Unit 9
Convolutional and Deep Neural Networks

Prof. Phil Schniter

THE OHIO STATE UNIVERSITY

ECE 5307: Introduction to Machine Learning, Sp23

Learning objectives

- Recognize CNNs as learning and exploiting hierarchical patterns
- 2D convolution:
  - Understand as local pattern matching
  - Understand boundary conditions and relation between convolution and correlation
  - Know how to implement convolution with scipy.signal and PyTorch
- Convolutional neural networks:
  - Understand convolutional layers, dense layers, subsampling, pooling
  - Understand backpropagation training
  - Recognize training tricks like batch-norm, dropout, data augmentation
- Transfer learning and pre-trained networks:
  - Be familiar with AlexNet, VGG, Inception, ResNet, and other famous networks
  - Know how to work with pre-trained networks in PyTorch

Outline

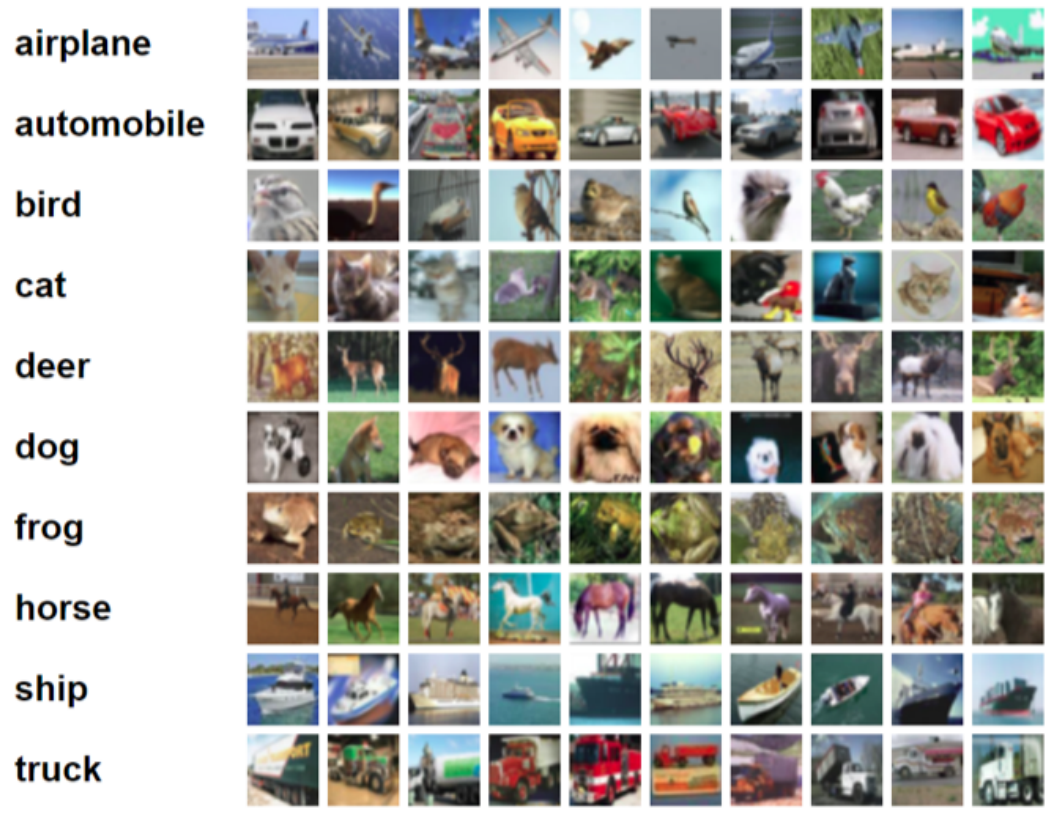
- Motivation: ImageNet Large-Scale Visual Recognition Challenge
- Deep Networks and Feature Hierarchies
- 2D Convolution Basics
- Convolutional Neural Networks
- Creating and Visualizing Convolutional Layers in PyTorch
- Training CNNs: Backpropagation, Batch-Norm, Dropout, Etc.
- Transfer Learning from Famous Pre-Trained Networks

Large-scale image classification

Before 2009, image recognition worked on small datasets. For example . . .

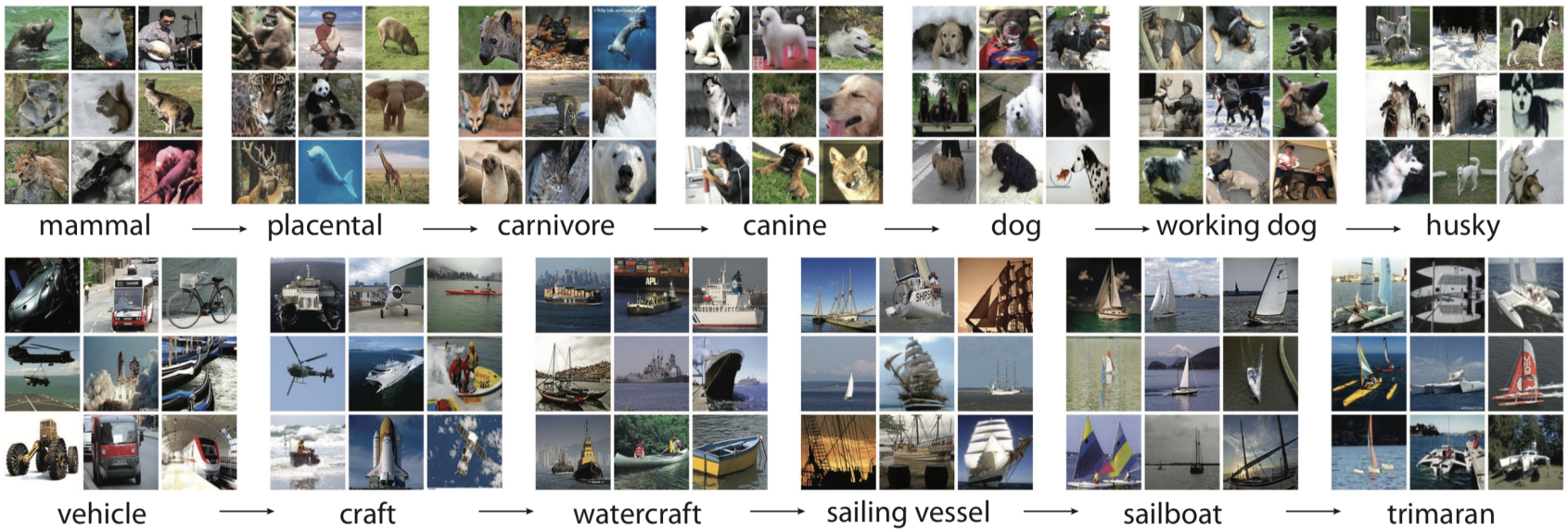
- MNIST:
  - 70 000 examples
  - 10 classes
  - 28×28 images
- CIFAR 10:
  - 60 000 examples
  - 10 classes
  - 32×32 images
- PASCAL VOC:
  - 11 530 examples
  - 20 classes
  - variable-size images

https://www.cs.toronto.edu/~kriz/cifar.html



ImageNet (2009)

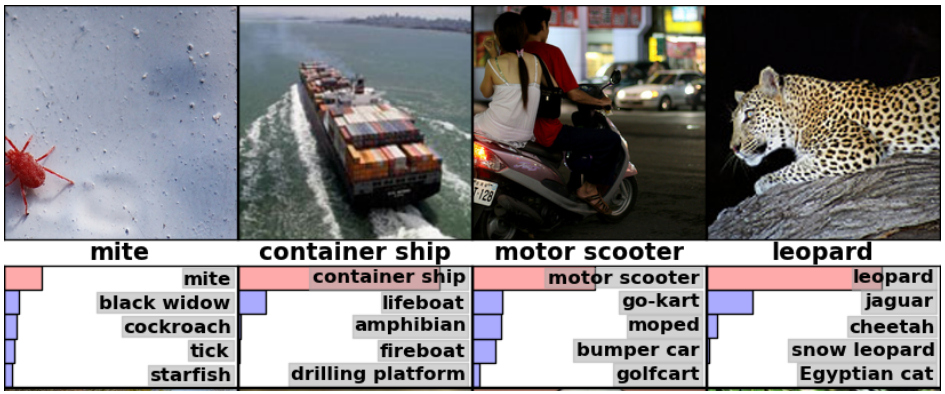
- Idea: To develop better algorithms, collect more data!
- ImageNet:
  - Goal: "map out the entire world of objects" (see this great article)
  - 3.2 million images and 5 247 classes (hierarchical, based on WordNet)
  - images labeled using Amazon's Mechanical Turk
  - detailed in this 2009 CVPR paper

