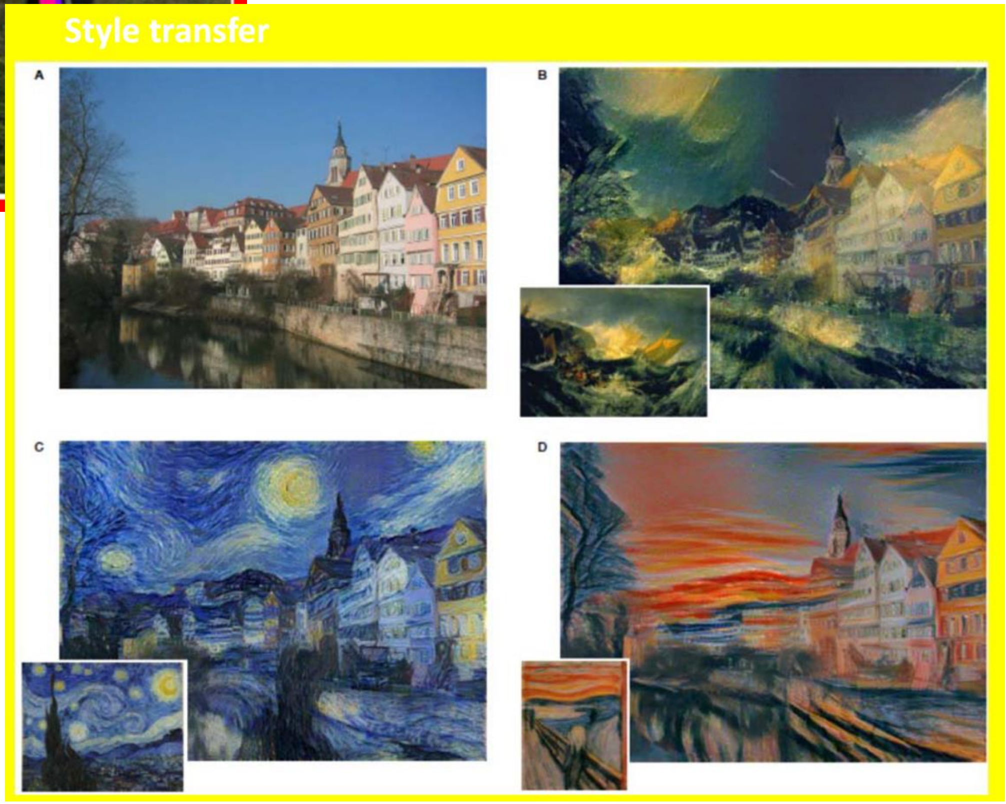
Can view convolutions as feature extractors for MLP classifier
(this feature extraction is learned)

CNNS ARE WIDELY USED, ESPECIALLY IN VISION TASKS

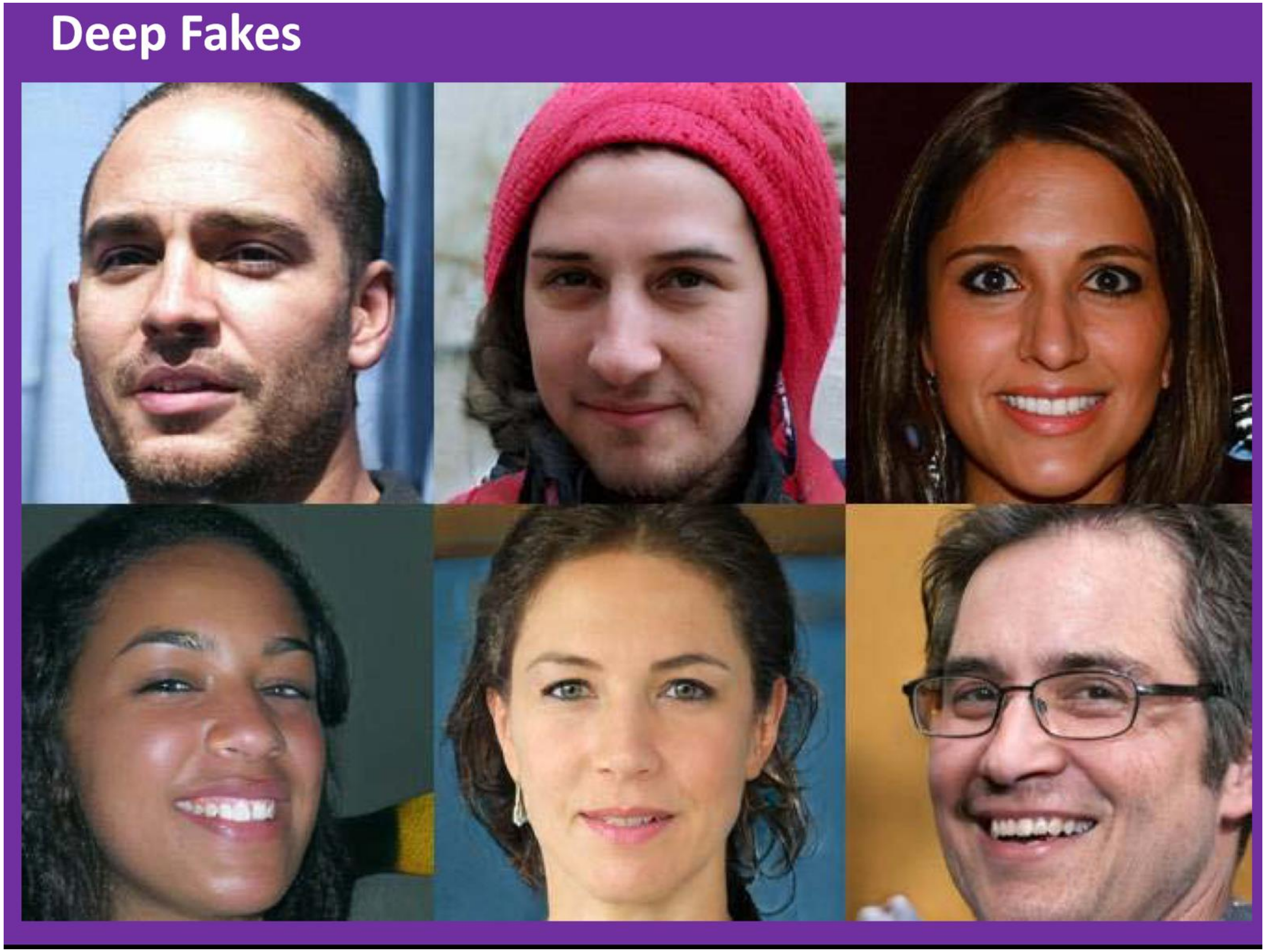


CNNS ARE WIDELY USED, ESPECIALLY IN VISION TASKS





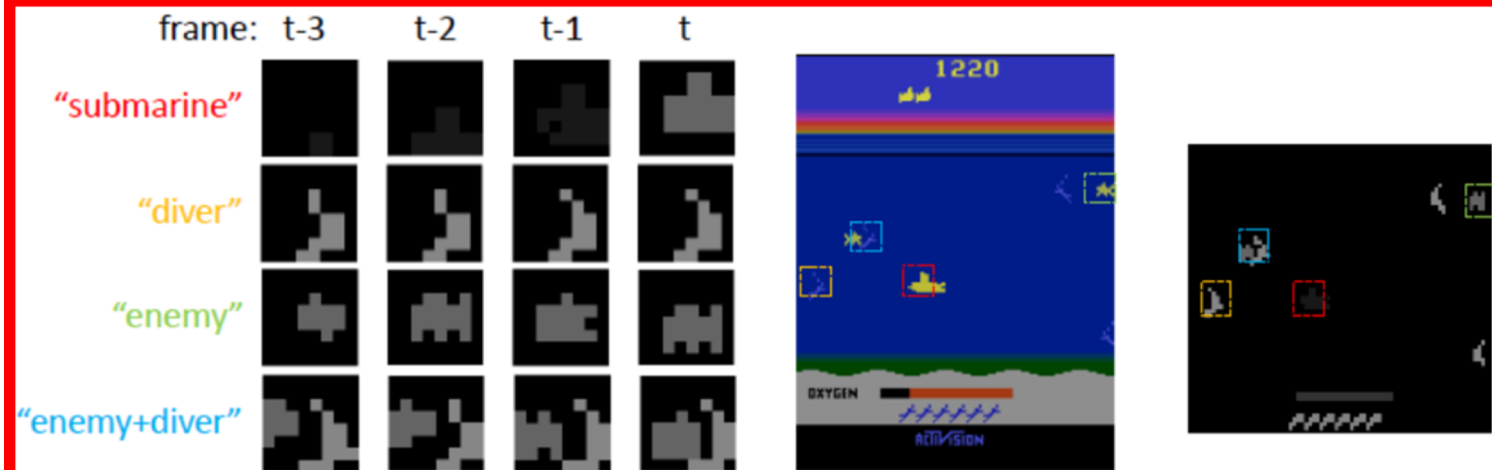
CNNS ARE WIDELY USED, ESPECIALLY IN VISION TASKS



https://thispersondoesnotexist.com https://thisxdoesnotexist.com/

CNNS: USE WHEN FEATURE INFORMATION IS LOCALIZED

Policy selection



Captioning

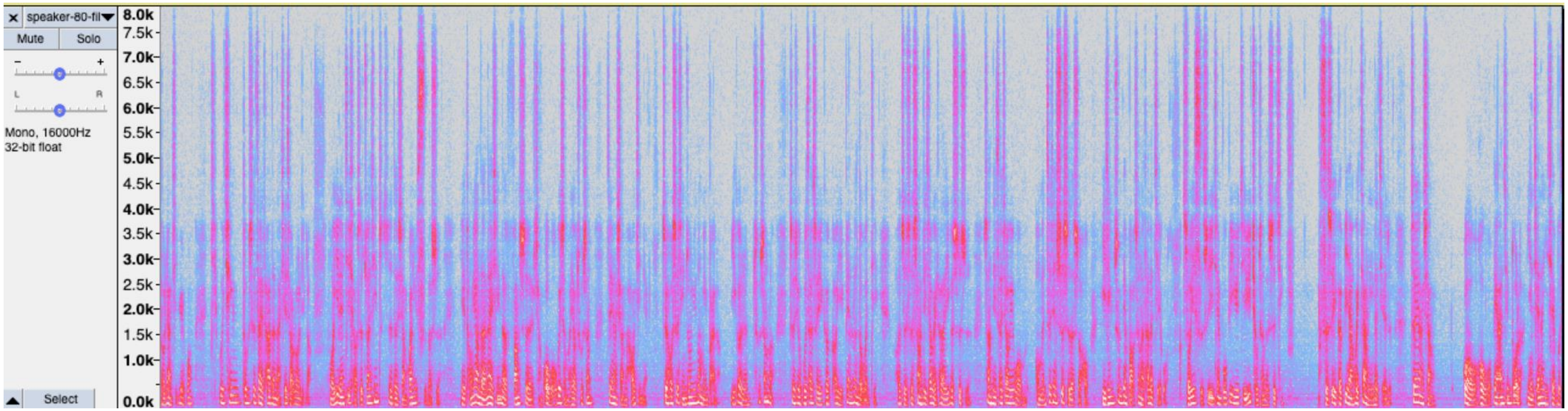

a train is traveling down the tracks at a train station


a cake with a slice cut out of it


a bench sitting on a patch of grass next to a sidewalk

CNNS: USE WHEN FEATURE INFORMATION IS LOCALIZED

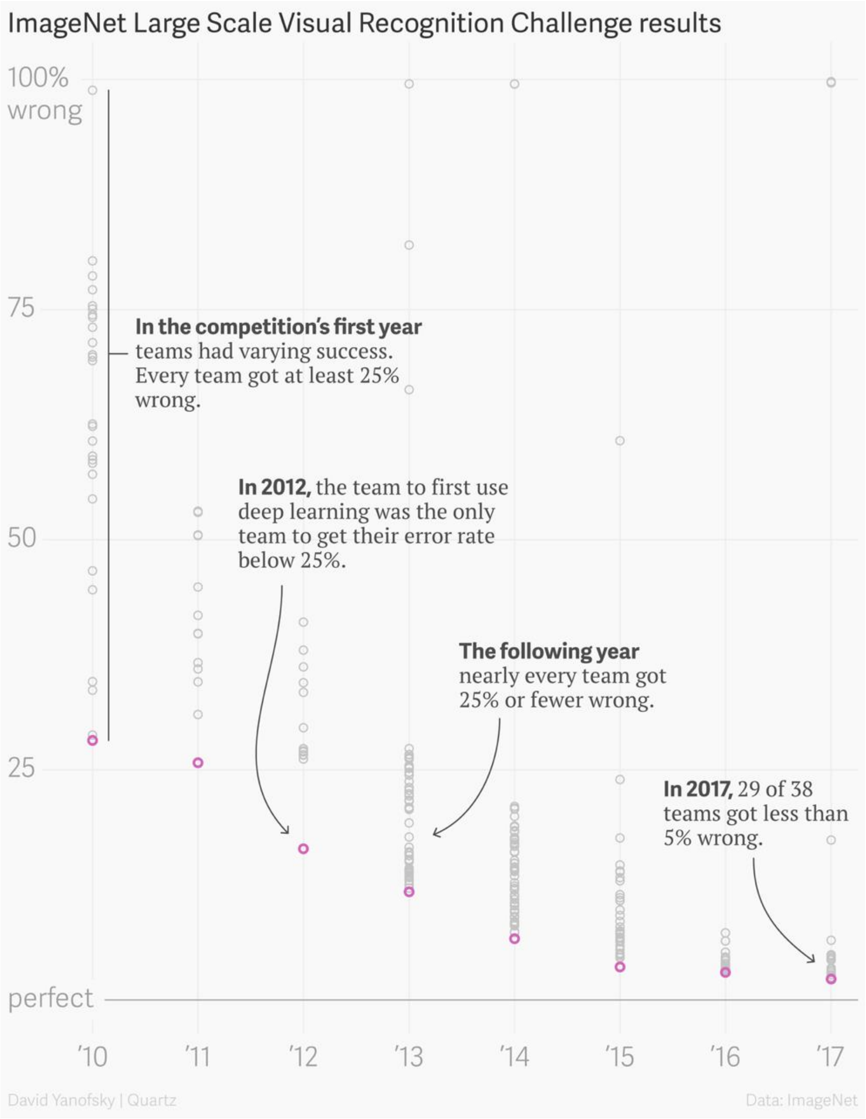
frequency



time

does not need to be a "natural" image —
e.g., signal classification from spectrograms

CNNS: CHANGING WHAT IS POSSIBLE WITH CV



CNNs *changed the game* for many computer vision tasks

The leap that transformed AI research—and possibly the world

CNNS: 1D, 2D, 3D

there are 1D and 3D convolutional layers, but conv2D is most widely used

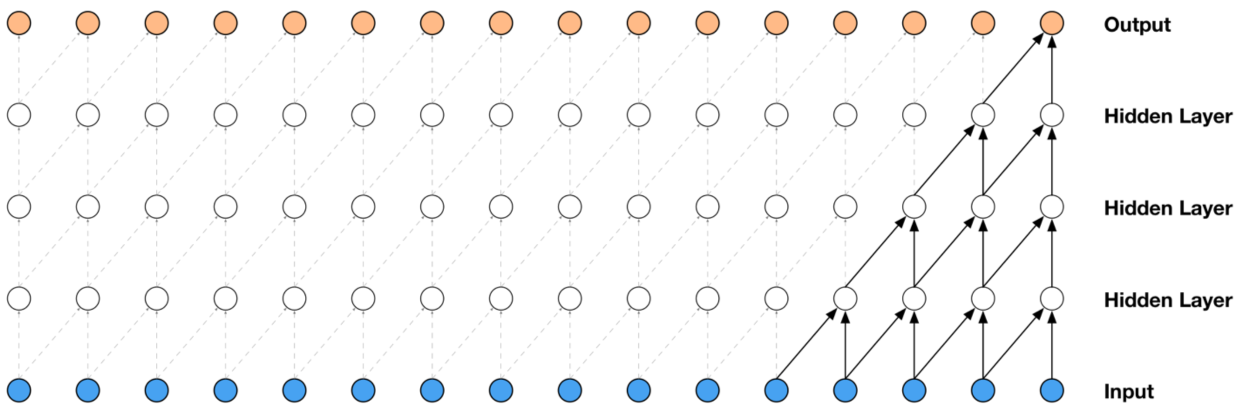


Figure 2: Visualization of a stack of causal convolutional layers.

1D Conv layers

1D CNN ~ time series data

3D CNN ~ video data

(recurrent networks are options too and can be combined with conv)

Oord, Aaron van den, et al. "Wavenet: A generative model for raw audio." arXiv preprint arXiv:1609.03499 (2016).