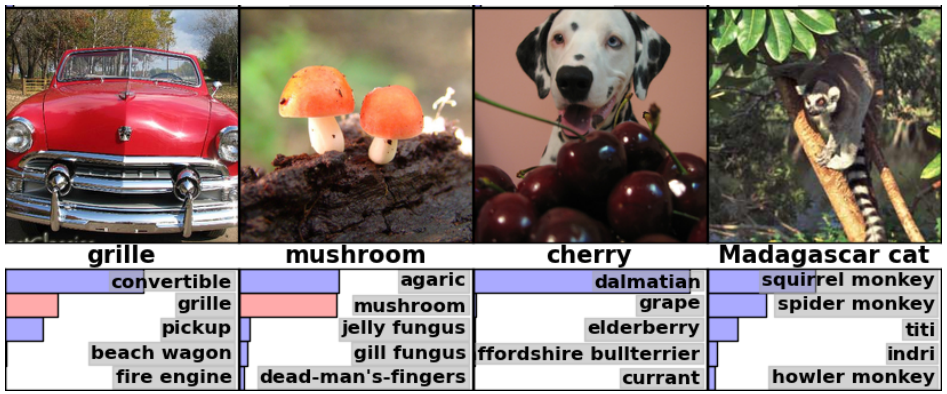


ImageNet Large-Scale Visual Recognition Challenge

- ILSVRC: Competition on ImageNet dataset held yearly from 2010–2017
- Challenging! For example, in 2012 . . .
  1. Classification: 1000 classes, list 5 most likely per image
  2. Fine-grained classification: 120 dog categories
  3. Classification with localization: list 5 most likely with bounding boxes

Note that images can contain multiple objects, each with different scales, rotations, lightings, occlusions:



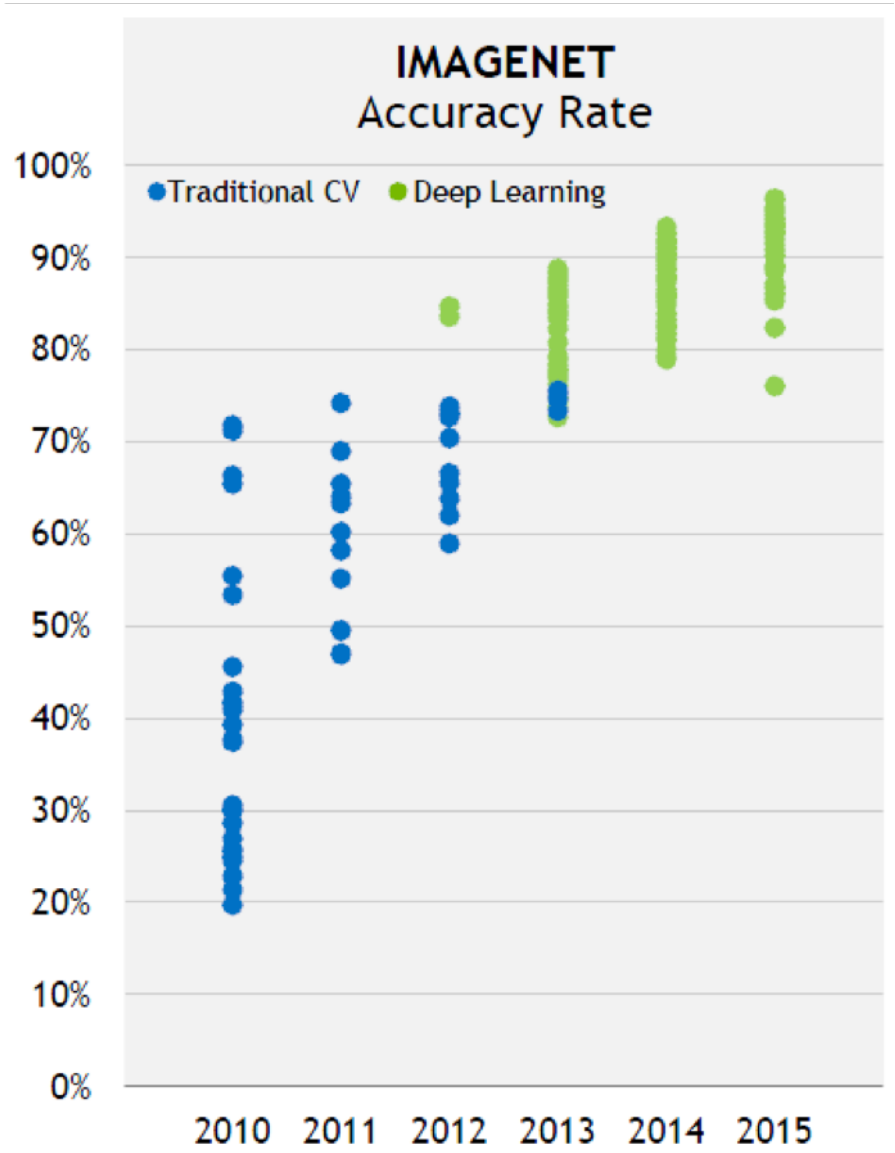


Deep neural networks enter in 2012

Breakthrough by first deep net in 2012

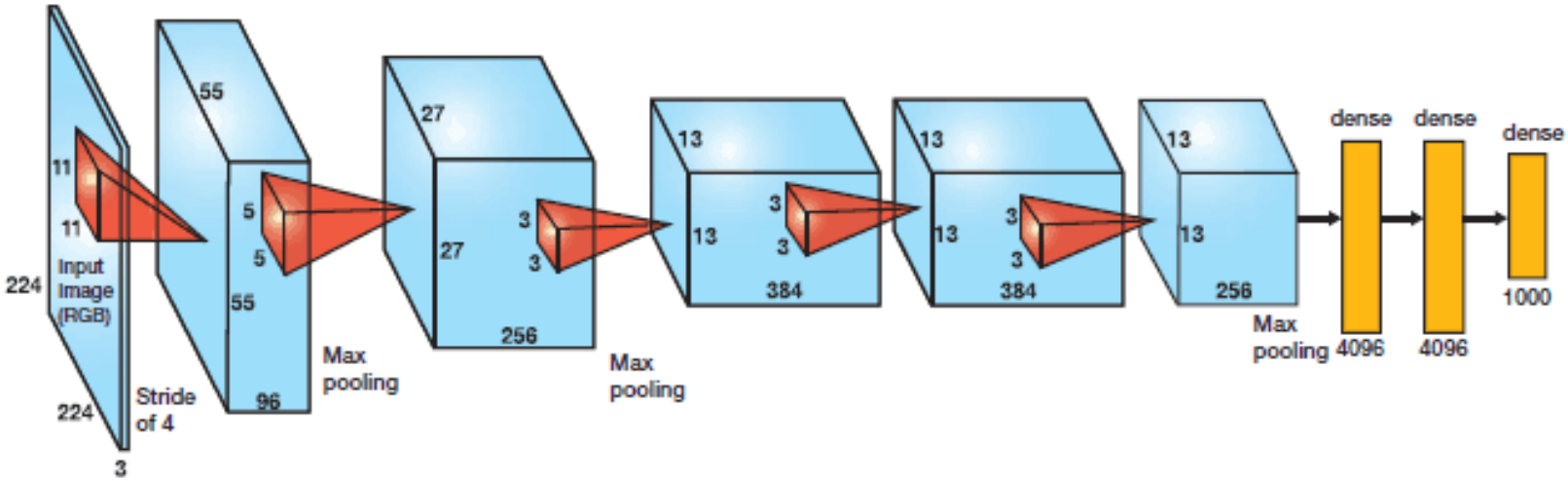
- "AlexNet" by Hinton @ U Toronto
- Easily won the ILSVRC competition
  - Top-5 error rate was 15.3%
  - 2nd place error rate was 25.6%

Soon, all methods were deep nets!



AlexNet

- Main idea: Build a very deep network (very deep at the time)
- 5 convolutional layers + 3 fully-connected layers
- Final layer is 1000-class softmax
- Various other tricks (new at the time): ReLU, max pooling, dropout
- 60 million parameters, 650 000 neurons (or "feature maps")



Outline

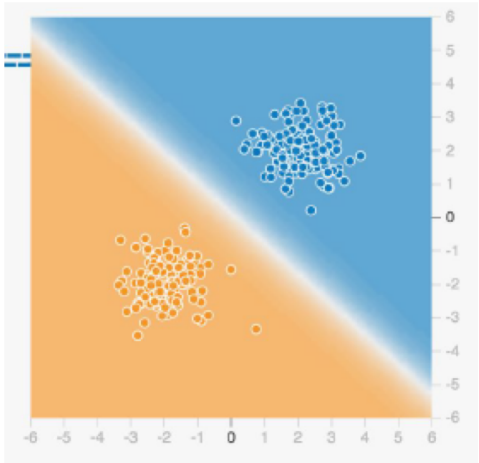
- Motivation: ImageNet Large-Scale Visual Recognition Challenge
- Deep Networks and Feature Hierarchies
- 2D Convolution Basics
- Convolutional Neural Networks
- Creating and Visualizing Convolutional Layers in PyTorch
- Training CNNs: Backpropagation, Batch-Norm, Dropout, Etc.
- Transfer Learning from Famous Pre-Trained Networks

Review of neural networks: Hidden layer

- Recall that each hidden unit in a 2-layer neural net computes an activation

$$a = h_{\mathsf{H}}(z) \quad \text{from the linear score} \quad z = \boldsymbol{w}^{\mathsf{T}}\boldsymbol{x} + b$$

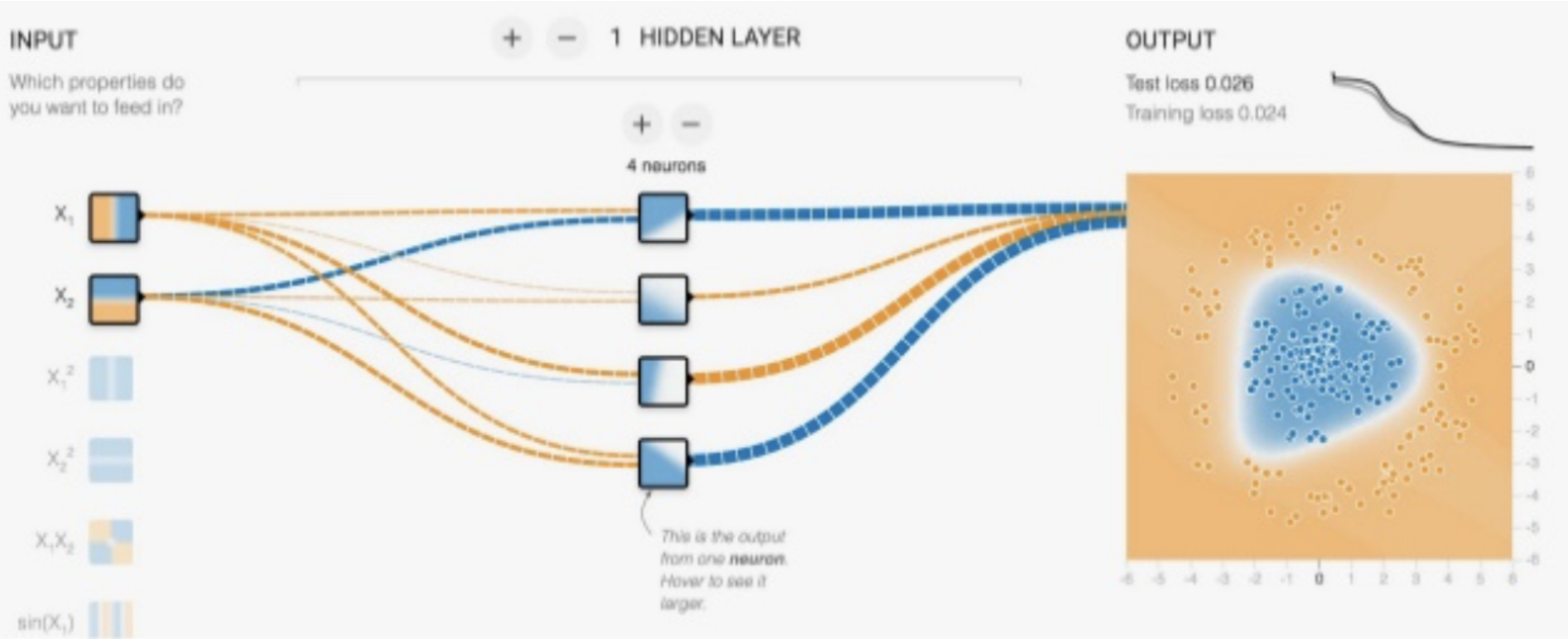
- With sigmoidal $h_{\mathsf{H}}(z) = \frac{1}{1+\exp(-z)}$, each unit partitions in the input space into two half-spaces
  - The half-spaces are linearly separated
  - The boundary is perpendicular to $\boldsymbol{w}$ and shifted by b
  - The picture on right shows the $d = 2$ case, $\boldsymbol{x} = [x_1, x_2]^{\mathsf{T}}$



- We say that "a is tuned to $\boldsymbol{w}$" since a is largest when $\boldsymbol{x}$ is colinear to $\boldsymbol{w}$
  - a is largest when z is largest
  - z is largest when $\boldsymbol{x}$ is colinear to $\boldsymbol{w}$
    - Cauchy-Schwarz: $\boldsymbol{w}^{\mathsf{T}}\boldsymbol{x} \leq \|\boldsymbol{w}\|\|\boldsymbol{x}\|$ with equality when $\boldsymbol{x} = C\boldsymbol{w}$ for any $C > 0$

Review of neural networks: Output layer

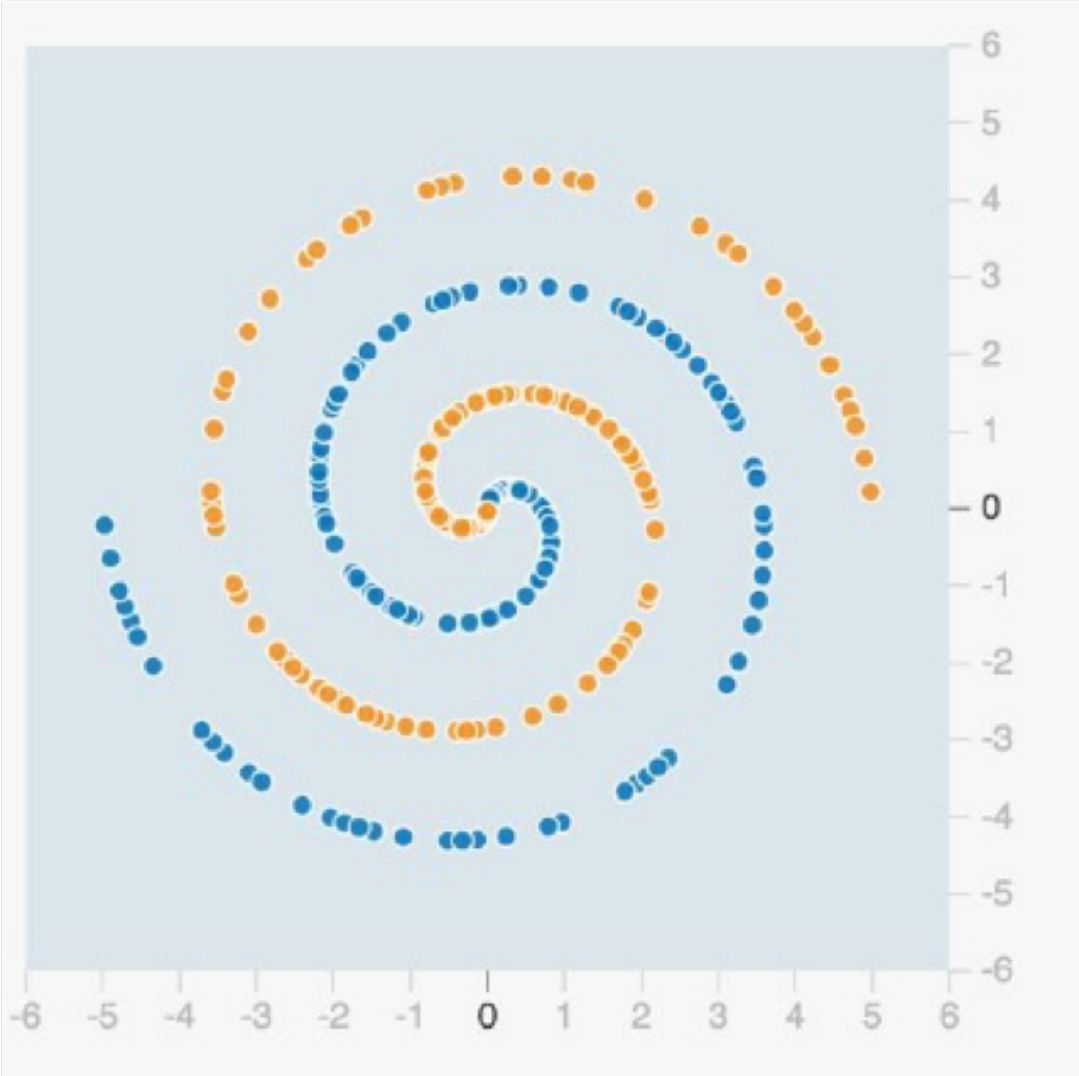
- 2-layer nets implement *nonlinear* boundaries using *intersections* of half-spaces
- Below is a visualization from TensorBoard (via TensorBoardX in PyTorch)



https://www.slideshare.net/HadoopSummit/google-cloud-platform-empowers-tensorflow-and-machine-learning