OUTLINE FOR SLIDES

- Motivation, applications
- Basic 2D convolution operations
    - PyTorch 2Dconv layer
- Pooling and stride
- Fashion MNIST example
- Visualization methods
- Some common CNN structures
- Reduced complexity CNN architectures
- Outline of Back-propagation for CNNs

2D CONVOLUTION

USC Viterbi School of Engineering

2D CONVOLUTION OPERATIONS

2D convolution:

$$y[i,j] = x[i,j] * h[i,j] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} x[m,n]h[i-m,j-n]$$
$$= \sum_{m=-L}^{L} \sum_{n=-L}^{L} x[m,n]h[i-m,j-n]$$

and 2D correlation:

$$y[i,j] = x[i,j] \star h[i,j] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} x[m,n]h[i+m,j+n]$$
$$= \sum_{m=-L}^{L} \sum_{n=-L}^{L} x[m,n]h[i+m,j+n]$$

Note: last expressions assume that $h[i,j]$ is zero for $|i| > L$, and $|j| > L$

2D CONVOLUTION OPERATIONS

Since we will be learning the 2D filter $h[i,j]$ we can adapt a correlation convention as "convolution"

typical notation and terminology in the deep learning literature

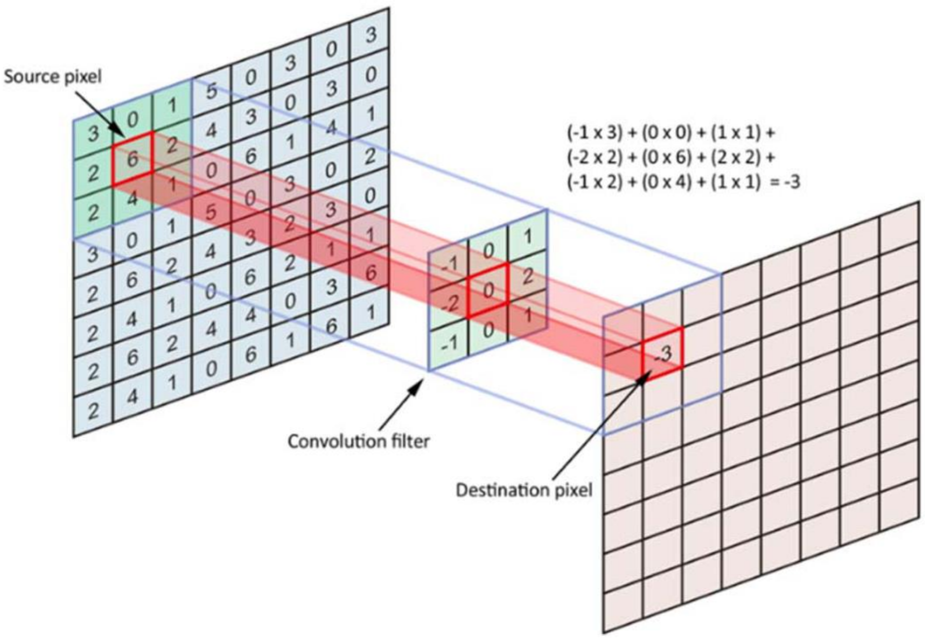
$$y[i,j] = x[i,j] \star K[i,j] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} K[m,n] x[i+m, j+n]$$

$K[i,j]$ ~ (2D) Filter kernel
"y is x convolved with K"

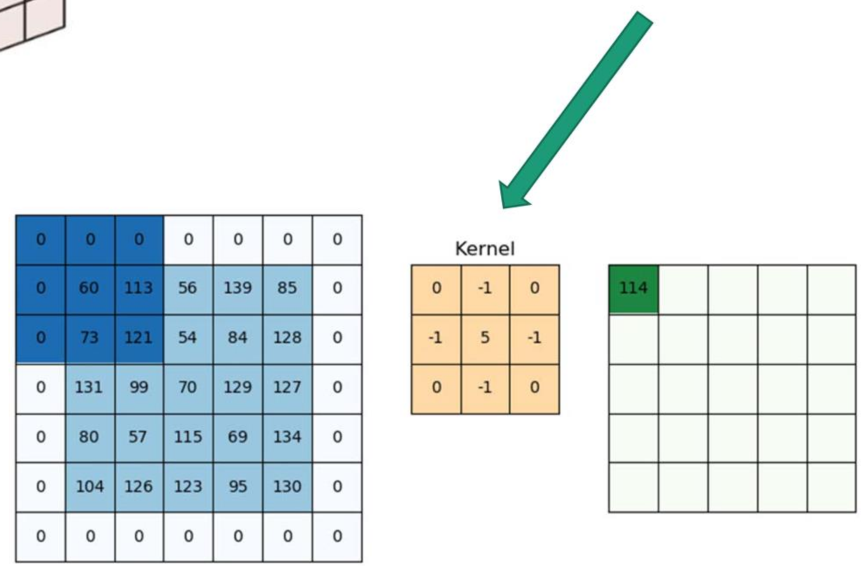
$$y[i,j] = x[i,j] \star K[i,j] = \sum_{(m,n) \in \mathrm{supp}(K)} K[m,n] x[i+m, j+n]$$

typically, the support region of the kernel is small — *e.g.*, 3x3 kernels are very common

2D CONVOLUTION OPERATIONS

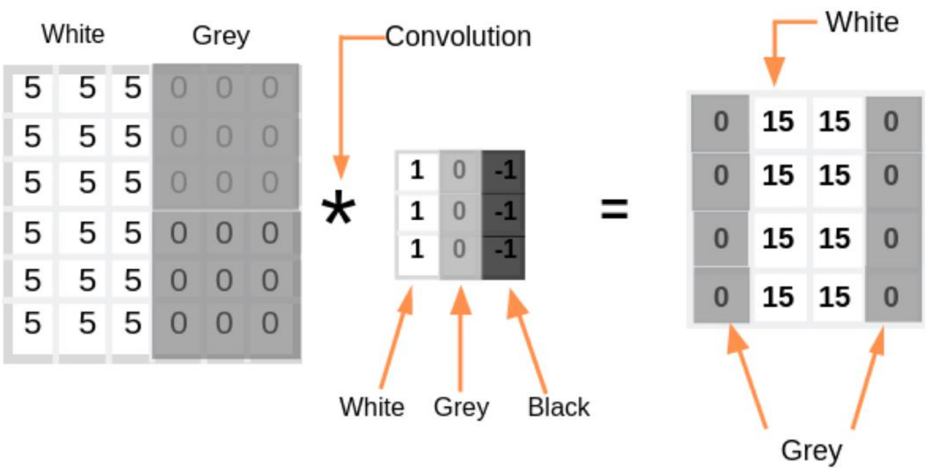


This is what you learn!



TRADITIONAL 2D IMAGE FILTERS

2D filters are widely used in the field of image processing



example: edge detection filter

many computer vision tasks require many types filters to produce features

**CNNs learn these filters from the dataset —
learn good feature extraction**

2D CONVOLUTION OPERATIONS — PADDING

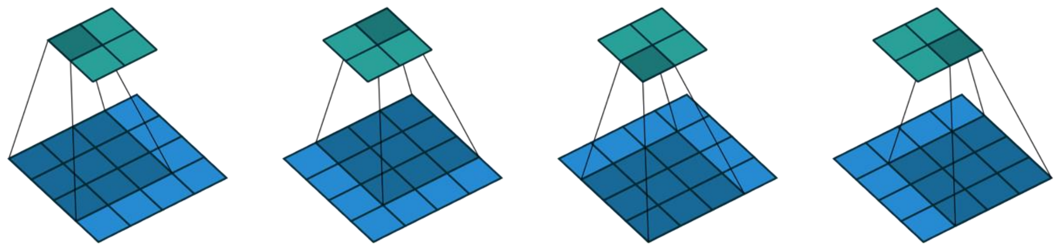


Figure 2.1: (No padding, no strides) Convolving a 3×3 kernel over a 4×4 input using unit strides (i.e., $i = 4$, $k = 3$, $s = 1$ and $p = 0$).

no padding
empty padding in PyTorch

output will be
smaller than input

here, 4x4 → 2x2

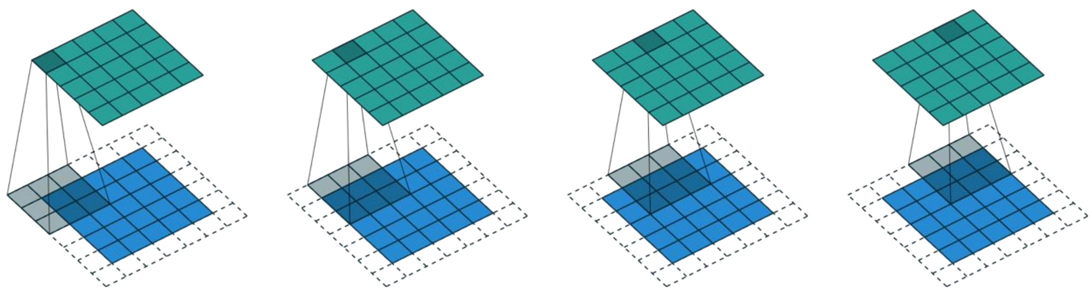


Figure 2.3: (Half padding, no strides) Convolving a 3×3 kernel over a 5×5 input using half padding and unit strides (i.e., $i = 5$, $k = 3$, $s = 1$ and $p = 1$).

symmetric padding
padding:[1 | [1,1]] in PyTorch

output will be
same size as input

here, 5x5 → 5x5

Dumoulin, Vincent, and Francesco Visin. "A guide to convolution arithmetic for deep learning." arXiv preprint arXiv:1603.07285 (2016).

2D CONVOLUTION OPERATIONS — PADDING

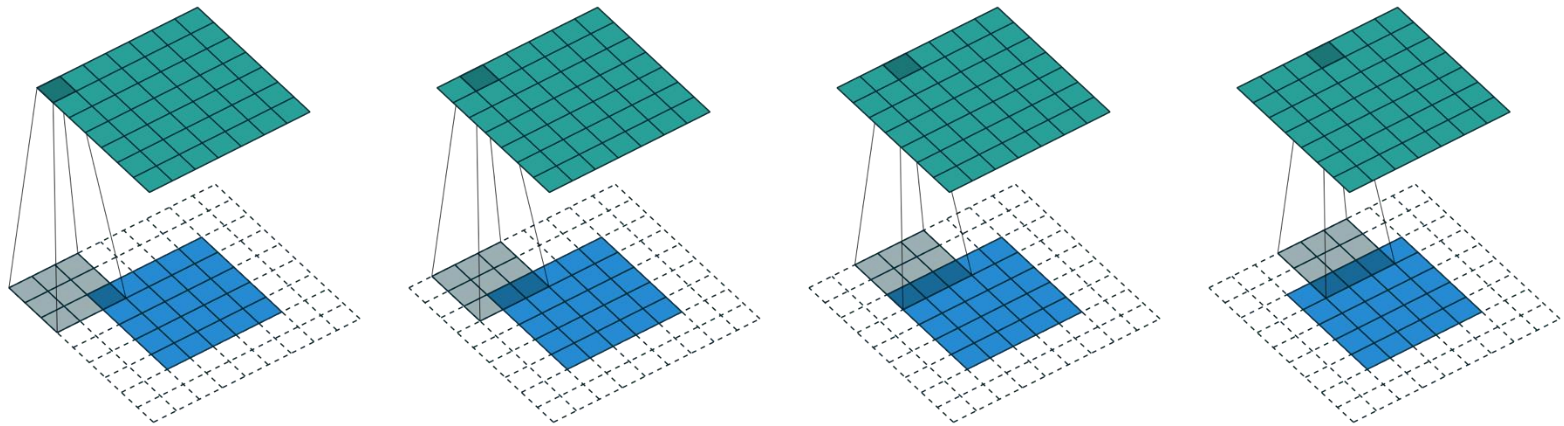


Figure 2.4: (Full padding, no strides) Convolving a 3×3 kernel over a 5×5 input using full padding and unit strides (i.e., $i = 5$, $k = 3$, $s = 1$ and $p = 2$).

other padding conventions exist —
e.g., "full padding"

output will be larger than input

here, 5x5 → 7x7

Dumoulin, Vincent, and Francesco Visin. "A guide to convolution arithmetic for deep learning." arXiv preprint arXiv:1603.07285 (2016).

CONVOLUTION OPERATIONS — PADDING WITH LAYERS

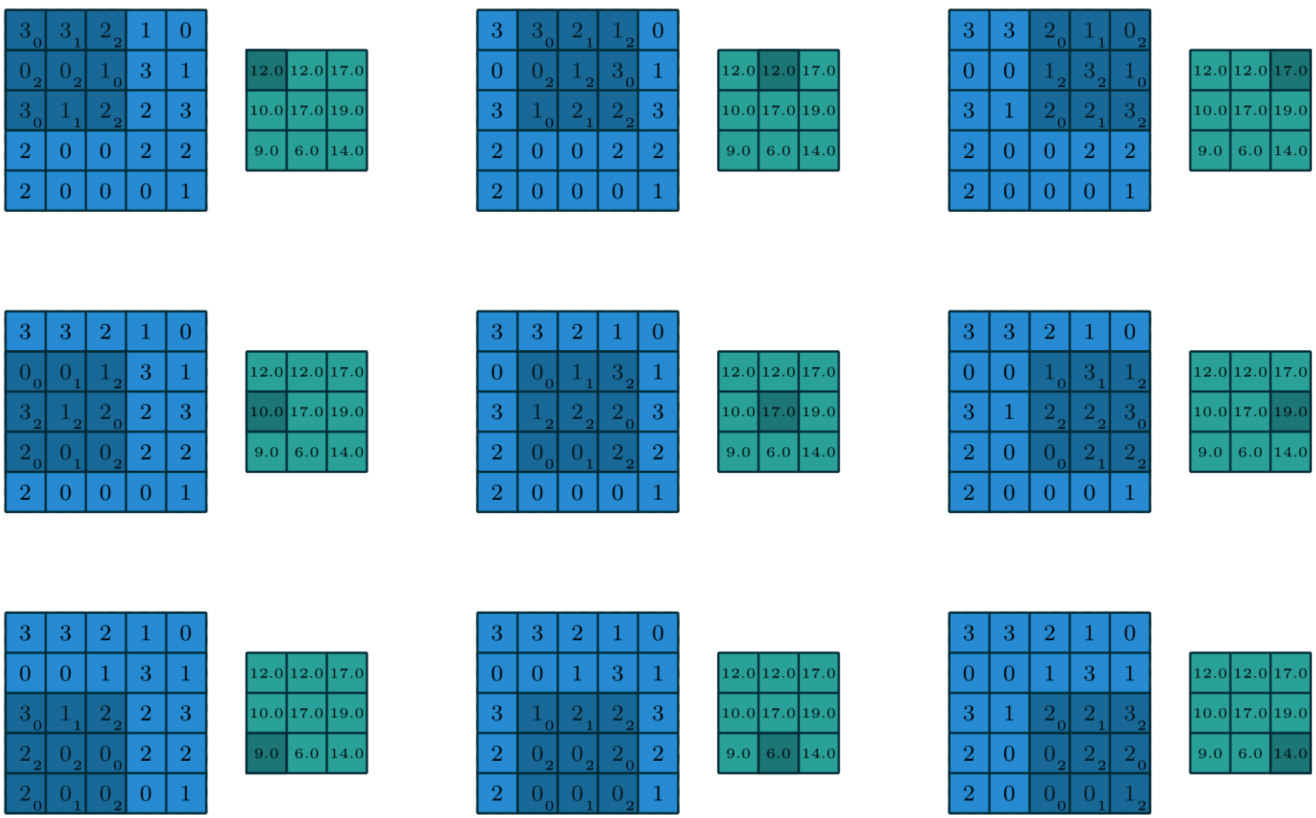
Padding Layers

`nn.ReflectionPad1d`	Pads the input tensor using the reflection of the input boundary.
`nn.ReflectionPad2d`	Pads the input tensor using the reflection of the input boundary.
`nn.ReplicationPad1d`	Pads the input tensor using replication of the input boundary.
`nn.ReplicationPad2d`	Pads the input tensor using replication of the input boundary.
`nn.ReplicationPad3d`	Pads the input tensor using replication of the input boundary.
`nn.ZeroPad2d`	Pads the input tensor boundaries with zero.
`nn.ConstantPad1d`	Pads the input tensor boundaries with a constant value.
`nn.ConstantPad2d`	Pads the input tensor boundaries with a constant value.
`nn.ConstantPad3d`	Pads the input tensor boundaries with a constant value.

- replication
- reflection
- zero
- constant

PyTorch padding layers provide greater control

2D CONVOLUTION OPERATIONS



kernel

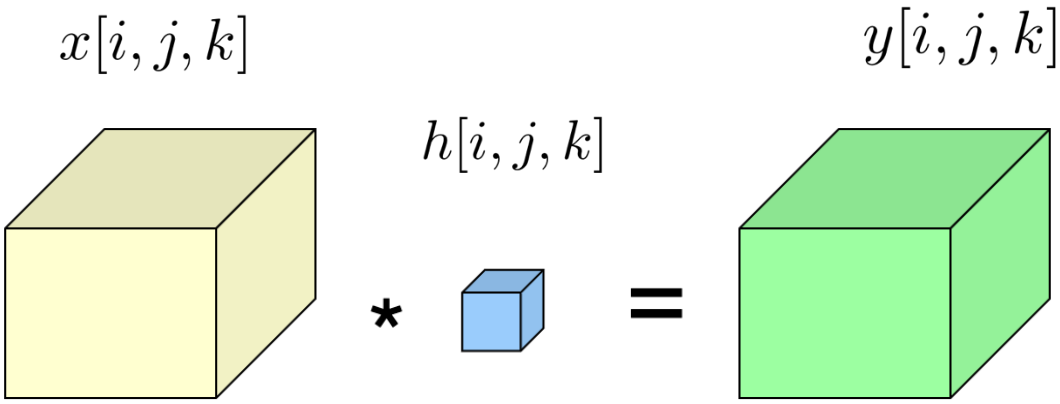
0	1	2
2	2	0
0	1	2

detailed example for **3x3 kernel** with **no padding** and 5x5 input

Dumoulin, Vincent, and Francesco Visin. "A guide to convolution arithmetic for deep learning." arXiv preprint arXiv:1603.07285 (2016).