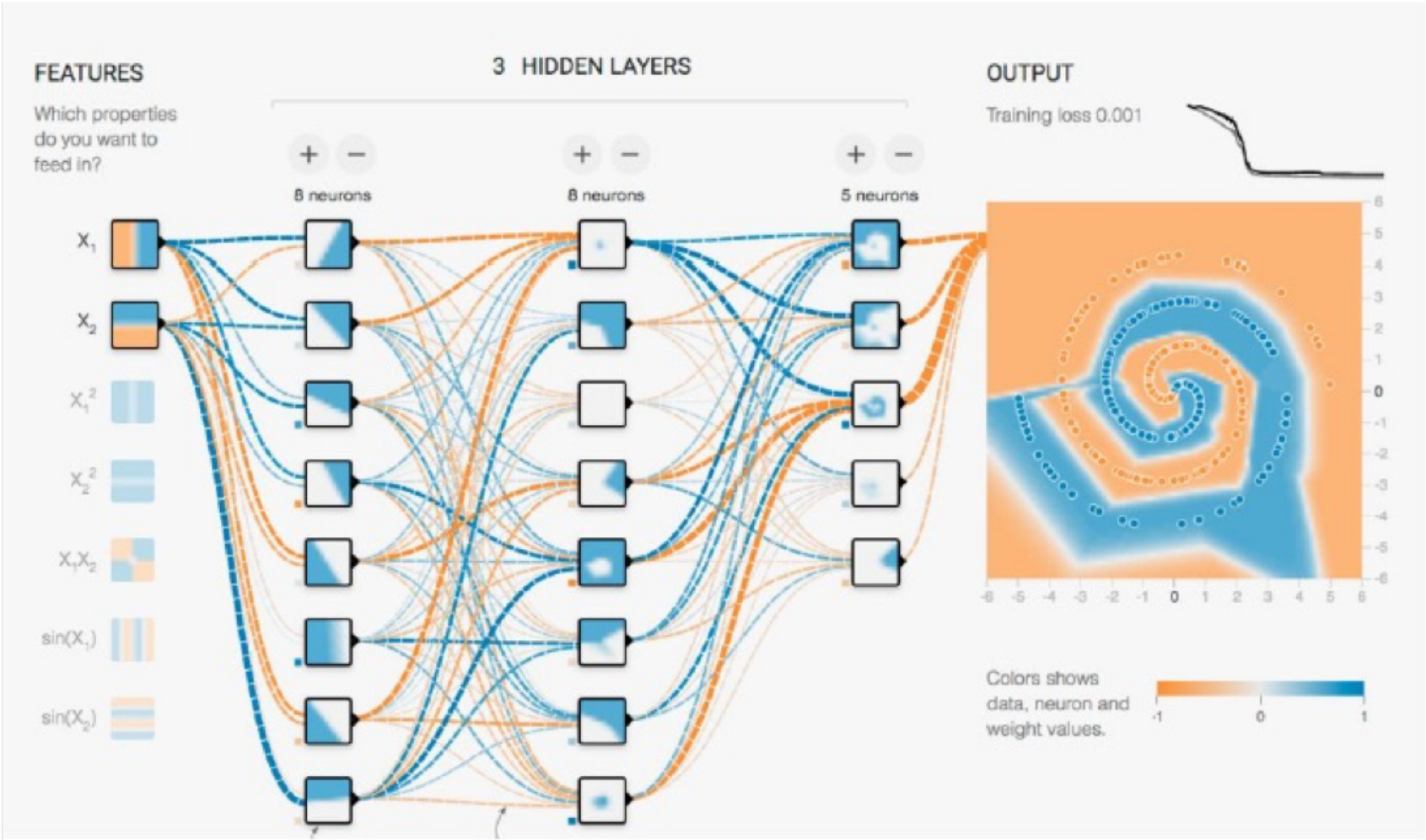
What about a more complicated region?

- Can we handle more complicated regions?
- Yes! Using more neurons and more layers



Multi-layer networks

- With 4 layers, we build hierarchies of transformed patterns



https://www.slideshare.net/HadoopSummit/google-cloud-platform-empowers-tensorflow-and-machine-learning

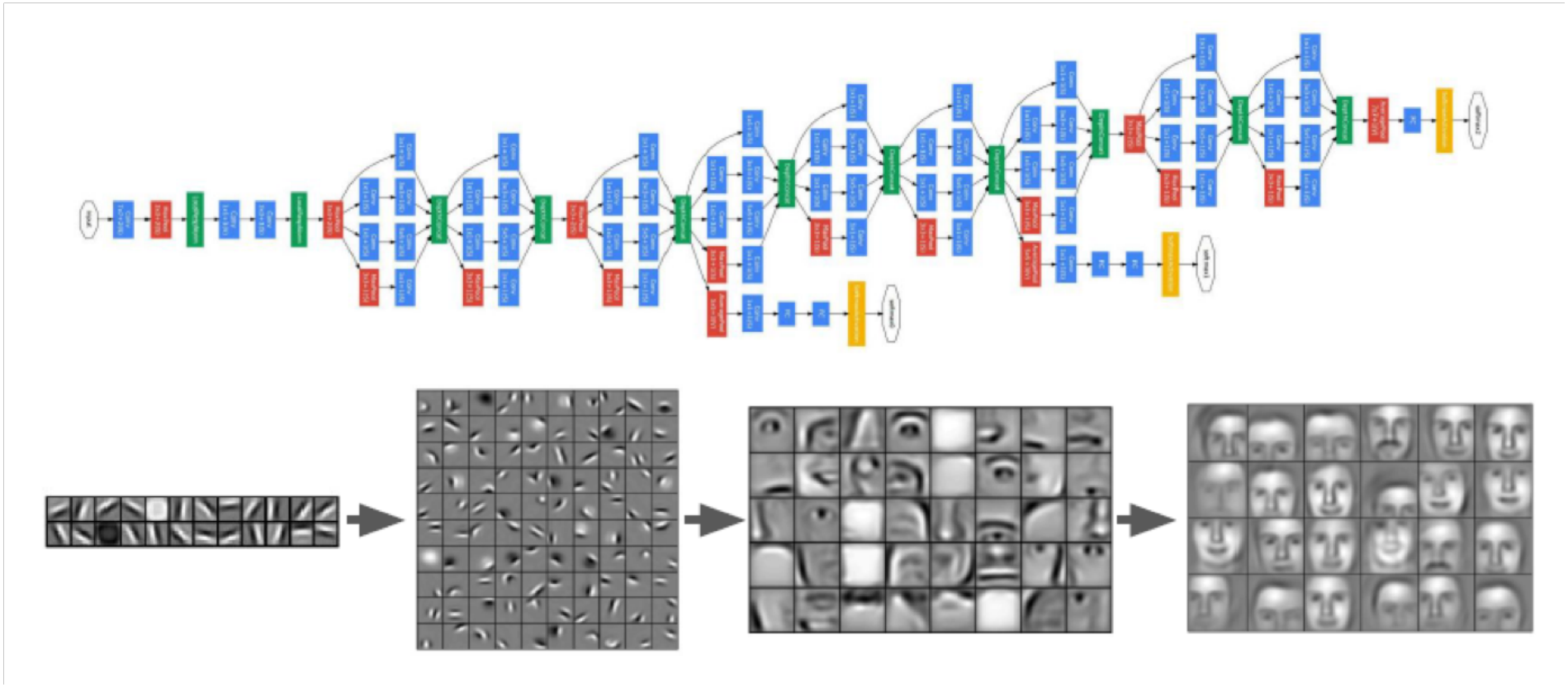
But how do we classify...

versus

?

Deep networks

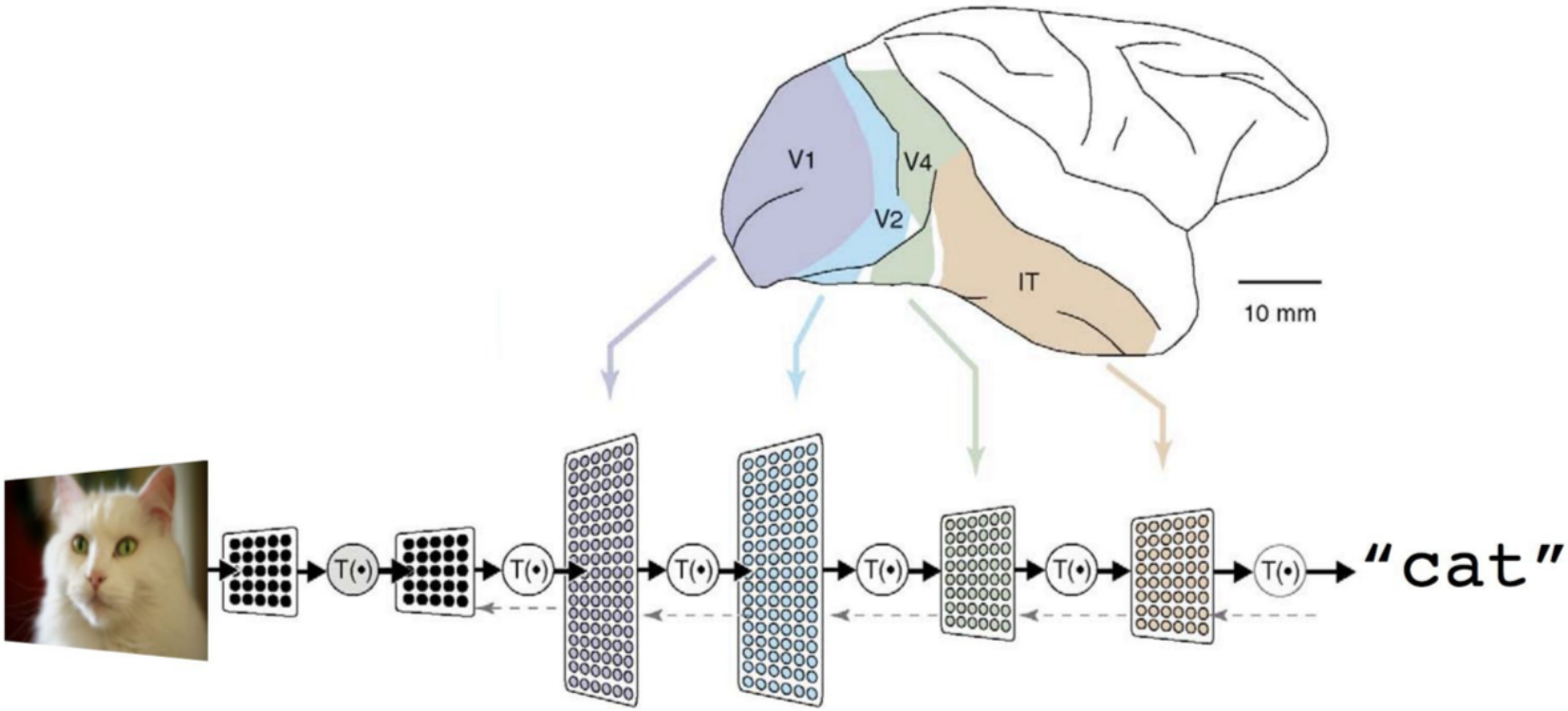
- With deep networks, we can build complex hierarchies!



Honglak Lee, et al, "Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations"

Inspiration from biology

- The brain uses multi-layer processing



https://www.slideshare.net/HadoopSummit/google-cloud-platform-empowers-tensorflow-and-machine-learning

History of deep learning

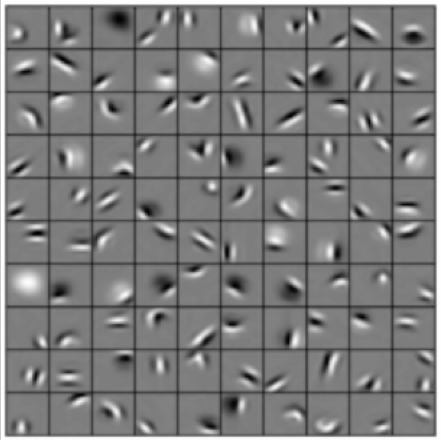
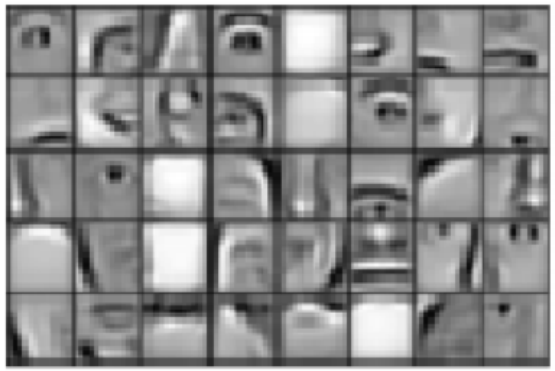
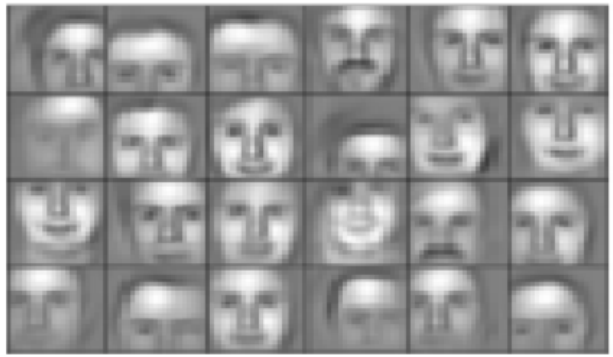
- Early history:
  - first deep network: Ivakhnenko 1965
  - first convolutional deep network: Fukushima 1980
  - first backpropagation in deep networks: Werbos 1982, LeCun 1989
- Challenges in early days:
  - limited training data (many parameters $\Rightarrow$ overfitting)
  - limited computational power
  - the vanishing gradient problem
    - Sigmoid or tanh activation fxns yield per-layer gradients in open interval $(0, 1)$
    - So, as we concatenate many layers, the gradients from early layers become very weak!
- Typical components of modern deep nets:
  - ReLU activation (has gradient values equal to exactly 0 or 1)
  - convolutional layers and pooling layers
  - tricks like batch-norm, dropout, skip connections, etc

Outline

- Motivation: ImageNet Large-Scale Visual Recognition Challenge
- Deep Networks and Feature Hierarchies
- 2D Convolution Basics
- Convolutional Neural Networks
- Creating and Visualizing Convolutional Layers in PyTorch
- Training CNNs: Backpropagation, Batch-Norm, Dropout, Etc.
- Transfer Learning from Famous Pre-Trained Networks

Local patterns

- Early network layers typically find local patterns
  - small patterns within a larger image
  - examples: line segments, curves, bumps
- Subsequent layers combine those local patterns to build more complex patterns

- The local patterns can be located anywhere in the image
  - such patterns are called "translation invariant"
- How do we find them?