3D CONVOLUTION

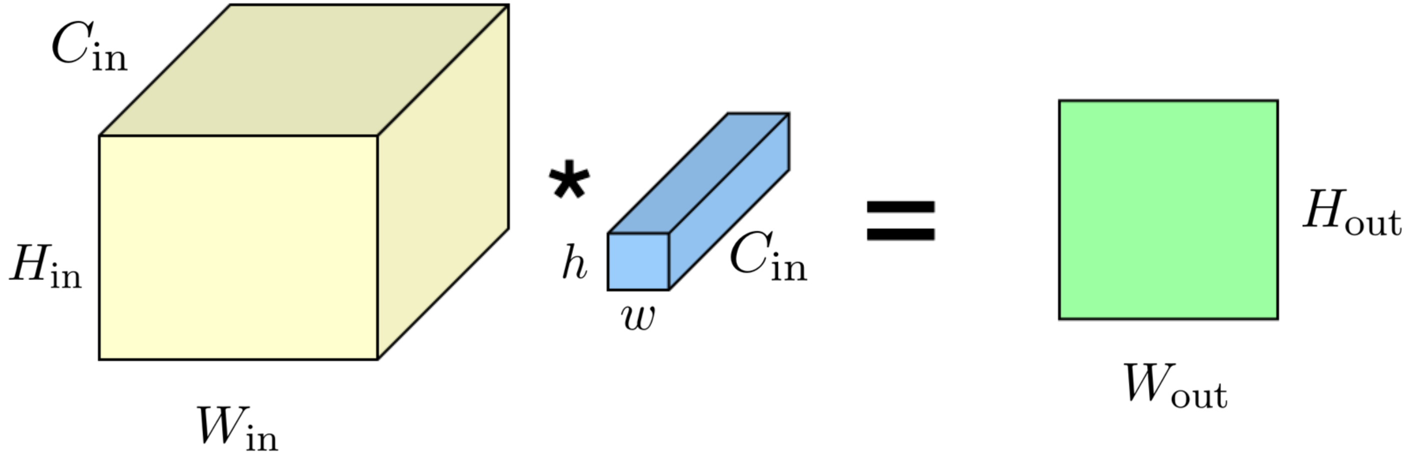
$$y[i,j,k] = x[i,j,k] \star h[i,j,k] = \sum_{(m,n,o)\in \mathrm{supp}(K)} h[m,n,o]x[i+m,j+n,k+o]$$



"slide" h over and compute 3D dot product for each output voxel

CONV2D FILTERING IN DEEP LEARNING

height
x
width
x
channels

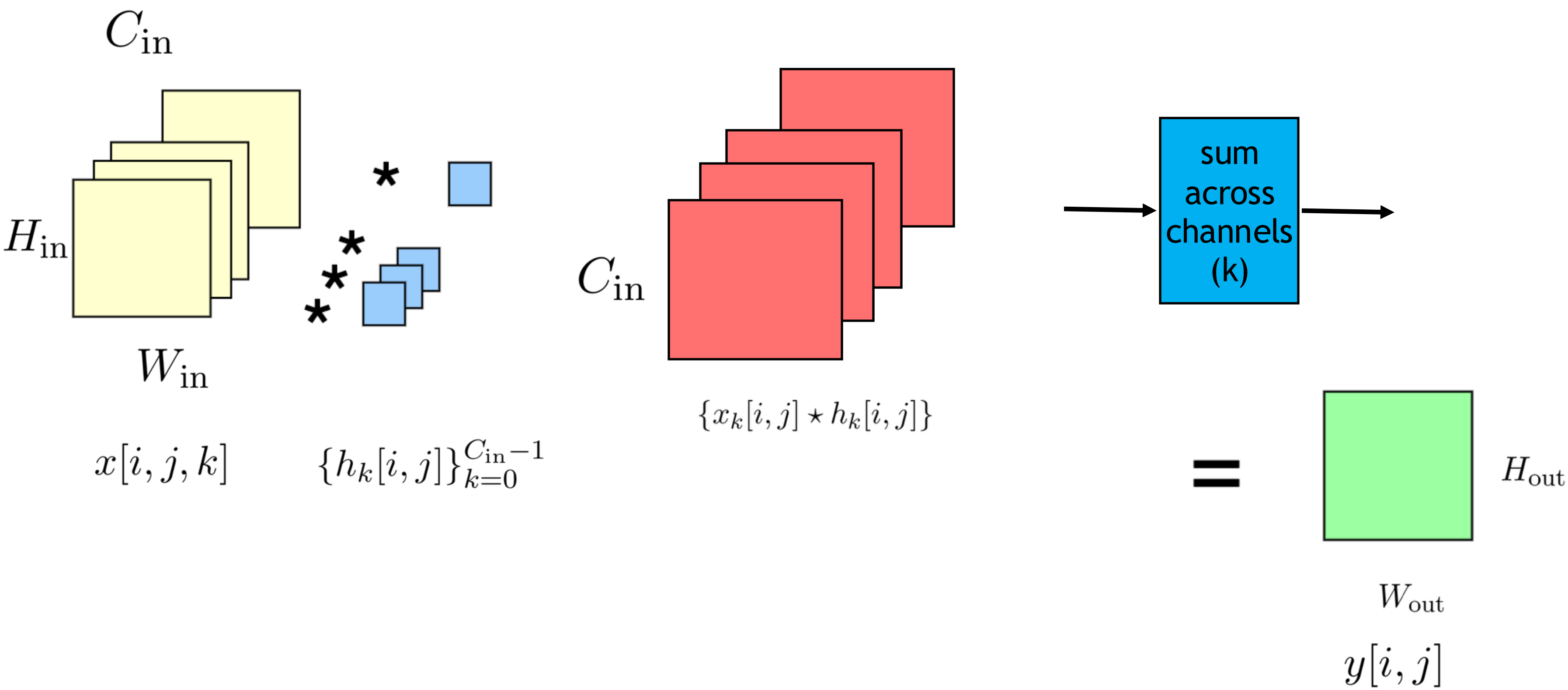


$x[i, j, k]$ $h[i, j, k]$ $y[i, j]$

typically, $h = w \sim 3$

convolution is done with no padding in the depth dimension,
so at each "shift" a single output pixel is generated

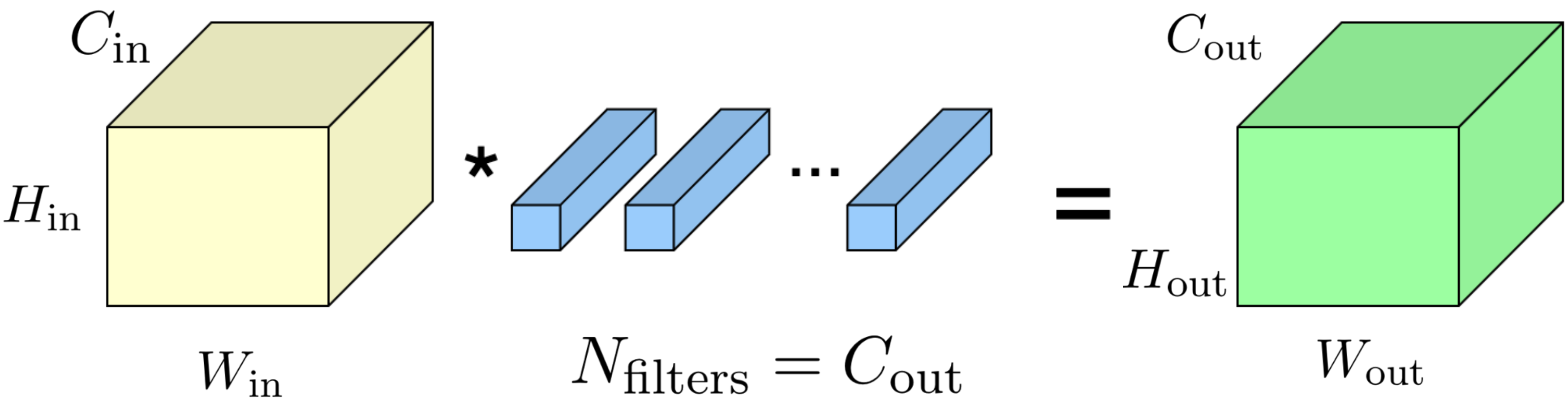
CONV2D FILTERING IN DEEP LEARNING



typically, $h = w \sim 3$

functionally equivalent to previous slide

CONV2D FILTERING IN DEEP LEARNING



$x[i, j, k]$

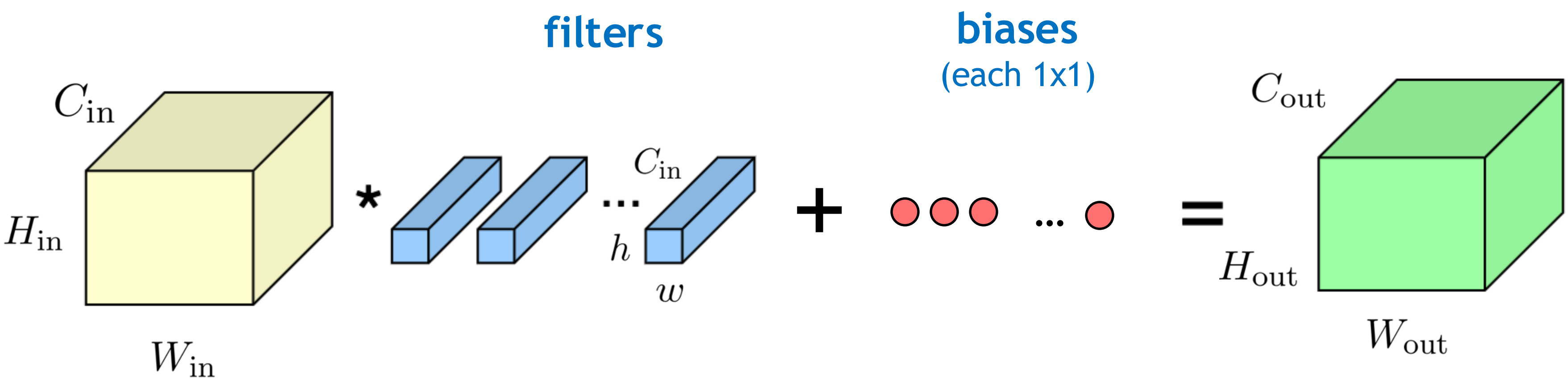
height
x
width
x
channels

$y[i, j, k]$

input feature map

output feature map

CONV2D LAYER



$$N_{\text{filters}} = C_{\text{out}} \qquad N_{\text{biases}} = N_{\text{filters}}$$

$x[i, j, k]$

input feature map

this replaces:

$\boldsymbol{y} = W\boldsymbol{x} + \boldsymbol{b}$

in MLPs — *i.e.*, produces linear activations

$y[i, j, k]$

output feature map

CONV2D LAYER IN PYTORCH

```
CLASS torch.nn.Conv2d(in_channels: int, out_channels: int, kernel_size: Union[T, Tuple[T, T]], stride: Union[T, Tuple[T, T]] = 1, padding: Union[T, Tuple[T, T]] = 0, dilation: Union[T, Tuple[T, T]] = 1, groups: int = 1, bias: bool = True, padding_mode: str = 'zeros')
```
[SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

```
nn.Conv2d(3, 32, 3, padding: [1])
```

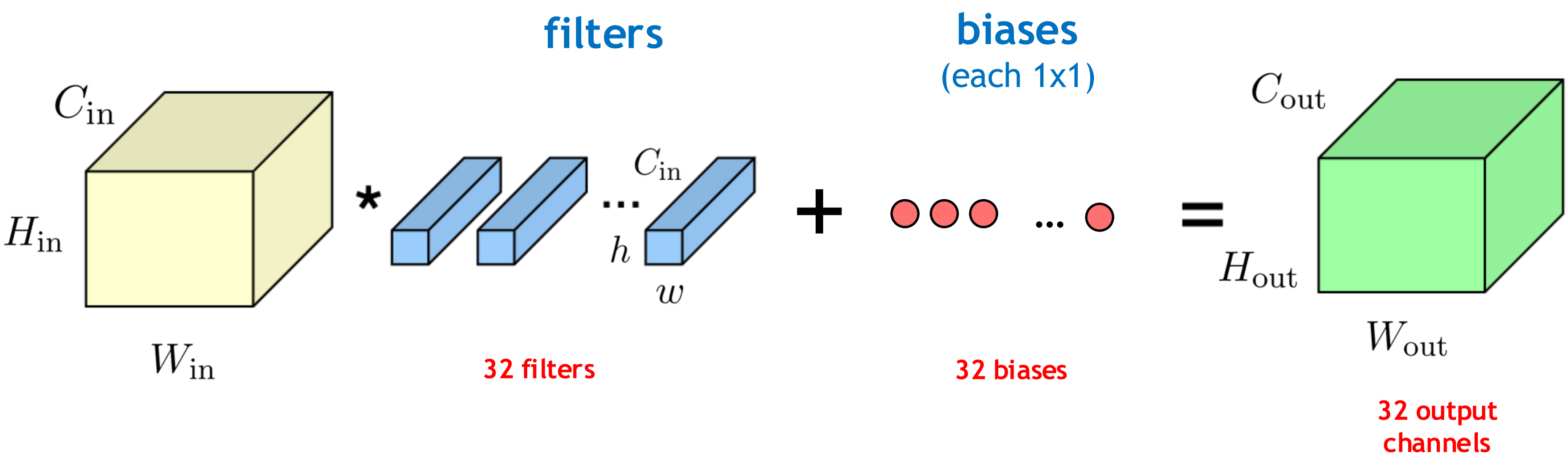
32 filters, each $(H, W, C) = (H, W, D) = (3, 3, C_{in})$

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html#torch.nn.Conv2d

CONV2D LAYER IN PYTORCH

```
nn.Conv2d(16, 32, 3, padding:[1])
```



assume padding="same" and:

$$C_{\text{out}} = 32$$
$$C_{\text{in}} = 16$$
$$H_{\text{in}} = 64$$
$$W_{\text{in}} = 64$$
$$h = w = 3$$

input activations (IFM size): 16*64*64 = 65,536

output activations (OFM size): 32*64*64 = 131,072

filter weights/coefficients: 32*(3*3*16) = 4,608

biases: 32

Total trainable parameters in this Conv2D: **4,640**

CONV2D LAYER IN PYTORCH

```
nn.Conv2d(16, 32, 3, padding:[1])
```

input activations (IFM size): 16*64*64 = 65,536

output activations (OFM size): 32*64*64 = 131,072

Total trainable parameters in this Conv2D: **4,640**

how does this compare to a dense layer with same number of input/output activations?

65,536 * 131,072 + 131,072 = **8,590,065,664**

why does the Conv2D layer have some many fewer trainable parameters?

PARAMETER REUSE IN CNNS

```
nn.Conv2d(16, 32, 3, padding:[1])
```

Total trainable parameters in this Conv2D: 4,640

Total trainable parameters for comparable dense layer: 8,590,065,664

why does the Conv2D layer have some many fewer trainable parameters?

parameters are reused!!

each filter is used many times over the input feature map

sparse connectivity

output (i, j) depend only on inputs in neighborhood of (i, j)

"Positive" View: CNNs have fewer parameters than MLPs with same number of activations

"Negative" View: CNNs do more computations per trainable parameter

TWO KEY CNN CONCEPTS

Localized features in the inputs

(*e.g.*, natural images)

Parameter Reuse

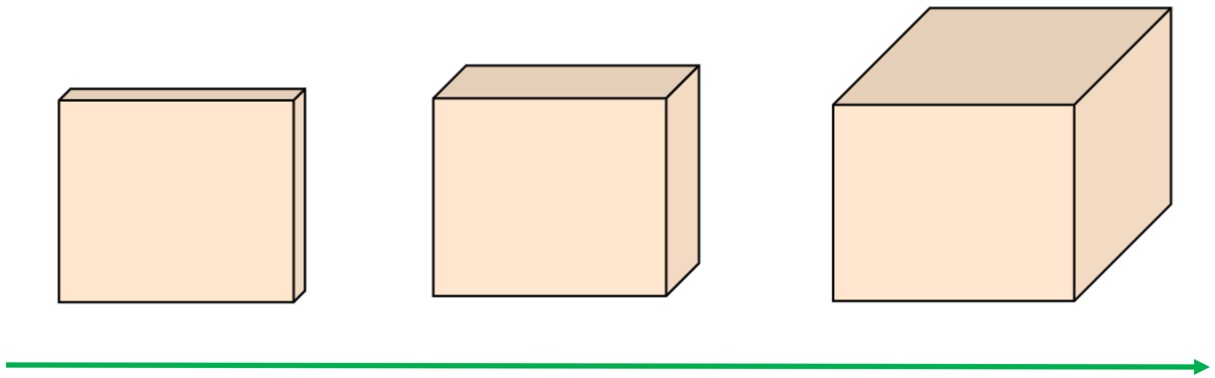
(*e.g.*, filter is used many times over input feature map)

OUTLINE FOR SLIDES

- Motivation, applications
- Basic 2D convolution operations
  - PyTorch 2Dconv layer
- Pooling and stride
- Fashion MNIST example
- Visualization methods
- Some common CNN structures
- Reduced complexity CNN architectures
- Outline of Back-propagation for CNNs