Pattern matching

- Q: How do we find a local pattern?
- A: Shift a template or "kernel" around, and take an inner product at each shift
- Concretely: Say we want to find the pattern $\boldsymbol{W} \in \mathbb{R}^{K_1 \times K_2}$ in a larger image $\boldsymbol{X} \in \mathbb{R}^{N_1 \times N_2}$. Then, at each offset (j_1, j_2), compute the correlation

$$Z[j_1, j_2] = \sum_{k_1=0}^{K_1-1} \sum_{k_2=0}^{K_2-1} W[k_1, k_2] X[j_1 + k_1, j_2 + k_2]$$

- If the pattern (or similar) is present at (j_1, j_2), then $Z[j_1, j_2]$ should be large

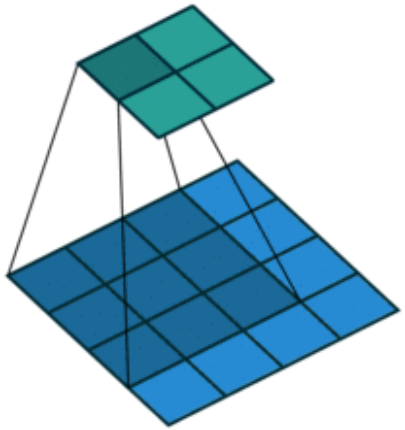
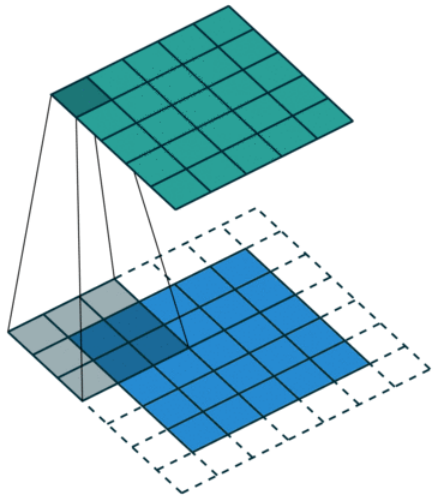
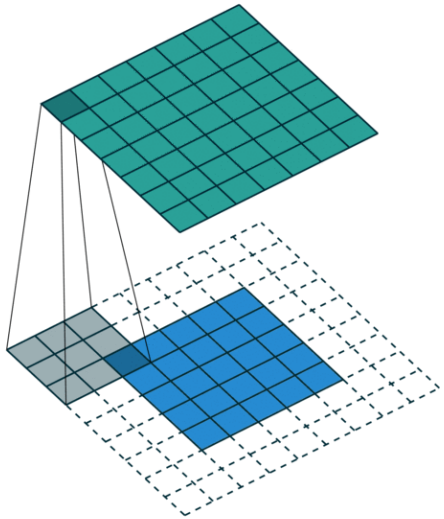
Terminology

- In digital signal processing . . .
  - 2D correlation is $Z[j_1, j_2] = \sum_{k_1=0}^{K_1-1} \sum_{k_2=0}^{K_2-1} W[k_1, k_2] X[j_1+k_1, j_2+k_2]$
    - implemented in `scipy.signal.correlate2d`
    - sometimes called "convolution without flipping/reversal"
  - 2D convolution is $Z[j_1, j_2] = \sum_{k_1=0}^{K_1-1} \sum_{k_2=0}^{K_2-1} W[k_1, k_2] X[j_1-k_1, j_2-k_2] = (W * X)[j_1, j_2]$
    - implemented in `scipy.signal.convolve2d`
    - sometimes called "convolution with flipping/reversal"

- But in most neural-net packages (e.g., PyTorch) . . .
  - "convolution" does not include flipping/reversal! (see torch.nn.Conv2d)
  - so "convolution" actually means correlation . . . keep this in mind!
- In either case, the set of convolution weights $\boldsymbol{W}$ is called the "kernel"

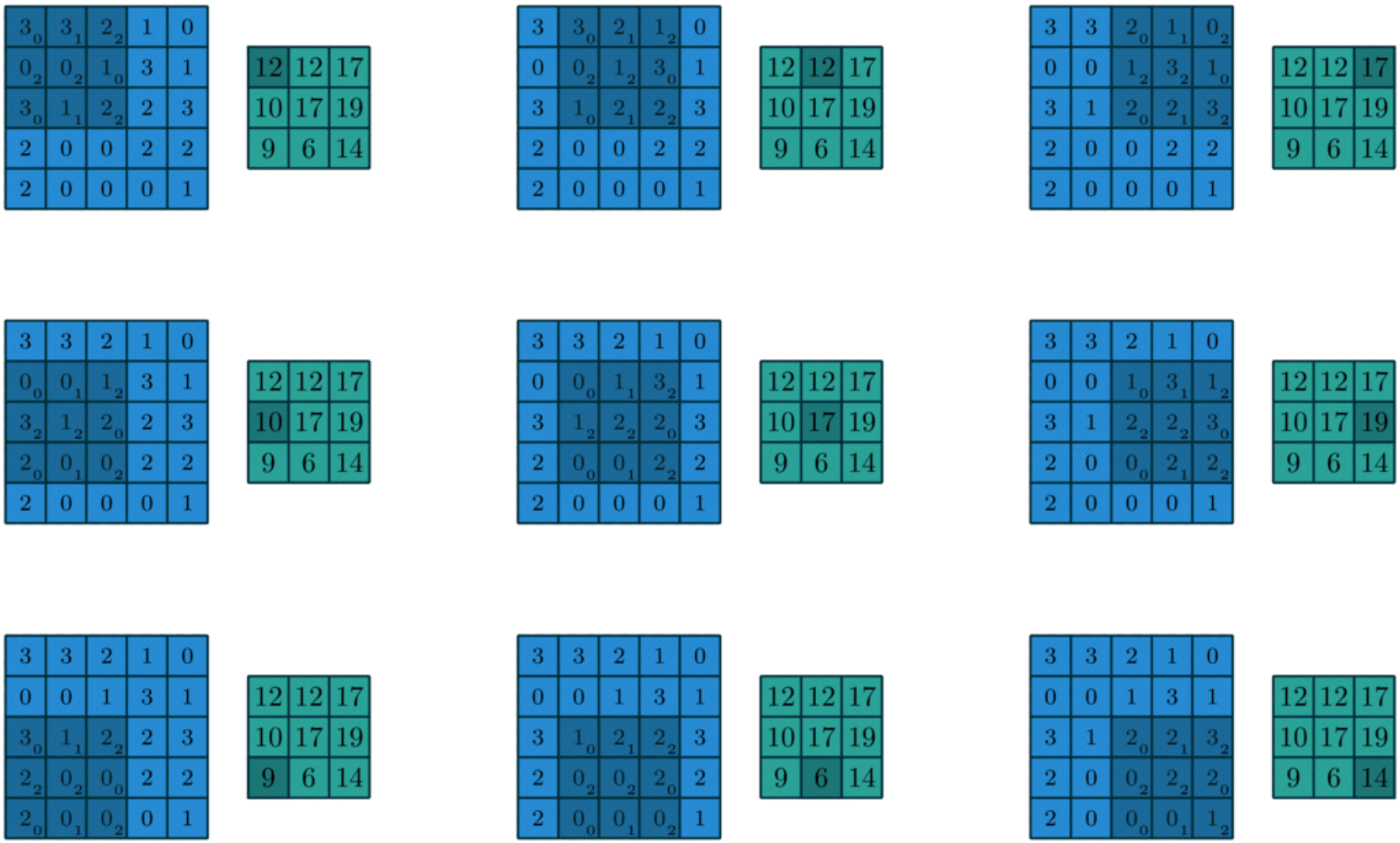
Types of zero padding in convolution

- There are three common ways to handle the boundary:

"valid" mode	"same" mode	"full" mode
(no zero-padding)	(partial zero-padding)	(full zero-padding)
		
output size:	output size:	output size:
$(N_1-K_1+1) \times (N_2-K_2+1)$	$N_1 \times N_2$	$(N_1+K_1-1) \times (N_2+K_2-1)$

- There's a good deep-net convolution tutorial here with animated gifs here

Numerical example of "valid" convolution



2D convolution in Python

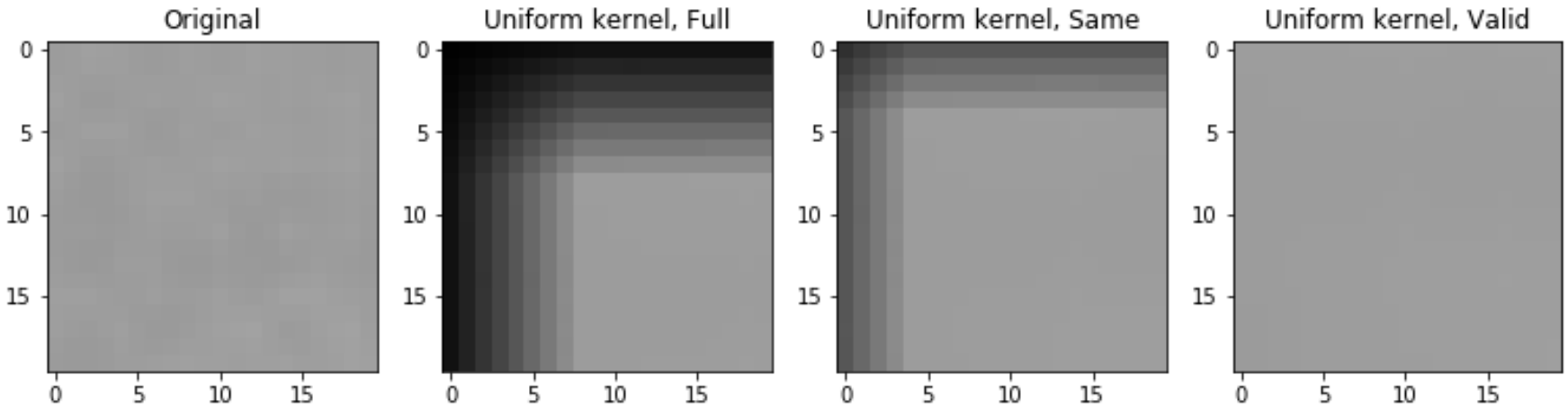
- Next we'll demonstrate 2D convolution in Python
- For this we'll use the packages:
  - `scipy.signal`: for signal processing
  - `skimage`: for image processing
- We'll experiment with the famous "cameraman" image, shown on the right
  - A built-in image in `skimage.data`
- We'll examine two applications of 2D convolution:
  - local averaging (i.e., blurring)
    - used for noise suppression
  - edge detection

```
im = skimage.data.camera()
disp_image(im)
```