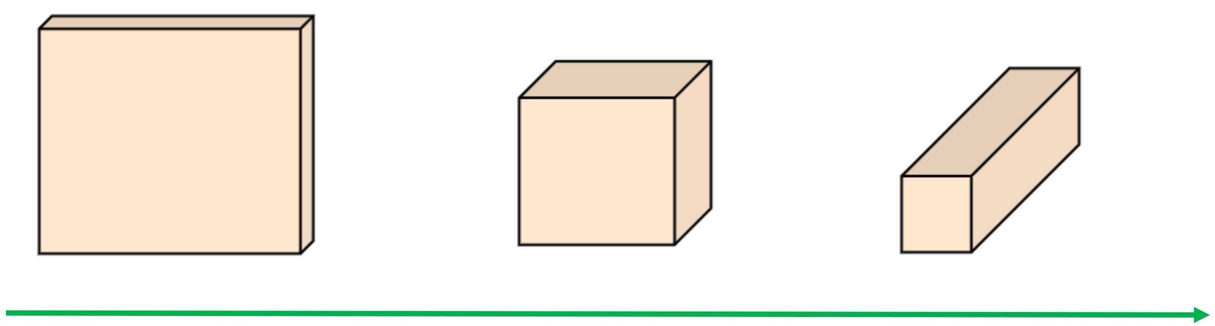
POOLING AND STRIDE

TYPICAL CNN STRUCTURES/PATTERNS



more channels as you go deeper

need to manage this –
i.e., reduce height and width



need some kind of "down-sampling"

doubling number of channels is common

USC Viterbi School of Engineering

DOWN-SAMPLING: STRIDE > 1

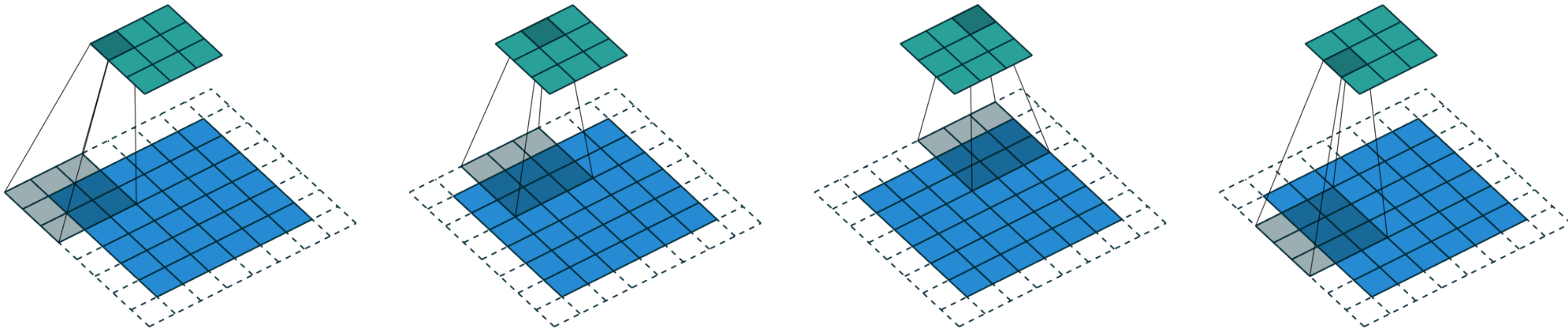


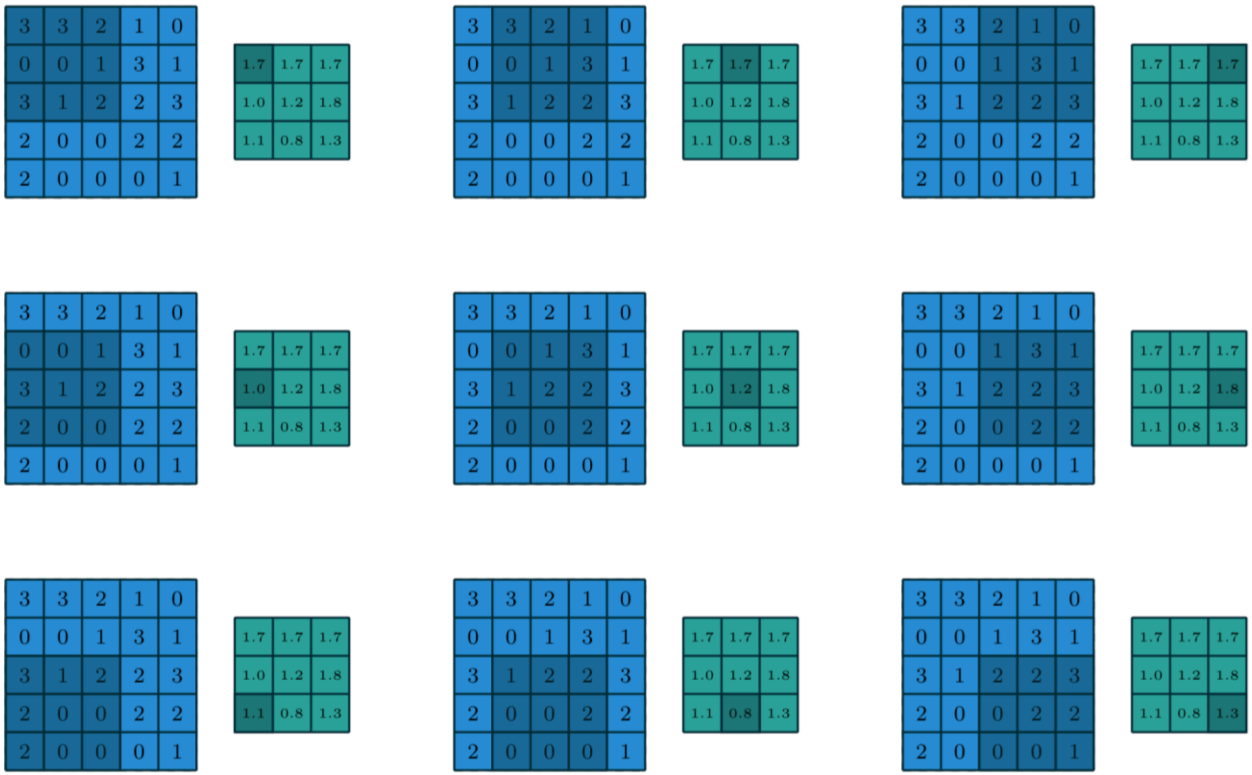
Figure 2.7: (Arbitrary padding and strides) Convolving a 3×3 kernel over a 6×6 input padded with a 1×1 border of zeros using 2×2 strides (i.e., $i = 6$, $k = 3$, $s = 2$ and $p = 1$). In this case, the bottom row and right column of the zero padded input are not covered by the kernel.

convolution, but the stride is >1

reduces H, W

Dumoulin, Vincent, and Francesco Visin. "A guide to convolution arithmetic for deep learning." arXiv preprint arXiv:1603.07285 (2016).

DOWN-SAMPLING: AVERAGE POOLING



average pooling layer

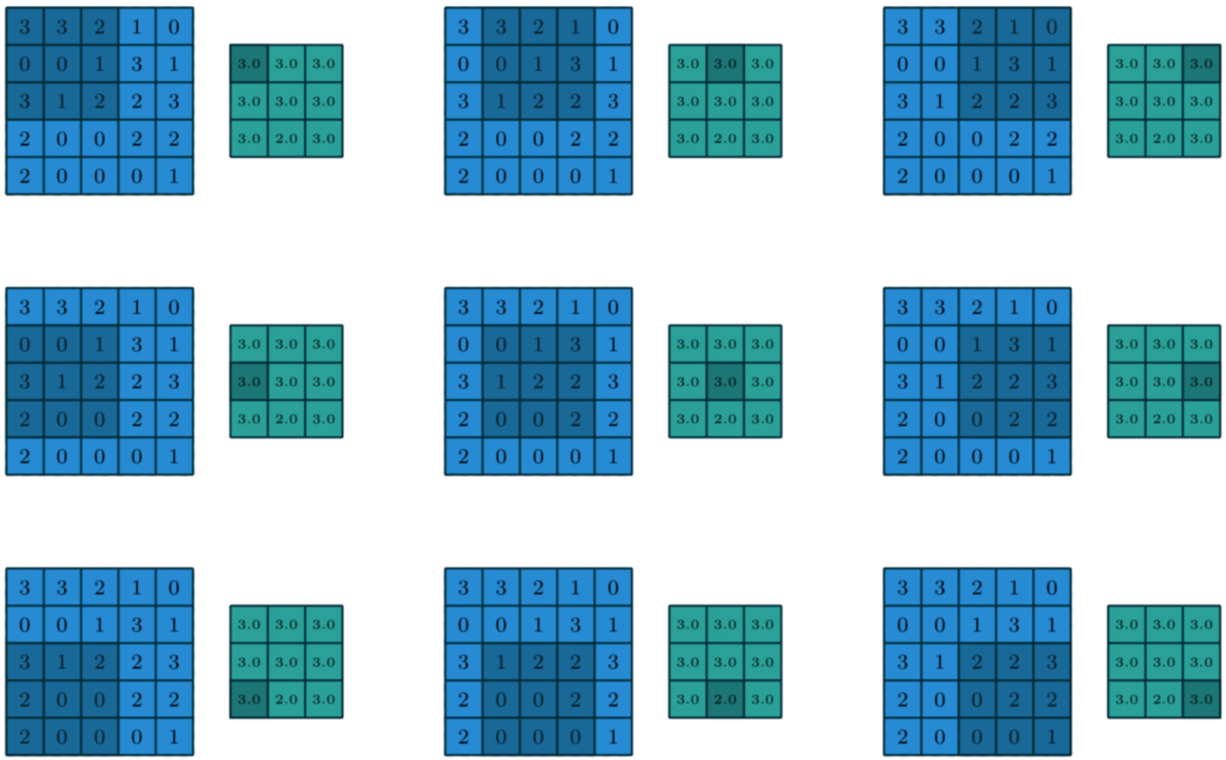
like convolution w/o padding and 1/9 for all 3x3 **fixed** kernel coefficients & stride = pool_size

reduces H, W

Figure 1.5: Computing the output values of a 3×3 average pooling operation on a 5×5 input using 1×1 strides.

Dumoulin, Vincent, and Francesco Visin. "A guide to convolution arithmetic for deep learning." arXiv preprint arXiv:1603.07285 (2016).

DOWN-SAMPLING: MAX POOLING



max pooling layer

like convolution,
but take max
element in
kernel support
&stride = pool_size

reduces H, W

Figure 1.6: Computing the output values of a 3×3 max pooling operation on a 5×5 input using 1×1 strides.

Dumoulin, Vincent, and Francesco Visin. "A guide to convolution arithmetic for deep learning." arXiv preprint arXiv:1603.07285 (2016).

USC Viterbi School of Engineering

MAX POOLING EXAMPLE — KERNEL SIZE = (2,2)

```python
import numpy as np
import torch
import torch.nn as nn

layer = nn.MaxPool2d(2)

test_input = torch.tensor(np.arange(100).reshape((1, 1, 10, 10)).astype(float))
test_output = layer(test_input)

print(test_input)
print(test_output)
```

```
tensor([[[[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.],
          [10., 11., 12., 13., 14., 15., 16., 17., 18., 19.],
          [20., 21., 22., 23., 24., 25., 26., 27., 28., 29.],
          [30., 31., 32., 33., 34., 35., 36., 37., 38., 39.],
          [40., 41., 42., 43., 44., 45., 46., 47., 48., 49.],
          [50., 51., 52., 53., 54., 55., 56., 57., 58., 59.],
          [60., 61., 62., 63., 64., 65., 66., 67., 68., 69.],
          [70., 71., 72., 73., 74., 75., 76., 77., 78., 79.],
          [80., 81., 82., 83., 84., 85., 86., 87., 88., 89.],
          [90., 91., 92., 93., 94., 95., 96., 97., 98., 99.]]]],
       dtype=torch.float64)
tensor([[[[11., 13., 15., 17., 19.],
          [31., 33., 35., 37., 39.],
          [51., 53., 55., 57., 59.],
          [71., 73., 75., 77., 79.],
          [91., 93., 95., 97., 99.]]]], dtype=torch.float64)
```

DOWN-SAMPLING IN PYTORCH

https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

dilation is "spreading" the 2D kernel values over larger filed of view

```
nn.Conv2d(
    in_channels: int, out_channels: int,
    kernel_size: Union[T, Tuple[T, T]],
    stride: Union[T, Tuple[T, T]] = 1,
    padding: Union[T, Tuple[T, T]] = 0,
    dilation: Union[T, Tuple[T, T]] = 1,
    padding_mode: str = 'zeros',
    groups: int = 1, bias: bool = True
)
```

https://pytorch.org/docs/stable/generated/torch.nn.AvgPool2d.html

default strides for max/avg pooling is kernel_size

```
nn.AvgPool2d(
  kernel_size = (2,2),
  padding = (1,1)
)
```

DILATION IN CONV2D

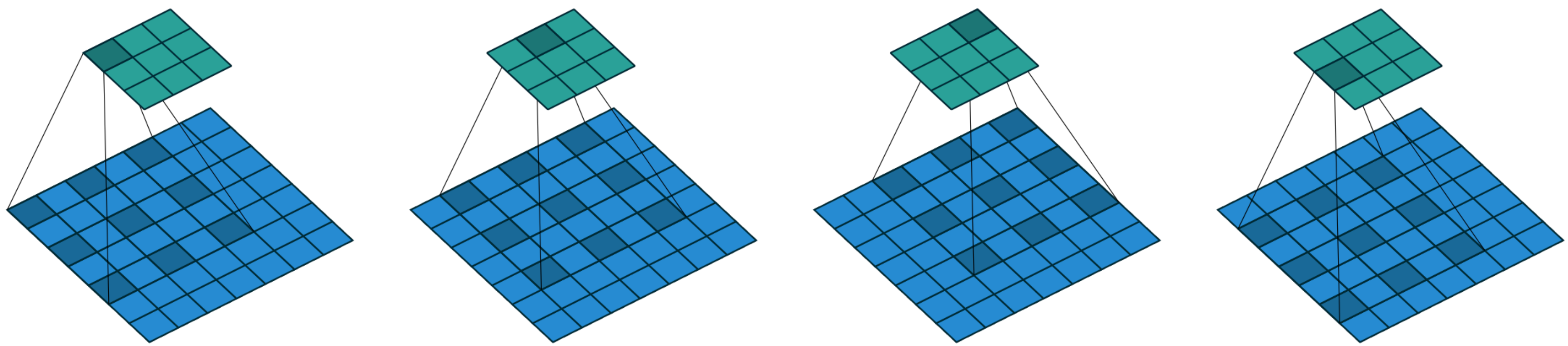


Figure 5.1: (Dilated convolution) Convolving a 3×3 kernel over a 7×7 input with a dilation factor of 2 (i.e., $i = 7$, $k = 3$, $d = 2$, $s = 1$ and $p = 0$).

not as common

```
nn.Conv2d(dilation: n)
```

Dumoulin, Vincent, and Francesco Visin. "A guide to convolution arithmetic for deep learning." arXiv preprint arXiv:1603.07285 (2016).

OUTLINE FOR SLIDES

- Motivation, applications
- Basic 2D convolution operations
  - PyTorch 2Dconv layer
- Pooling and stride
- Fashion MNIST example
- Visualization methods
- Some common CNN structures
- Reduced complexity CNN architectures
- Outline of Back-propagation for CNNs

EXAMPLE

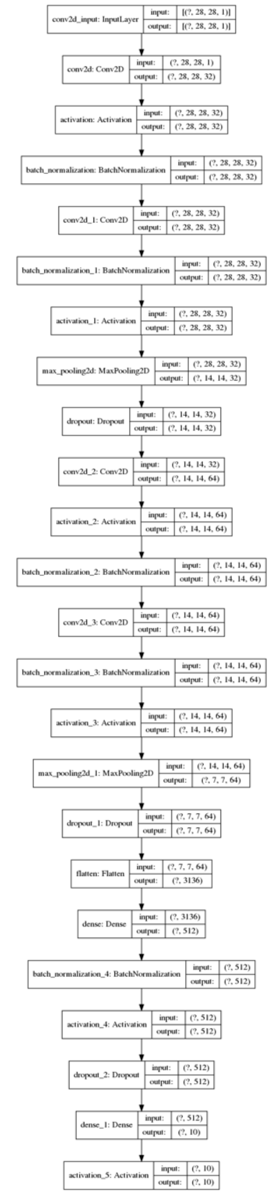
LET'S JUMP IN… PYTORCH

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        320
_________________________________________________________________
activation (Activation)      (None, 28, 28, 32)        0
_________________________________________________________________
batch_normalization (BatchNo (None, 28, 28, 32)        128
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 32)        9248
_________________________________________________________________
activation_1 (Activation)    (None, 28, 28, 32)        0
_________________________________________________________________
batch_normalization_1 (Batch (None, 28, 28, 32)        128
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 64)        18496
_________________________________________________________________
activation_2 (Activation)    (None, 14, 14, 64)        0
_________________________________________________________________
batch_normalization_2 (Batch (None, 14, 14, 64)        256
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 14, 14, 64)        36928
_________________________________________________________________
activation_3 (Activation)    (None, 14, 14, 64)        0
_________________________________________________________________
batch_normalization_3 (Batch (None, 14, 14, 64)        256
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 7, 7, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 3136)              0
_________________________________________________________________
dense (Dense)                (None, 512)               1606144
_________________________________________________________________
activation_4 (Activation)    (None, 512)               0
_________________________________________________________________
batch_normalization_4 (Batch (None, 512)               2048
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                5130
_________________________________________________________________
activation_5 (Activation)    (None, 10)                0
=================================================================
Total params: 1,679,082
Trainable params: 1,677,674
Non-trainable params: 1,408
```

fmnist_cnn.py

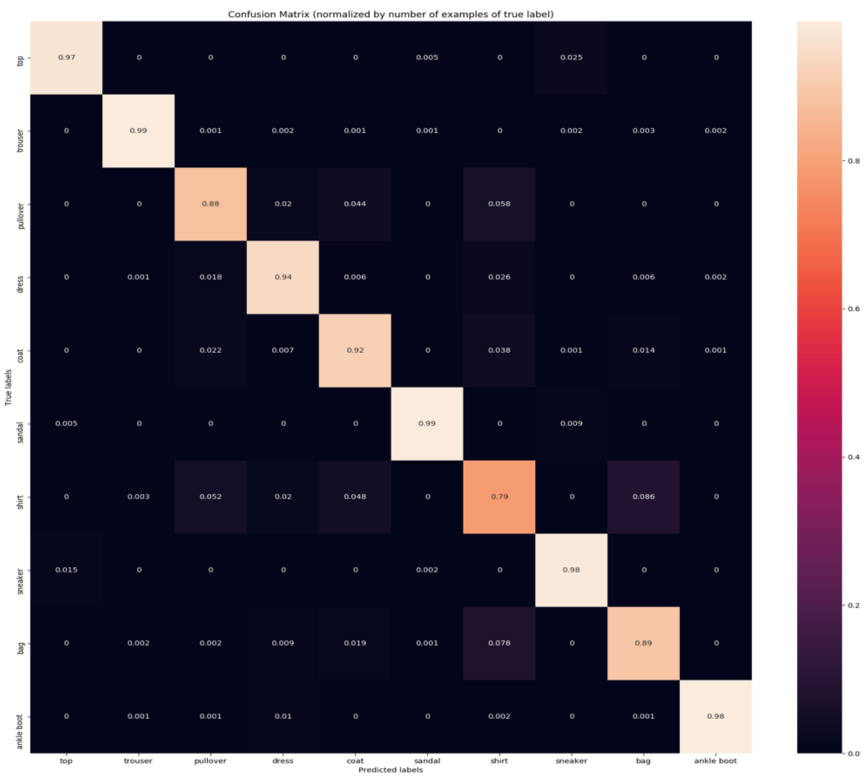
This achieves ~ 93.5% accuracy on Fashion MNSIT

(compare to ~88% with MLP)

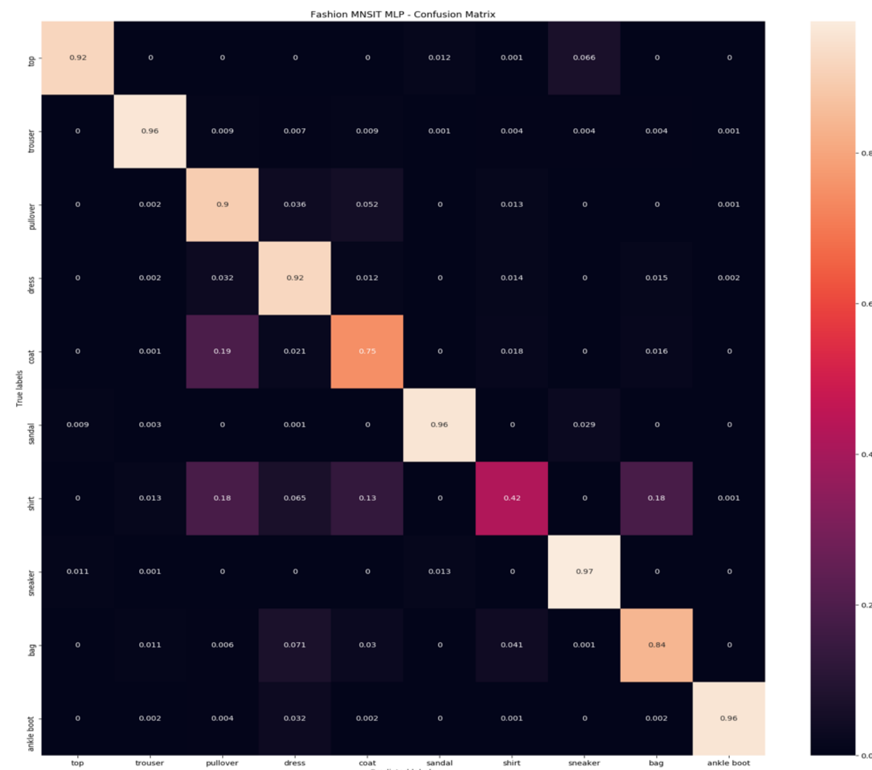


LET'S JUMP IN… PYTORCH

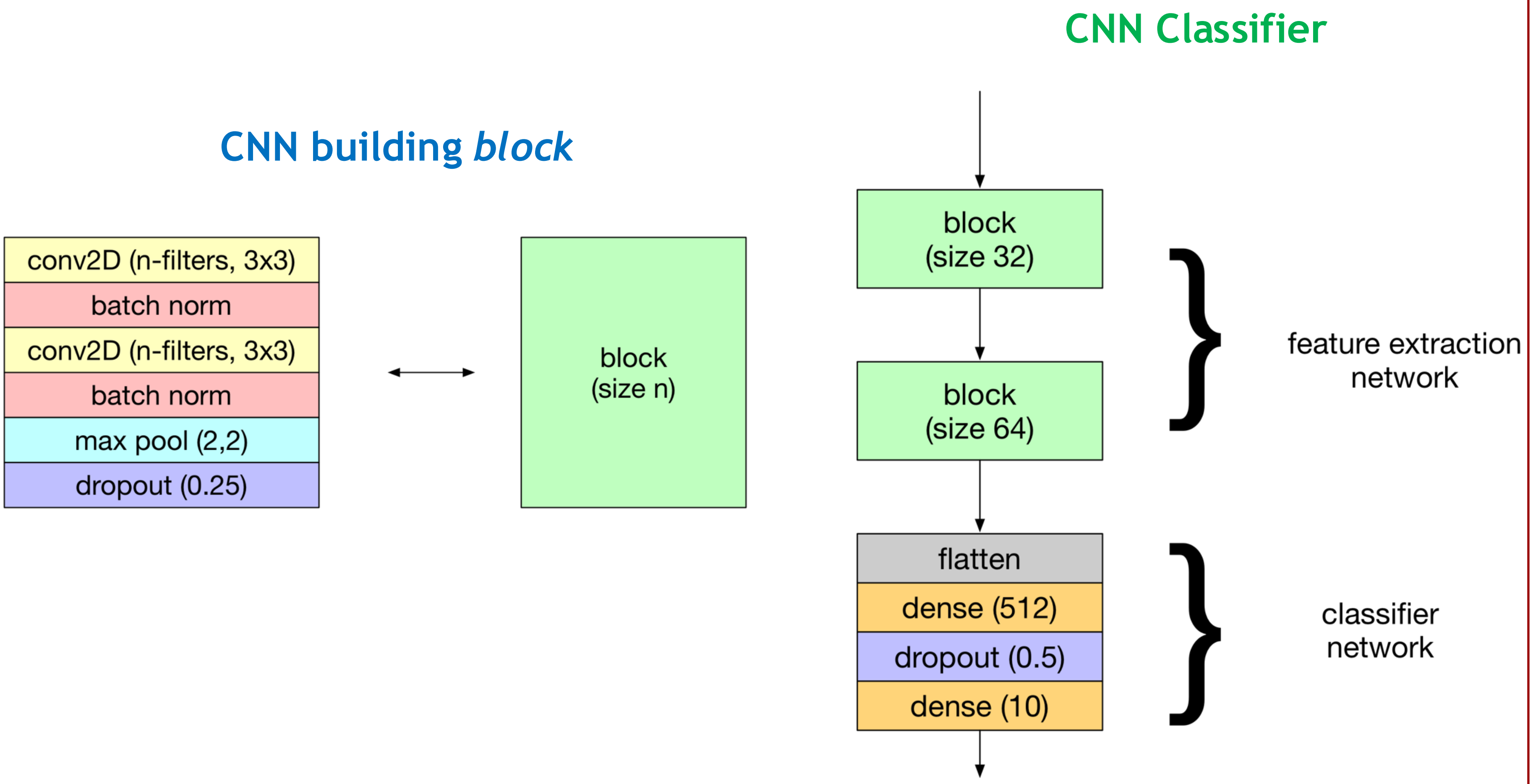
CNN



MLP



THIS IS A TYPICAL BLOCK-BASED CNN PATTERN



OUTLINE FOR SLIDES

- Motivation, applications
- Basic 2D convolution operations
  - PyTorch 2Dconv layer
- Pooling and stride
- Fashion MNIST example
- Visualization methods
- Some common CNN structures
- Reduced complexity CNN architectures
- Outline of Back-propagation for CNNs

VISUALIZATION

DOGS VS. CATS 😃

DOGS VS. CATS 😃

Dataset available here

https://www.kaggle.com/c/dogs-vs-cats

let's explore a simple CNN and see if we can get some insight into what the filters are looking for and how they respond to a given input image

DOGS-V-CATS: CATS AND DOGS – CNN.IPYNB

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 32, 150, 150]	896
Conv2d-2	[-1, 64, 150, 150]	18,496
MaxPool2d-3	[-1, 64, 75, 75]	0
Conv2d-4	[-1, 128, 75, 75]	73,856
Conv2d-5	[-1, 128, 75, 75]	147,584
MaxPool2d-6	[-1, 128, 37, 37]	0
Conv2d-7	[-1, 256, 37, 37]	295,168
Conv2d-8	[-1, 512, 37, 37]	1,180,160
MaxPool2d-9	[-1, 512, 18, 18]	0
Conv2d-10	[-1, 512, 18, 18]	2,359,808
Conv2d-11	[-1, 512, 18, 18]	2,359,808
MaxPool2d-12	[-1, 512, 8, 8]	0
Dropout2d-13	[-1, 32768]	0
Linear-14	[-1, 512]	16,777,728
Linear-15	[-1, 1]	513

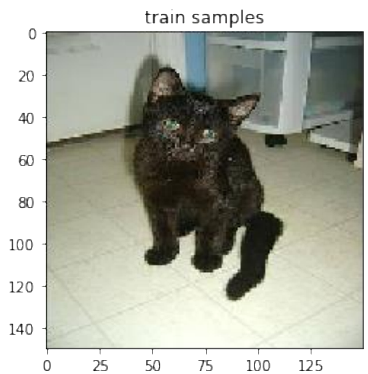
==

Total params: 23,214,017

Trainable params: 23,214,017

Non-trainable params: 0

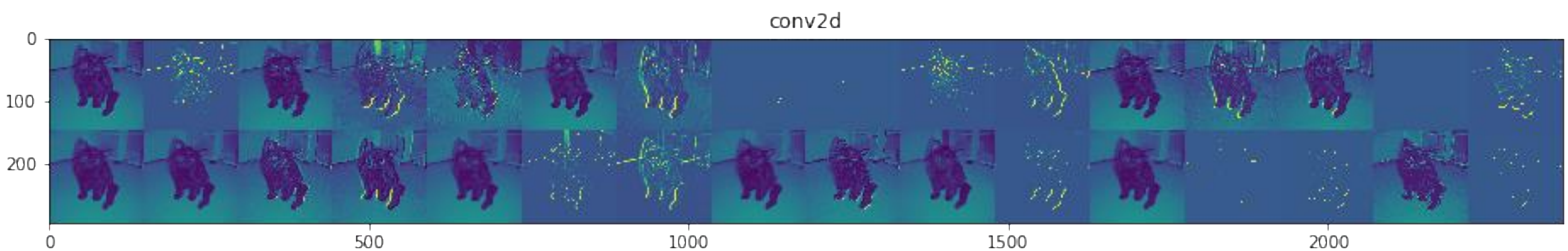
DOGS-V-CATS: VISUALIZING CNN FEATURE MAPS



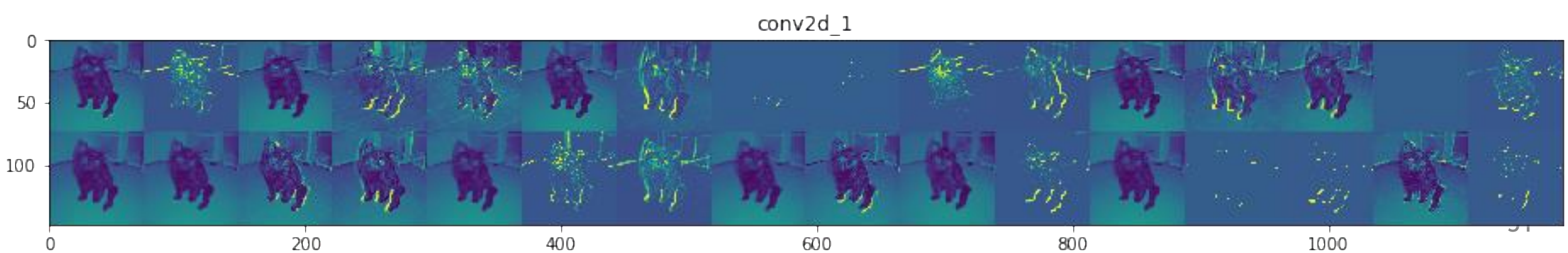
input image

Cats and Dogs – viz.ipynb

1st conv2D



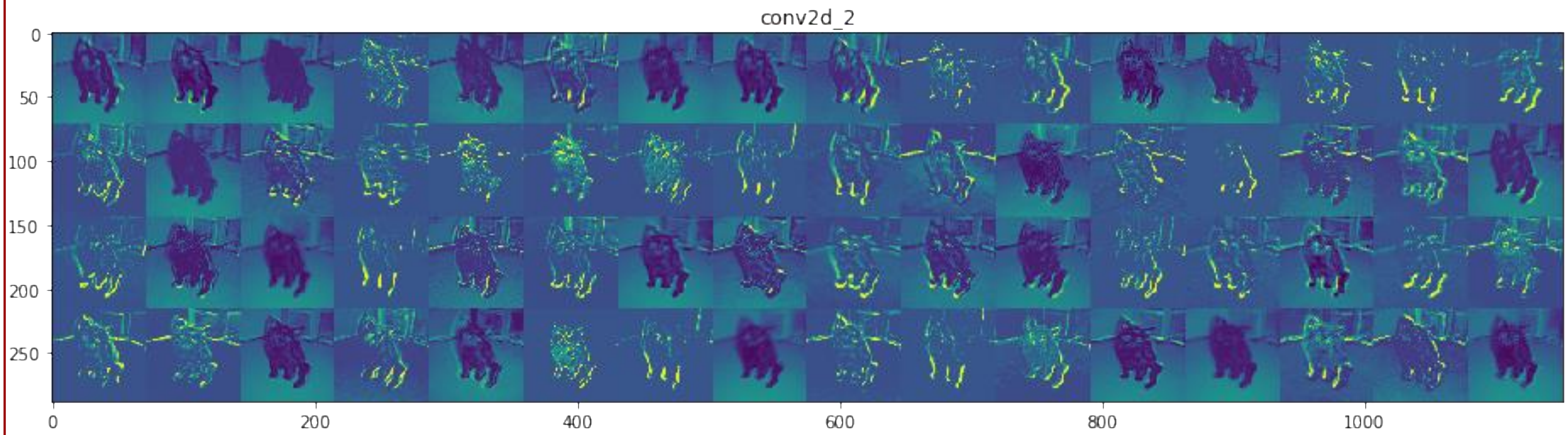
2nd conv2D



DOGS-V-CATS: VISUALIZING CNN FEATURE MAPS

Cats and Dogs – viz.ipynb

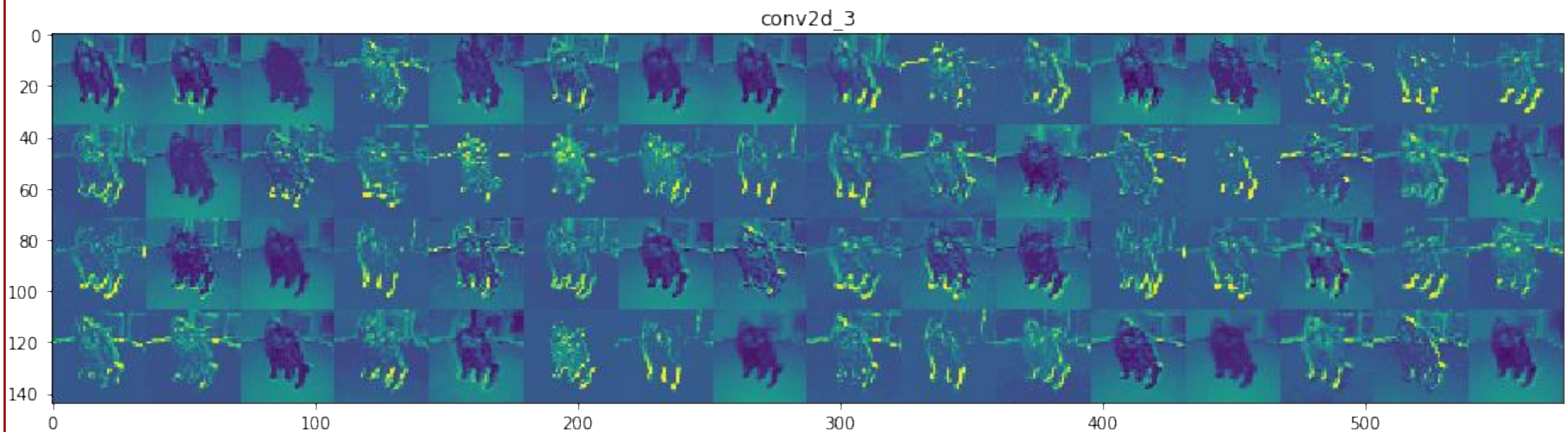
3rd conv2D



DOGS-V-CATS: VISUALIZING CNN FEATURE MAPS

Cats and Dogs – viz.ipynb

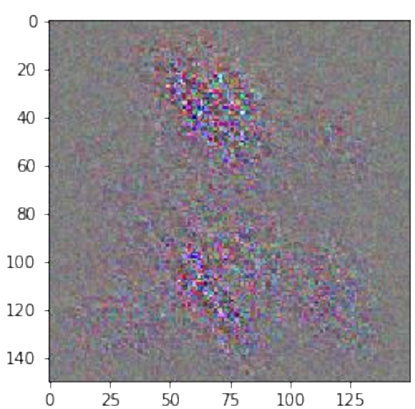
4th conv2D



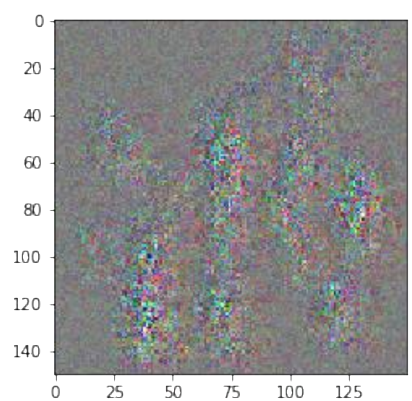
DOGS-V-CATS: MAX FILTER RESPONSE

train an input image so that it maximizes the output energy in a particular filter

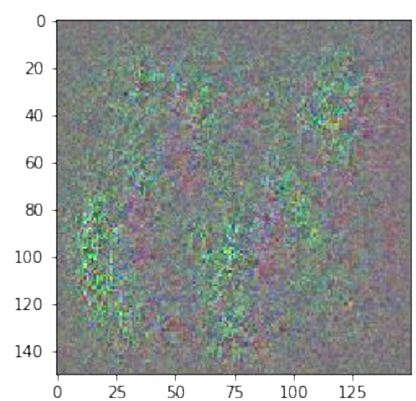
Cats and Dogs – viz.ipynb



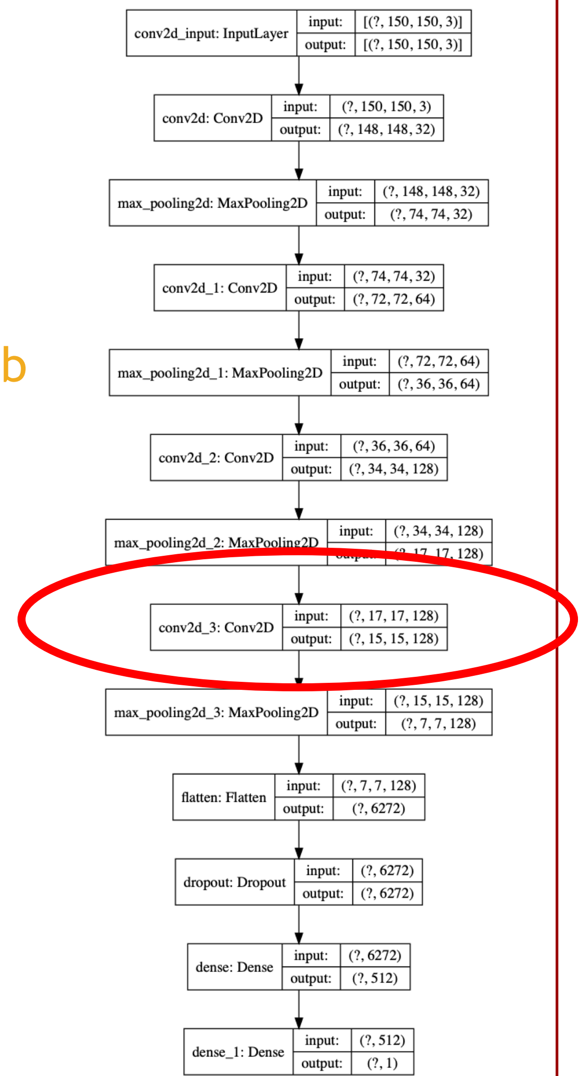
channel 16



channel 71

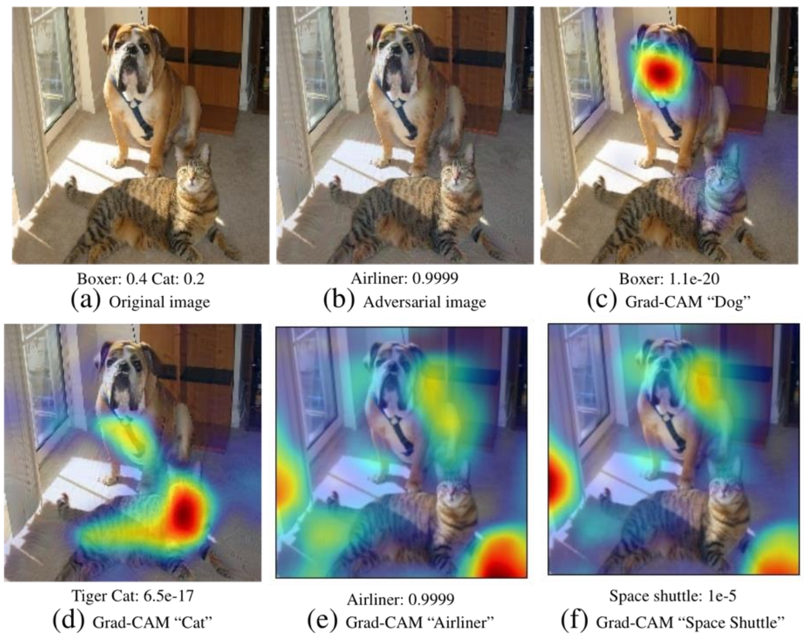


channel 121



CNN VISUALIZATION: GRAD-CAM

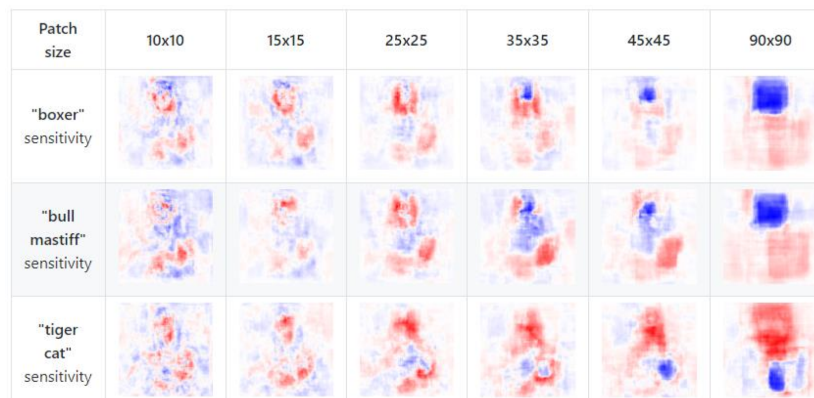
Gradient Weighted Class Activation Mapping



pyimagesearch tutorial (keras)

demo

https://github.com/kazuto1011/grad-cam-pytorch



see where a layer is "looking" for a given class

Selvaraju, Ramprasaath R., et al. "Grad-cam: Visual explanations from deep networks via gradient-based localization." Proceedings of the IEEE international conference on computer vision. 2017.

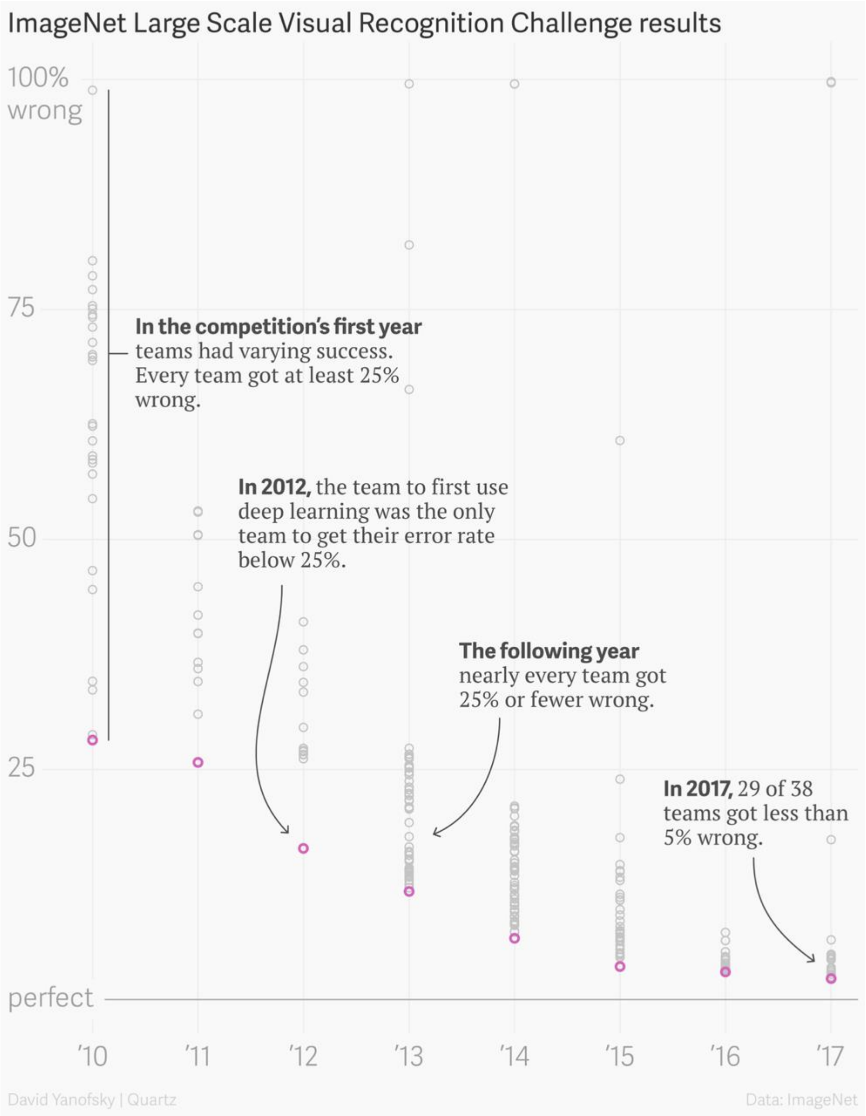
USC Viterbi
School of Engineering

OUTLINE FOR SLIDES

- Motivation, applications
- Basic 2D convolution operations
  - PyTorch 2Dconv layer
- Pooling and stride
- Fashion MNIST example
- Visualization methods
- Some common CNN structures
- Reduced complexity CNN architectures
- Outline of Back-propagation for CNNs

BLOCK STRUCTURES

CNNS: USE WHEN FEATURE INFORMATION IS LOCALIZED



2012: AlexNet

- ~60M parameters
- 16.4% top-5 error

2014: VGG

- ~140M parameters
- 10% top-5 error

2015: Inception (aka GoogLeNet)

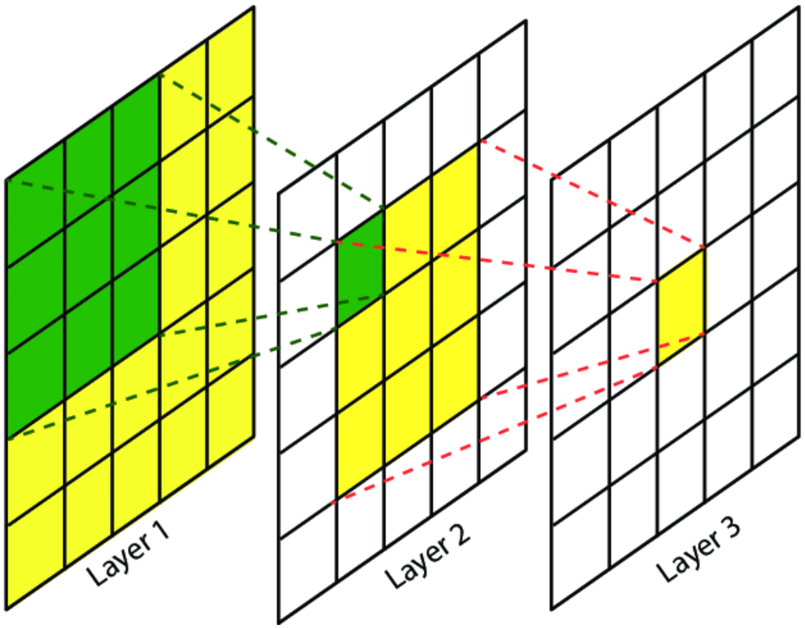
- ~4M parameters
- ~7% top-5 error

2015 ResNet

- ~60M parameters
- ~7% top-5 error

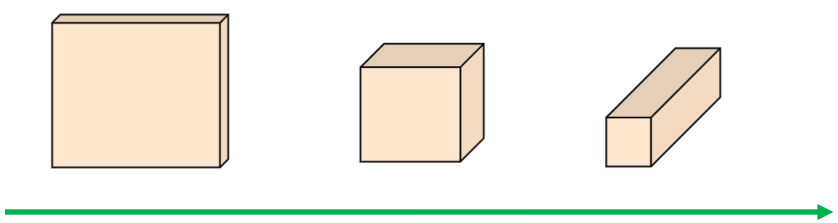
The leap that transformed AI research—
and possibly the world

RECEPTIVE FIELD AS WE GO DEEPER



deeper in the network, each pixel in the feature map can "see" more of the input image

reason why height and width of the feature map can be reduced as we go deeper



deeper into the network

Lin, Haoning, Zhenwei Shi, and Zhengxia Zou. "Maritime semantic labeling of optical remote sensing images with multi-scale fully convolutional network." Remote sensing 9.5 (2017): 480.

RECEPTIVE FIELD AS WE GO DEEPER

simple script to find input pixels that can affect output pixels for a specific CNN architecture (pytorch-receptive-field)