Local averaging: Illustration of boundary effects

```
kx = 9
ky = 9
G_unif = np.ones((kx,ky))/(kx*ky)
im_unif_full = scipy.signal.convolve2d(im, G_unif, mode='full')
im_unif_same = scipy.signal.convolve2d(im, G_unif, mode='same')
im_unif_valid = scipy.signal.convolve2d(im, G_unif, mode='valid')
```

```
Input shape = (512, 512)
Output shape (full) = (520, 520)
Output shape (same) = (512, 512)
Output shape (valid) = (504, 504)
```



Local averaging: Effect of kernel size

```
kx = 9
ky = 9
G_unif = np.ones((kx,ky))/(kx*ky)
im_unif_9 = scipy.signal.convolve2d(im, G_unif, mode='same')

kx = 15
ky = 15
G_unif = np.ones((kx,ky))/(kx*ky)
im_unif_15 = scipy.signal.convolve2d(im, G_unif, mode='same')
```

Original

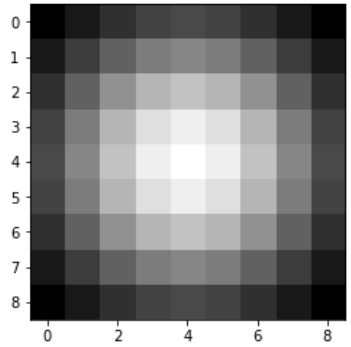
Uniform kernel, 9x9

Uniform kernel, 15x15

Local averaging: Effect of kernel type

- Previously we used a uniform kernel
- Now we try a Gaussian kernel
  - standard deviation σ controls effective width
  - should choose kernel size $K > 2\sigma + 1$
- Uniform & Gaussian cause different blurring

9x9 Gaussian blur kernel



Uniform kernel, 9x9

Gaussian kernel, 9x9

Uniform kernel, 15x15

Gaussian kernel, 15x15

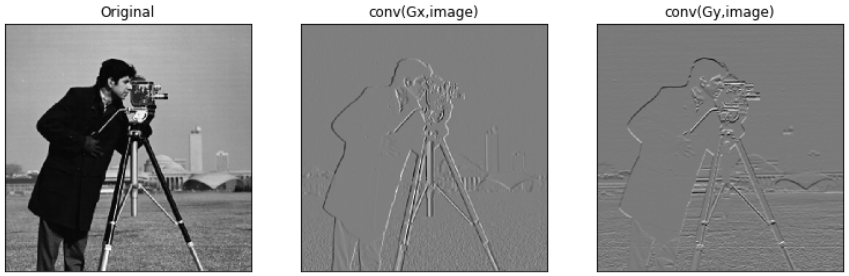
Edge detection via 2D convolution

- Can do edge detection by convolving an image with a gradient filter
- For example, use Sobel filters: $\boldsymbol{G}_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$, $\boldsymbol{G}_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$
- Say $\boldsymbol{Z}_x = \boldsymbol{G}_x * \boldsymbol{X}$ (convolution *with* flipping/reversal). Then we expect . . .
  - $Z_x[j_1, j_2] = 0$ in regions where $X[j_1, j_2]$ is constant
  - $Z_x[j_1, j_2] > 0$ with horizontal change from dark $\rightarrow$ light (vertical edge)
  - $Z_x[j_1, j_2] < 0$ with horizontal change from light $\rightarrow$ dark (vertical edge)
- Say $\boldsymbol{Z}_y = \boldsymbol{G}_y * \boldsymbol{X}$ (convolution *with* flipping/reversal). Then we expect . . .
  - $Z_y[j_1, j_2] = 0$ in regions where $X[j_1, j_2]$ is constant
  - $Z_y[j_1, j_2] > 0$ with vertical change from dark $\downarrow$ light (horizontal edge)
  - $Z_y[j_1, j_2] < 0$ with vertical change from light $\downarrow$ dark (horizontal edge)

Edge detection via 2D convolution

Convolution with flipping/reversal:

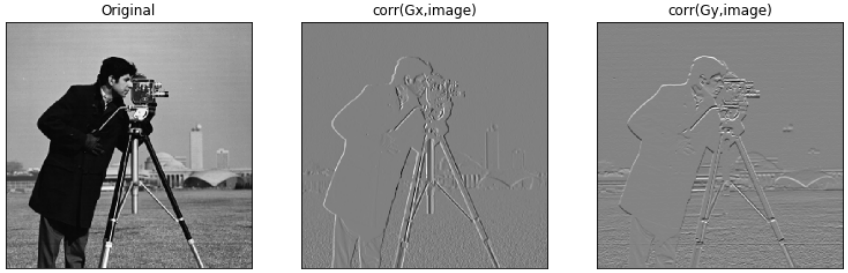
```
Gx = np.array([[1,0,-1],[2,0,-2],[1,0,-1]])  # Grad
Gy = np.array([[1,2,1],[0,0,0],[-1,-2,-1]])  # Grad

imx = scipy.signal.convolve2d(im, Gx, mode='valid')
imy = scipy.signal.convolve2d(im, Gy, mode='valid')
```



- works as expected

Correlation:

```
Gx = np.array([[1,0,-1],[2,0,-2],[1,0,-1]])  # Gradi
Gy = np.array([[1,2,1],[0,0,0],[-1,-2,-1]])  # Gradi

imx = scipy.signal.correlate2d(im, Gx, mode='valid')
imy = scipy.signal.correlate2d(im, Gy, mode='valid')
```



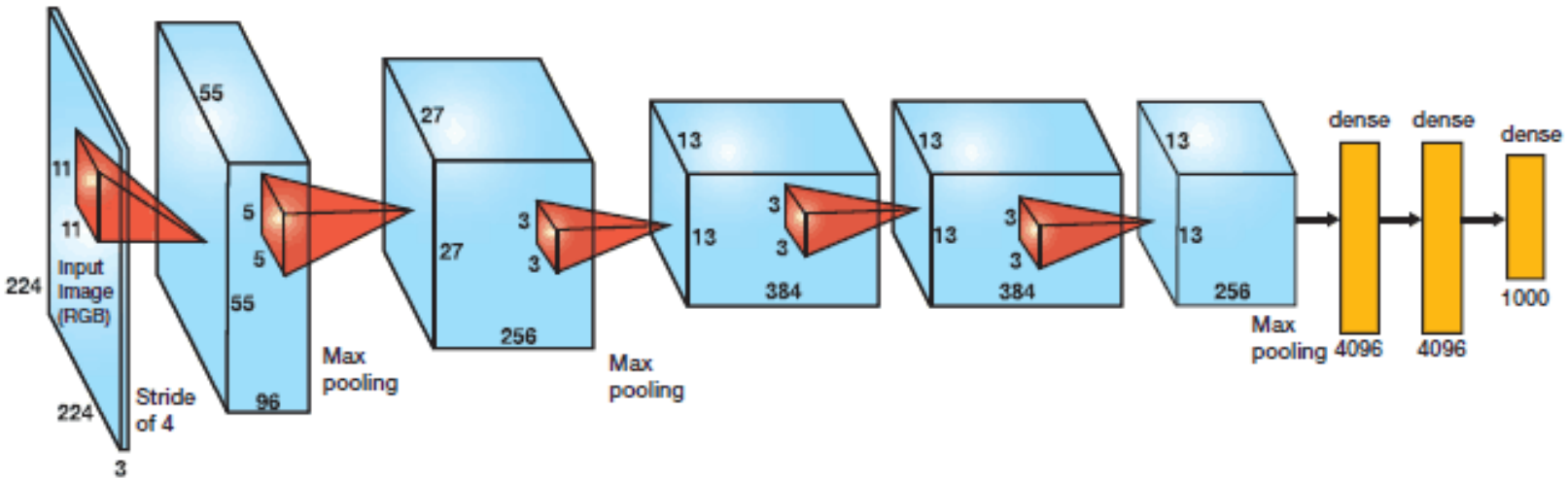
- works as expected: $Z[j_1, j_2]$ changes sign relative to convolution with flipping/reversal