```
receptive_field_dict = receptive_field(model, (3, 256, 256))
receptive_field_for_unit(receptive_field_dict, '2', (2,2))
```

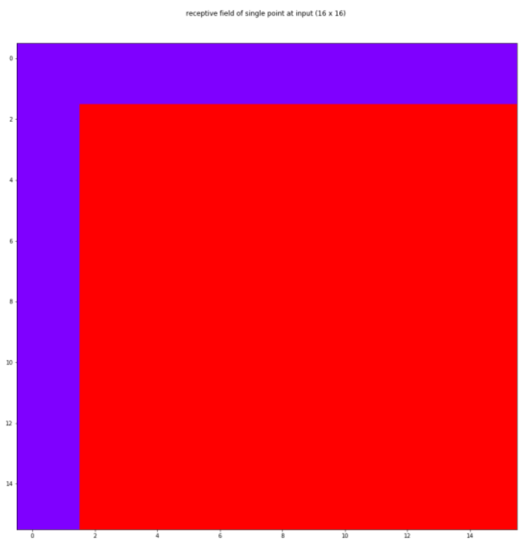
```
class Net(nn.Module):
 def __init__(self):
  super(Net, self).__init__()
  self.conv = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3,      bias=False)
  self.bn = nn.BatchNorm2d(64)
  self.relu = nn.ReLU(inplace=True)
  self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

 def forward(self, x):
  y = self.conv(x)
  y = self.bn(y)
  y = self.relu(y)
  y = self.maxpool(y)
  return y
```

```
------------------------------------------------------------------------------
    Layer (type)    map size      start       jump receptive_field
==============================================================================
        0             [256, 256]        0.5        1.0             1.0
        1             [128, 128]        0.5        2.0             7.0
        2             [128, 128]        0.5        2.0             7.0
        3             [128, 128]        0.5        2.0             7.0
        4               [64, 64]        0.5        4.0            11.0
==============================================================================
Receptive field size for layer 2, unit_position (1, 1),  is
 [(0, 6.0), (0, 6.0)]
```

Lin, Haoning, Zhenwei Shi, and Zhengxia Zou. "Maritime semantic labeling of optical remote sensing images with multi-scale fully convolutional network." Remote sensing 9.5 (2017): 480.

RECEPTIVE FIELD AS WE GO DEEPER

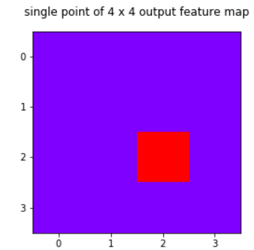
simple script to find input pixels that can affect output pixels for a specific CNN architecture



inverse image

←

receptive field



this could also be computed by hand by book-keeping the inverse image of each conv2D and pool layer

pytorch-receptive-field

POPULAR CNN ARCHITECTURES/PATTERNS

There are pretrained ImageNet models in PyTorch

("model-zoo")

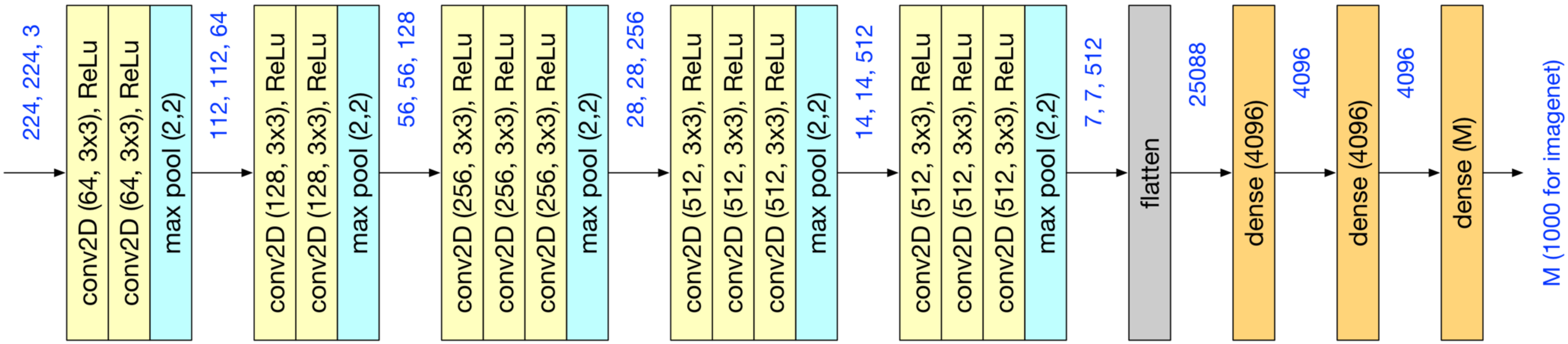
```
import torchvision.models as models
resnet18 = models.resnet18(pretrained=True)
alexnet = models.alexnet(pretrained=True)
squeezenet = models.squeezenet1_0(pretrained=True)
vgg16 = models.vgg16(pretrained=True)
densenet = models.densenet161(pretrained=True)
inception = models.inception_v3(pretrained=True)
googlenet = models.googlenet(pretrained=True)
shufflenet = models.shufflenet_v2_x1_0(pretrained=True)
mobilenet = models.mobilenet_v2(pretrained=True)
resnext50_32x4d = models.resnext50_32x4d(pretrained=True)
wide_resnet50_2 = models.wide_resnet50_2(pretrained=True)
mnasnet = models.mnasnet1_0(pretrained=True)
```

Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.713	0.901	138,357,544	23
VGG19	549 MB	0.713	0.900	143,667,240	26
ResNet50	98 MB	0.749	0.921	25,636,712	-
ResNet101	171 MB	0.764	0.928	44,707,176	-
ResNet152	232 MB	0.766	0.931	60,419,944	-
ResNet50V2	98 MB	0.760	0.930	25,613,800	-
ResNet101V2	171 MB	0.772	0.938	44,675,560	-
ResNet152V2	232 MB	0.780	0.942	60,380,648	-
InceptionV3	92 MB	0.779	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
MobileNet	16 MB	0.704	0.895	4,253,864	88
MobileNetV2	14 MB	0.713	0.901	3,538,984	88
DenseNet121	33 MB	0.750	0.923	8,062,504	121
DenseNet169	57 MB	0.762	0.932	14,307,880	169
DenseNet201	80 MB	0.773	0.936	20,242,984	201
NASNetMobile	23 MB	0.744	0.919	5,326,716	-
NASNetLarge	343 MB	0.825	0.960	88,949,818	-

The top-1 and top-5 accuracy refers to the model's performance on the ImageNet validation dataset.

https://pytorch.org/docs/stable/torchvision/models.html

COMMON CNN ARCHITECTURE PATTERNS - VGG16



Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

COMMON CNN ARCHITECTURE PATTERNS – RESNET(S)

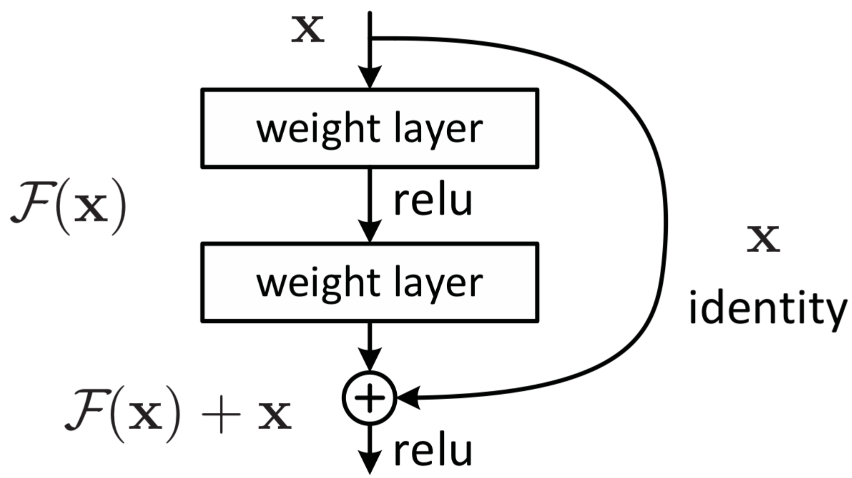


Figure 2. Residual learning: a building block.

residual connections:

aid in gradient flow (**reduce vanishing gradient**)

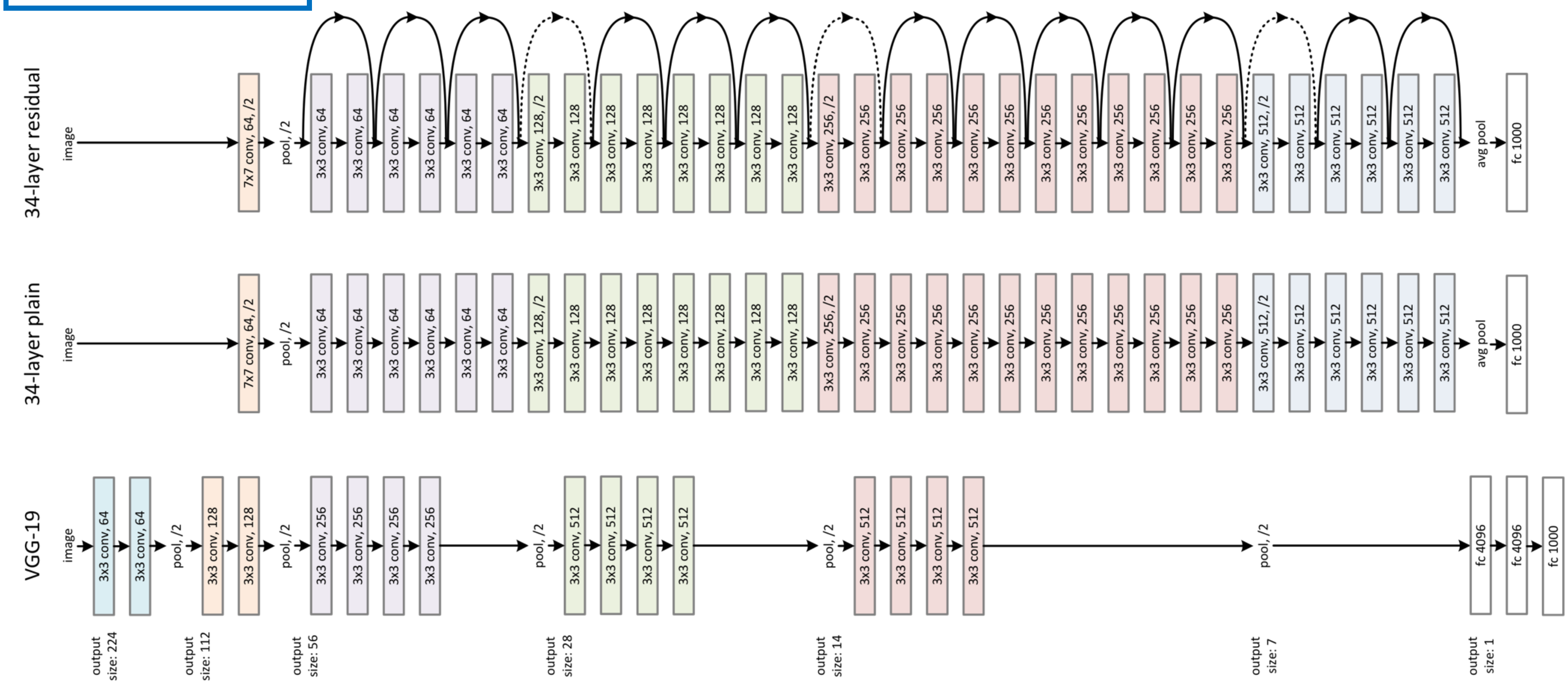
allow learning of "alternative" networks

– *e.g.*, can learn to bypass the two "weight layers" in this figure

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

COMMON CNN ARCHITECTURE PATTERNS – RESNET(S)

ResNet34



He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

COMMON CNN ARCHITECTURE PATTERNS – RESNET(S)

method	top-1 err.	top-5 err.
VGG [41] (ILSVRC'14)	-	8.43†
GoogLeNet [44] (ILSVRC'14)	-	7.89
VGG [41] (v5)	24.4	7.1
PReLU-net [13]	21.59	5.71
BN-inception [16]	21.99	5.81
ResNet-34 B	21.84	5.71
ResNet-34 C	21.53	5.60
ResNet-50	20.74	5.25
ResNet-101	19.87	4.60
ResNet-152	**19.38**	**4.49**

Table 4. Error rates (%) of **single-model** results on the ImageNet validation set (except † reported on the test set).

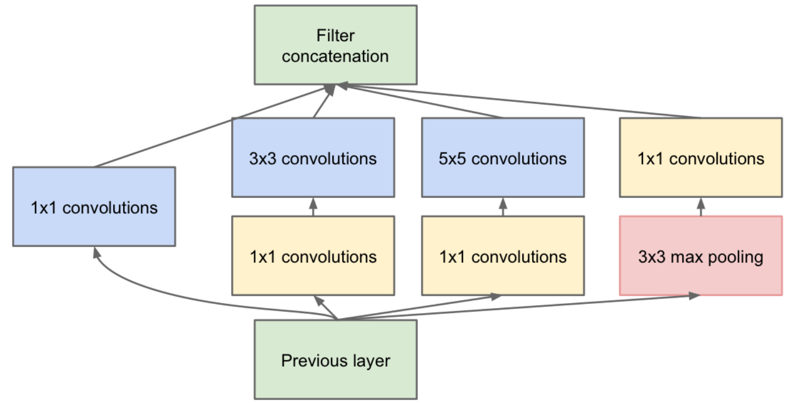
Note:

there are *v2* versions of these

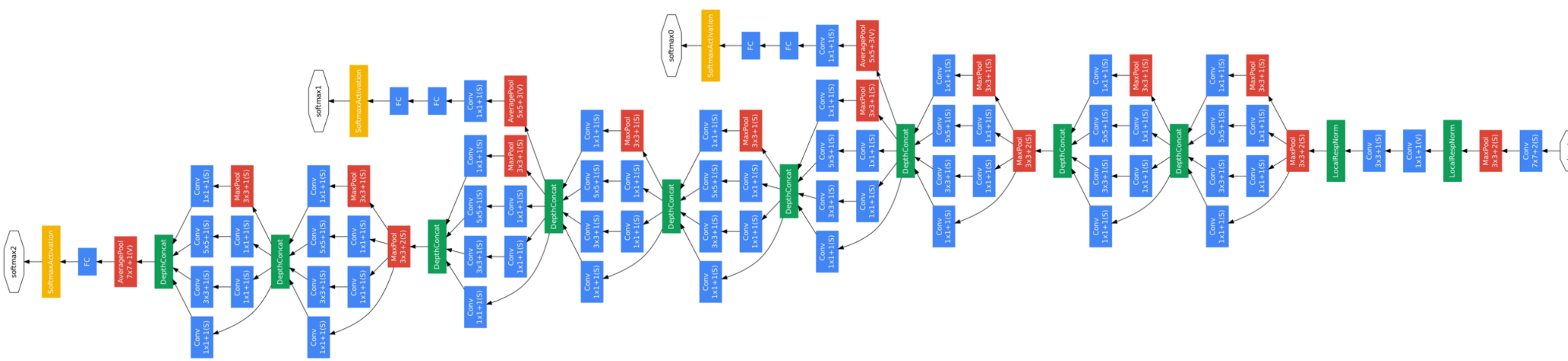
He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

COMMON CNN ARCHITECTURE PATTERNS - INCEPTION

aka GoogLeNet

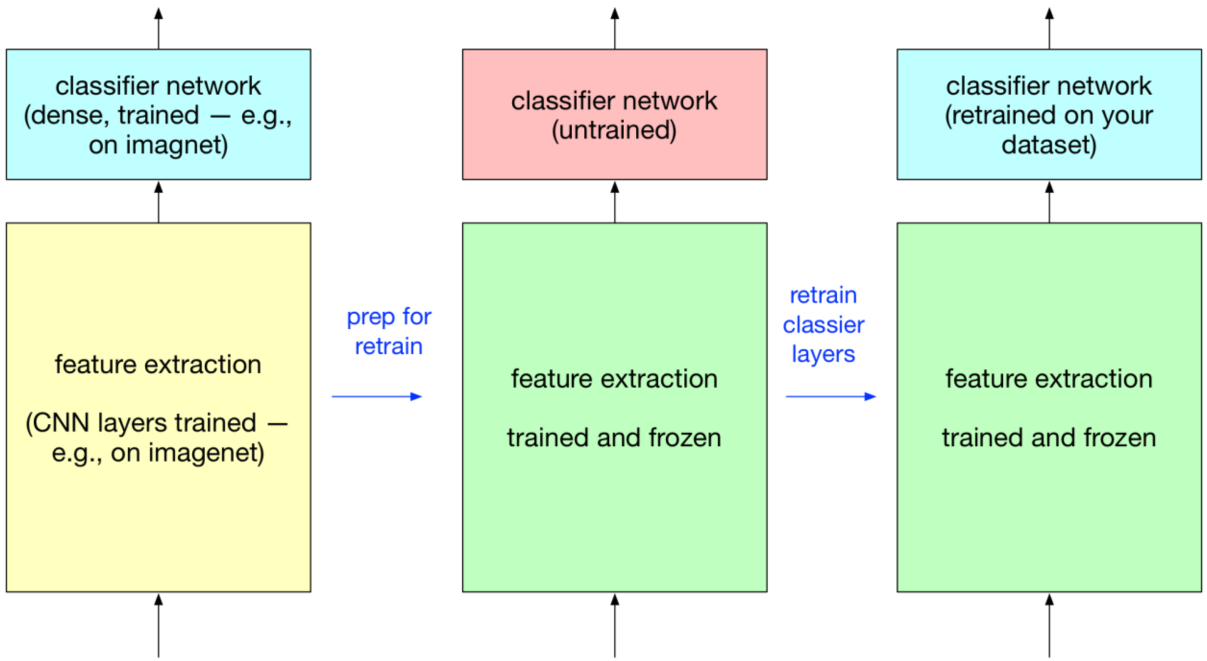


(b) Inception module with dimensionality reduction



Szegedy, Christian, et al. "Going deeper with convolutions." Proceedings of the IEEE conference on computer vision and pattern recognition. 2015.

USING FIXED CNN LAYERS FOR A DIFFERENT CV TASK



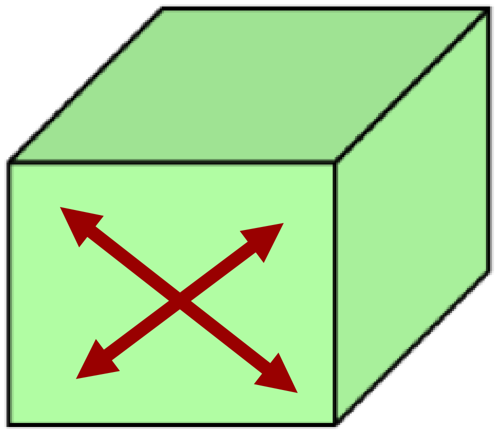
features needed for many CV tasks are similar to Imagenet classification features

you can reuse all or part of the feature extraction network

```
import torchvision.models as models
model = models.resnet50(pretrained=True)
```

USC Viterbi School of Engineering

ONE LAST LAYER TYPE: GLOBAL POOLING



pool over the pixels in a channel

```
torch.nn.MaxPool2d(kernel_size=image_size)

torch.nn.AvgPool2d(kernel_size=image_size)
```

follow with: `x.squeeze()`

Input: 4D tensor with shape (batch_size, rows, cols, channels)

Output: 2D tensor with shape (batch_size, channels)

this is used after the last conv2D/pool layer before the "flatten" in many recent models

reduces the complexity of the dense classification network without sacrificing performance

USC Viterbi School of Engineering

OUTLINE FOR SLIDES

- Motivation, applications
- Basic 2D convolution operations
  - PyTorch 2Dconv layer
- Pooling and stride
- Fashion MNIST example
- Visualization methods
- Some common CNN structures
- Reduced complexity CNN architectures
- Outline of Back-propagation for CNNs

REDUCING COMPLEXITY

REDUCED PARAMETER/COMPUTATION APPROACHES

For larger CNNs, the number of parameters is so large, that **storage complexity** becomes a significant issue

this is an issue for running these models in inference mode on mobile devices

computational complexity (during inference and training) is also an issue

there has been a lot of work on reducing the storage and computational complexity of CNNs — most have focused on inference of trained models

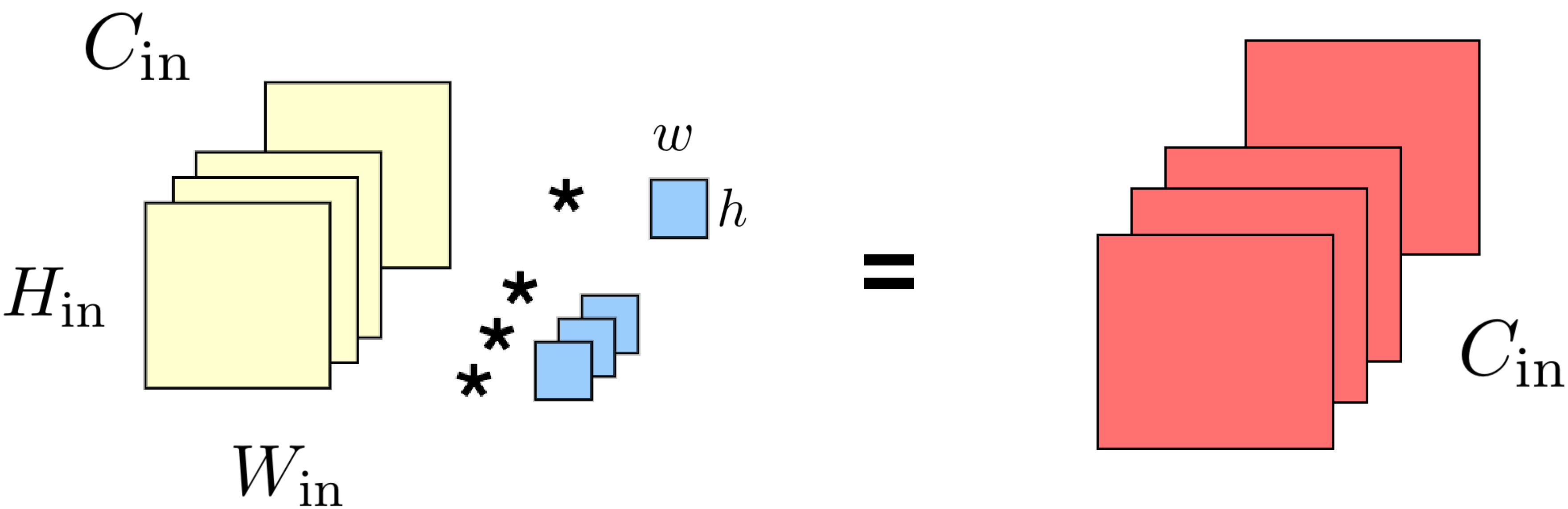
REDUCED PARAMETER/COMPUTATION APPROACHES

Two primary methods:

constrained filter structures: alter the standard conv2D operations to lower the computational/storage complexity

post-training processing to reduce complexity

CONSTRAINED FILTERING: DEPTH-WISE CONVOLUTION

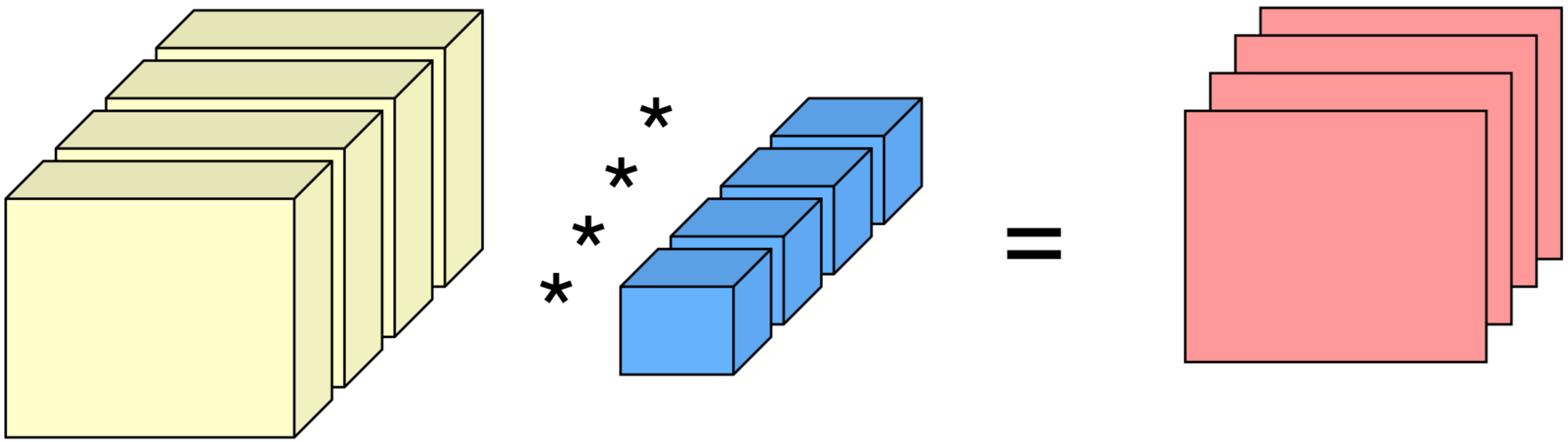


only do convolution **separately for channels**

– *i.e.*, no information is mixed across channels

USC Viterbi School of Engineering

CONSTRAINED FILTERING: GROUPWISE CONVOLUTION

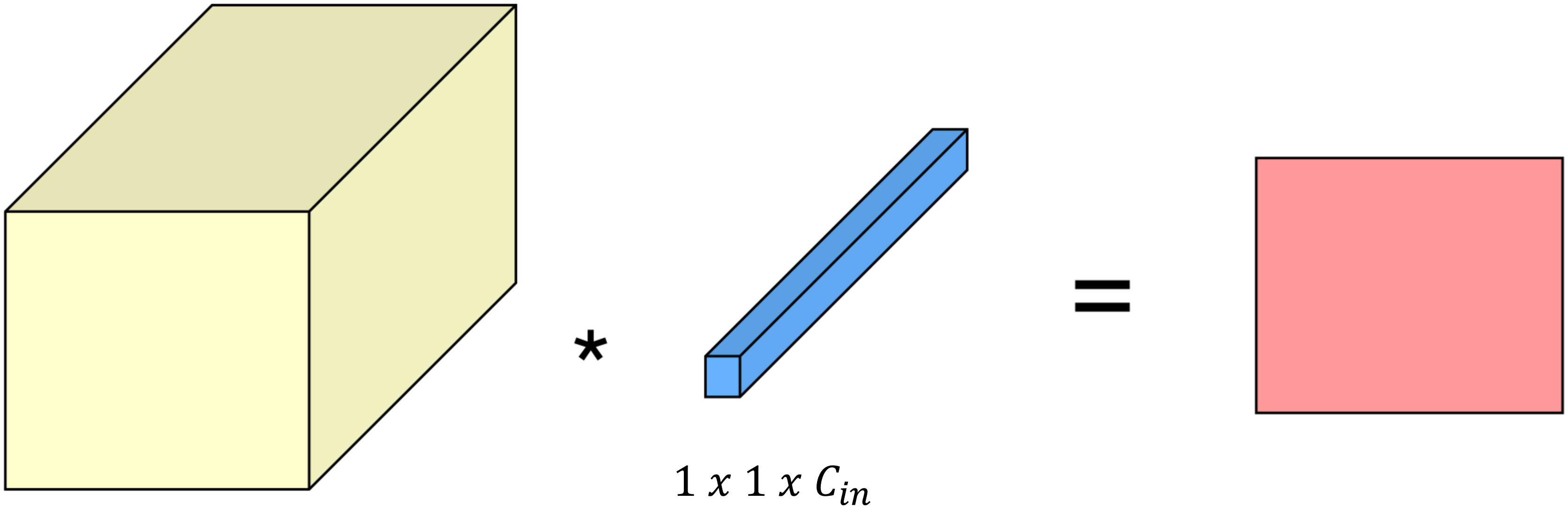


trade-off between standard conv2D filtering and depth-wise filtering

use more of these grouped-filters to get more output channels

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

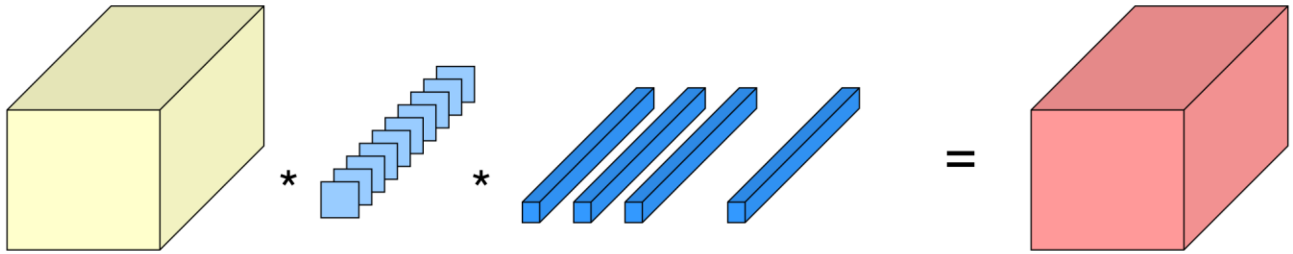
CONSTRAINED FILTERING: POINTWISE CONVOLUTION



standard Conv2D with filter size 1x1

***a.k.a.*, 1x1 convolution**

EXAMPLE: MOBILENET



$N_{\text{filters}} = C_{\text{out}}$

combine depth-wise convolution with many 1x1 convolutions

compare with standard Conv2D:

$$C_{\text{out}} = 32$$
$$C_{\text{in}} = 16$$
$$H_{\text{in}} = 64$$
$$W_{\text{in}} = 64$$
$$h = w = 3$$

4,640 parameters with standard approach

16, 3x3 depth-wise kernels: 144

32, 1x1 point-wise filters: 512

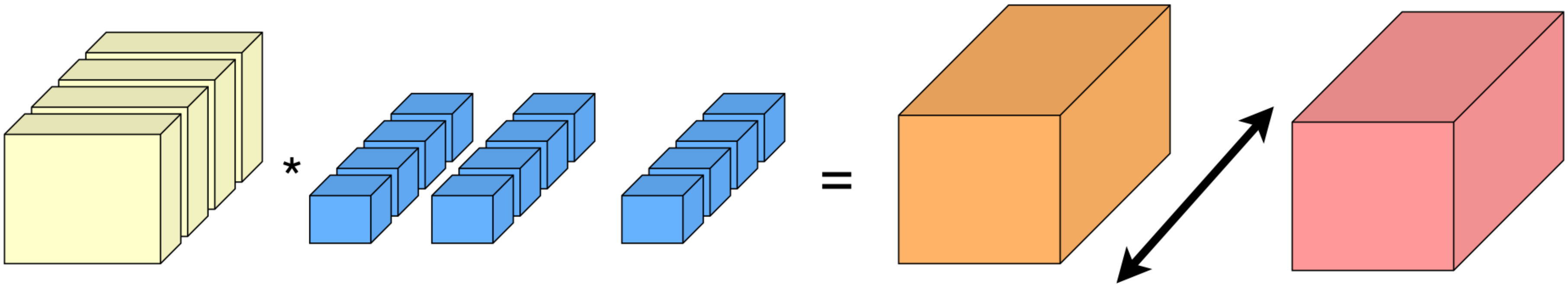
32, biases: 32

688 parameters for same output feature map size

Howard, Andrew G., et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications." arXiv preprint arXiv:1704.04861 (2017).

Sandler, Mark, et al. "Mobilenetv2: Inverted residuals and linear bottlenecks." Proceedings of the IEEE conference on computer vision and pattern recognition. 2018.

EXAMPLE: SHUFFLENET



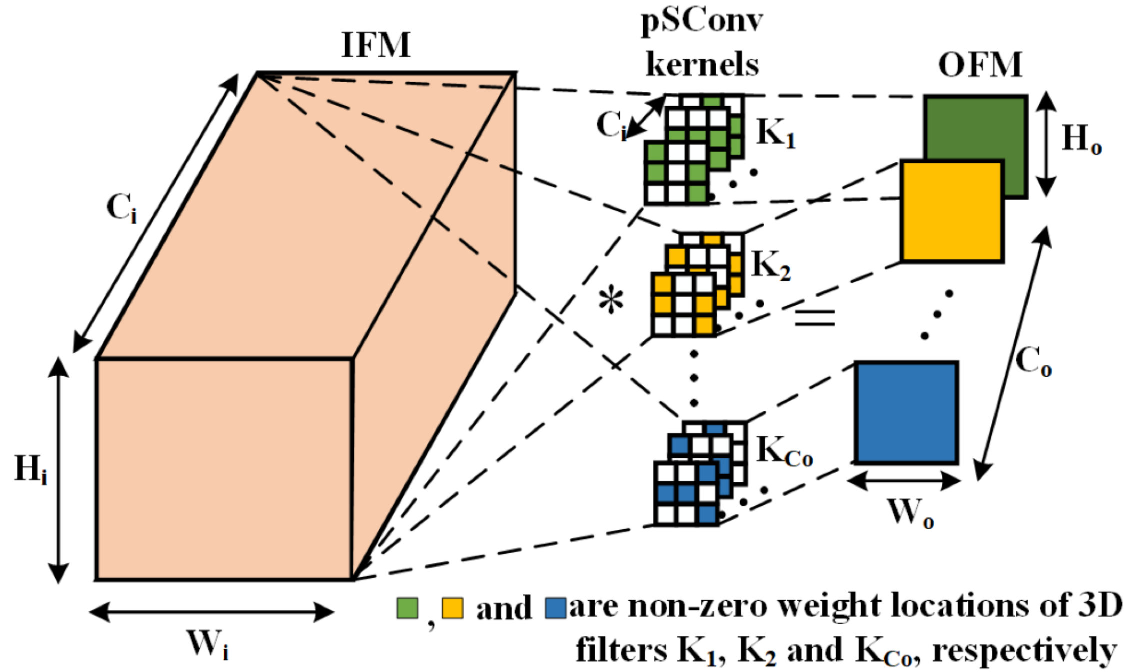
multiple grouped convolutions

shuffle across channels

group-wise convolutions with shuffling

Zhang, Xiangyu, et al. "Shufflenet: An extremely efficient convolutional neural network for mobile devices." Proceedings of the IEEE conference on computer vision and pattern recognition. 2018. APA

EXAMPLE: PRE-DEFINED SPARSITY



pre-define some of the filter coefficients to be zero and **hold fixed** through training and inference

targets specialized hardware acceleration — project concept is to map this to GPU

Kundu, Souvik, et al. "Pre-defined Sparsity for Low-Complexity Convolutional Neural Networks." IEEE Transactions on Computers (2020).

EXAMPLE: PRE-DEFINED SPARSITY

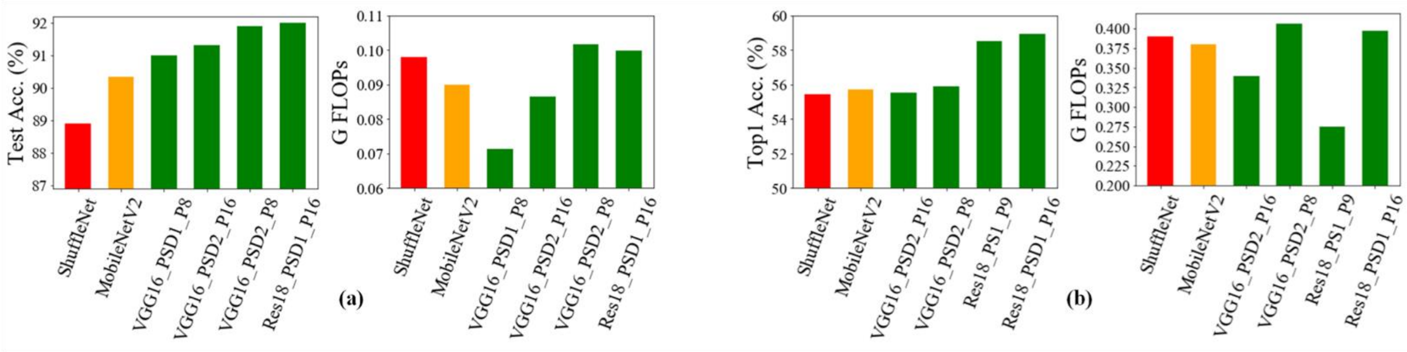


Fig. 11: Performance comparison of our proposed architectures that have similar or fewer FLOPs than ShuffleNet and MobileNetV2 with comparable or better classification accuracy on (a) CIFAR-10 and (b) Tiny ImageNet.

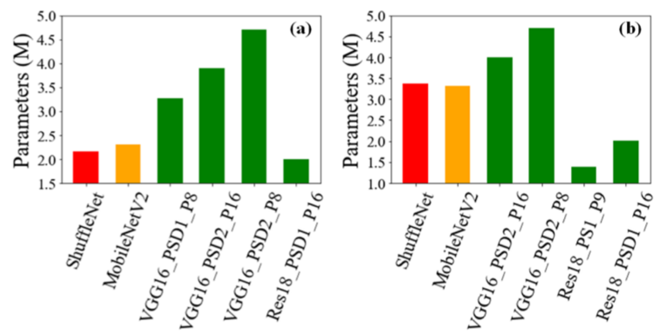


Fig. 12: Comparison of the number of model parameters of the network models described in Fig 11 for (a) CIFAR-10 and (b) Tiny ImageNet datasets.

Kundu, Souvik, et al. "Pre-defined Sparsity for Low-Complexity Convolutional Neural Networks." IEEE Transactions on Computers (2020).

POST-TRAINING APPROACHES

post-training processing to minimize complexity

Pruning: set near-zero weights to zero, fix these and do some retraining

Yang, Tien-Ju, Yu-Hsin Chen, and Vivienne Sze. "Designing energy-efficient convolutional neural networks using energy-aware pruning." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2017.

Quantization: map similar valued weights to the same value to save storage

Zhou, Aojun, et al. "Incremental network quantization: Towards lossless CNNs with low-precision weights." arXiv preprint arXiv:1702.03044 (2017).

Binaryization: find a set of binary weights that best approximate the trained network behavior

Rastegari, Mohammad, et al. "Xnor-net: Imagenet classification using binary convolutional neural networks." European conference on computer vision. Springer, Cham, 2016.

OUTLINE FOR SLIDES

- Motivation, applications
- Basic 2D convolution operations
  - PyTorch 2Dconv layer
- Pooling and stride
- Fashion MNIST example
- Visualization methods
- Some common CNN structures
- Reduced complexity CNN architectures
- Back-propagation for CNNs

CNN BACK PROPAGATION

BACK-PROPAGATION IN CNNS

recall the definition of a standard Conv2D operation:

$$y[i,j,k] = \sum_{c} \sum_{(m,n)} h_{c,k}[m,n]x[i+m, j+n, c]$$

$$h_{c,k}[m,n] = \text{2D kernel for input channel } c, \text{output channel } k$$

chain rule:

$$\frac{\partial C}{\partial x[i,j,k]} = \sum_{(i',j',k')} \frac{\partial y[i',j',k']}{\partial x[i,j,k]} \frac{\partial C}{\partial y[i',j',k']}$$

which values of h are involved here?

shorthand:

$$\partial_v[i,j,k] \triangleq \frac{\partial C}{\partial v[i,j,k]}$$

$$\partial_x[i,j,k] \triangleq \sum_{(i',j',k')} \boxed{\frac{\partial y[i',j',k']}{\partial x[i,j,k]}} \delta_y[i',j',k']$$

USC Viterbi School of Engineering

BACK-PROPAGATION IN CNNS

Let's start with the 2D convolution only…

$$y[i',j'] = \sum_{(m,n)} h[m,n]x[i'+m, j'+n]$$

$$= \sum_{(s,t)} h[s-i', t-j']x[s,t]$$

$s = i' + m$

$t = j' + n$

$$\delta_x[i,j] = \sum_{(i',j')} \frac{\partial y[i',j']}{\partial x[i,j]} \delta_y[i',j']$$

chain-rule term:

$$\frac{\partial y[i',j']}{\partial x[i,j]} = h[i-i', j-j']$$

$$\delta_x[i,j] = \sum_{(i',j')} h[i-i', j-j']x[i',j']$$

$$= \sum_{(m,n)} h[-m,-n]\delta_y[i+m, j+n]$$

$m = i' - i$

$n = j' - j$

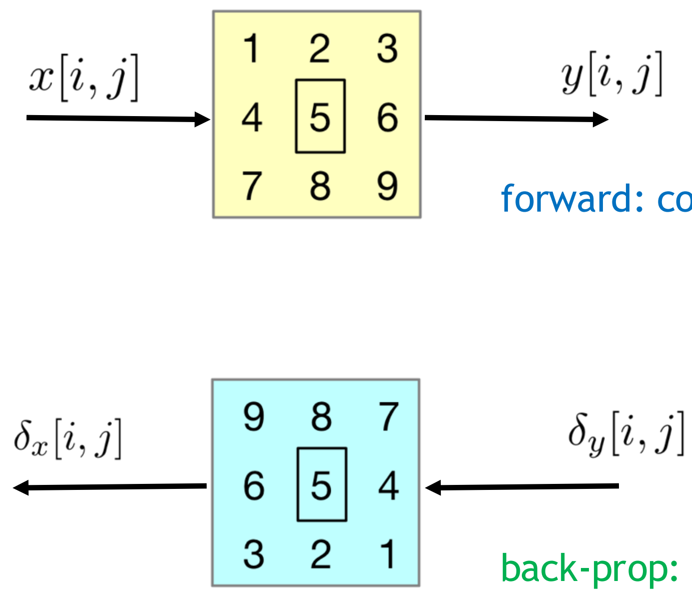
BACK-PROPAGATION IN CNNS

$$y[i,j] = \sum_{(m,n)} h[m,n]x[i+m, j+n]$$

forward: convolve with $h[i,j]$

$$\delta_x[i,j] = \sum_{(m,n)} h[-m,-n]\delta_y[i+m, j+n]$$

back-prop: convolve with $h[-i,-j]$



forward: convolve with $h[i,j]$

back-prop: convolve with $h[-i,-j]$

recall: W-transpose in MLP-BP

$$\boldsymbol{\delta}^{(l)} = \dot{\mathbf{a}}^{(l)} \left[\left(\mathbf{W}^{(l+1)}\right)^T \boldsymbol{\delta}^{(l+1)} \right]$$

BACK-PROPAGATION IN CNNS

this extends to the standard Conv2D convolution

$$y[i',j',k'] = \sum_{k} \sum_{(m,n)} h_{k,k'}[m,n]x[i'+m,j'+n,k]$$

$$\delta_x[i,j,k] = \sum_{(i',j',k')} \frac{\partial y[i',j',k']}{\partial x[i,j,k]} \delta_y[i',j',k']$$

$$i = i' + m$$
$$j = j' + n$$

$$\frac{\partial y[i',j',k']}{\partial x[i,j,k]} = h_{k,k'}[i-i',j-j']$$

$$\boxed{\begin{aligned}\delta_x[i,j,k] &= \sum_{(i',j',k')} h_{k,k'}[i-i',j-j']\delta_y[i',j',k'] \\ &= \sum_{(m,n,k')} h_{k,k'}[-m,-n]\delta_y[i+m,j+n,k']\end{aligned}}$$

standard 2DConv with
reflected 2D kernels

$$m = i' - i$$
$$n = j' - j$$

BACK-PROPAGATION IN CNNS: POOLING

average pooling:

results from standard differentiation

forward: Q "pixels" averaged

back-prop: $1/Q$ times the gradient flows back through theses Q "pixels"

max pooling:

non-differentiable.... just a convention that works!

forward: max over Q "pixels" $(i^*, j^*) \sim \mathrm{argmax}$

back-prop: gradient flows directly through (i^*, j^*) only

CNN/CV RELATED TOPICS

Image segmentation (*e.g.*, U-Net)

Object Detection (*e.g.*, YOLO)

GANs (*e.g.*, “deep fakes”)