Outline

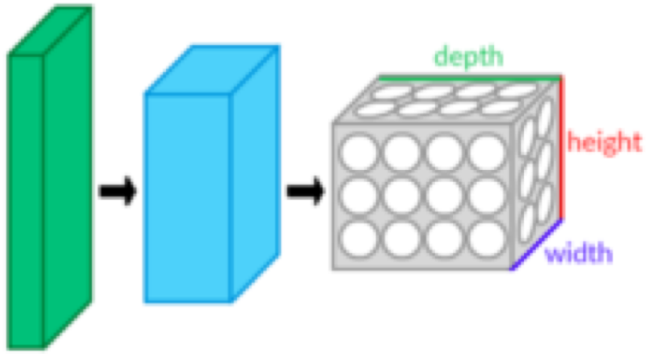
- Motivation: ImageNet Large-Scale Visual Recognition Challenge
- Deep Networks and Feature Hierarchies
- 2D Convolution Basics
- Convolutional Neural Networks
- Creating and Visualizing Convolutional Layers in PyTorch
- Training CNNs: Backpropagation, Batch-Norm, Dropout, Etc.
- Transfer Learning from Famous Pre-Trained Networks

Typical CNN structure

- Starts with several convolutional layers. Each layer does ...
  1. 2D convolution with several kernels, whose outputs are then linearly combined
  2. activation (e.g., ReLU)
  3. sub-sampling or pooling
- Finishes with several fully connected (or dense) layers. Each layer does ...
  1. matrix multiplication
  2. activation
- For example, AlexNet does ...



Tensors



- The input and output of each convolutional layer is a tensor
- A tensor is simply a multi-dimensional array.
- Examples of tensors:
  - grayscale image: [height, width]
  - color image: [height, width, channel] where channel is in {R,G,B} ← skimage
  - batch of images: [index, channel, height, width] ← PyTorch convention
- The number of dimensions in a tensor is called the degree or order or rank
  - Note that "rank" has a very different meaning for matrices!
- The shape of a tensor specifies the number of elements along each dimension
  - A $10 \times 20 \times 4$ tensor has shape $(10, 20, 4)$ and degree 3
- Tensors are basic datatypes in Python, e.g., `np.ndarray` and `torch.Tensor`

What convolutional layers do

- Each convolutional layer has
  - a weight tensor $\boldsymbol{W}$ with shape $(C_{\text{out}}, C_{\text{in}}, K_1, K_2)$,
  - an intercept or "bias" vector $\boldsymbol{b}$ with C_{out} elements
- Given an input tensor $\boldsymbol{A}$ with shape $(B, C_{\text{in}}, N_1, N_2)$, the convolutional layer computes an output tensor $\boldsymbol{Z}$ with elements

$$Z[i,m,j_1,j_2] = \sum_{n=1}^{C_{\text{in}}} \sum_{k_1=0}^{K_1-1} \sum_{k_2=0}^{K_2-1} W[m,n,k_1,k_2] A[i,n,j_1+k_1,j_2+k_2] + b[m]$$

$$\text{for } i = 1 \ldots B \text{ and } m = 1 \ldots C_{\text{out}}$$

  - All batch elements i are processed separately as follows:
  - For each output channel m: we correlate each input channel n using the $K_1 \times K_2$ kernel $W[m,n,:,:]$ and sum the result across input channels n
  - See visualization here
- Note: The above description uses PyTorch's ordering conventions

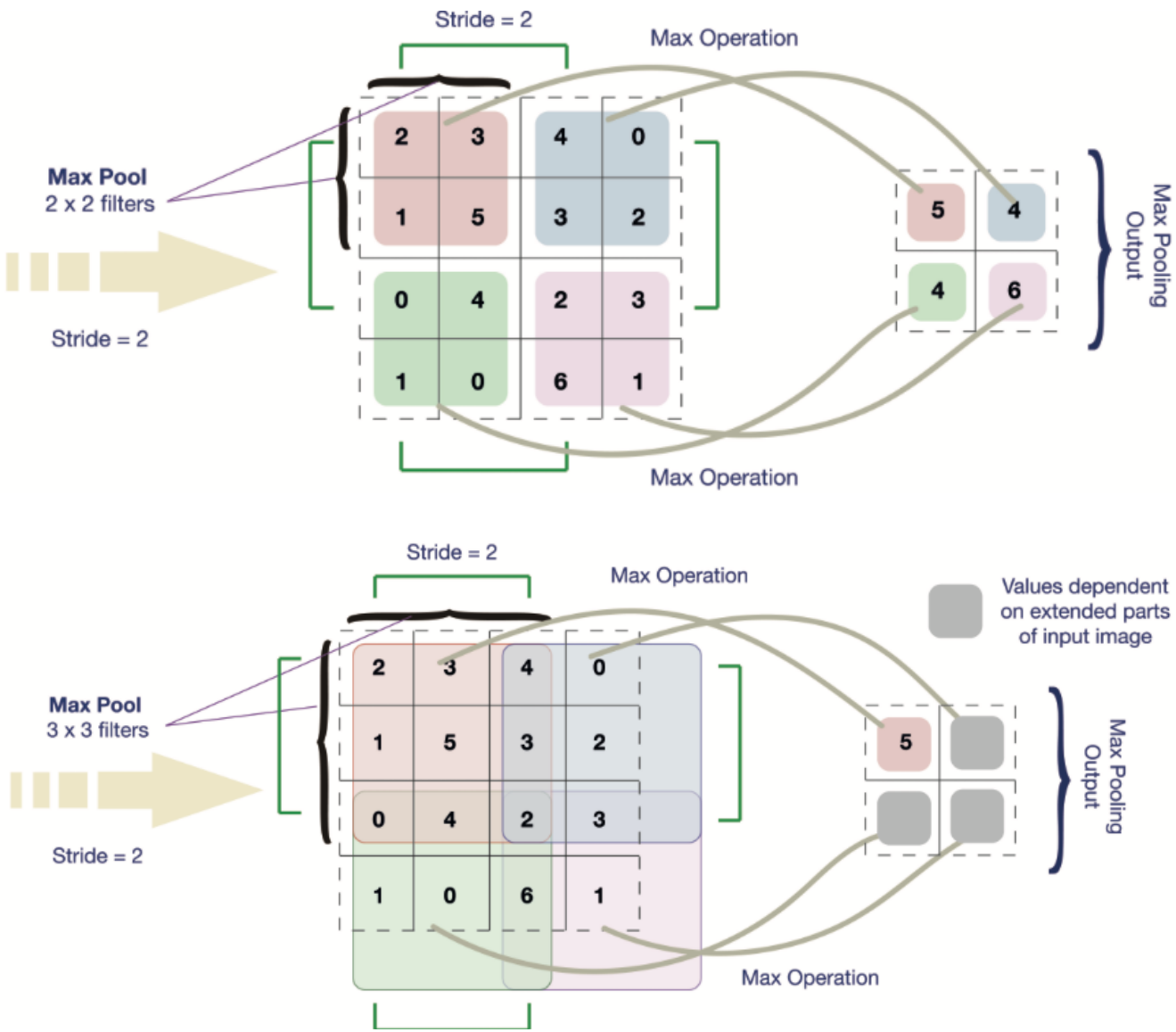
Subsampling or pooling

- After convolution and activation, there is often a pixel-pruning stage
- There are many options for this. The most popular ones are ...
  - Subsampling:
    - keep only the top-left pixel of every $S \times S$ region
    - S is called the "stride"
    - implemented as part of convolution (no wasted computations!)
    - called "downsampling" in signal processing
  - Max pooling:
    - keep only the largest value of each $K \times K$ region
    - shift the region by stride S horizontally & vertically
  - Average pooling:
    - keep only the average value of each $K \times K$ region
    - shift the region by stride S horizontally & vertically
    - called "decimation" in signal processing
- The above is performed independently on every channel and batch item

Illustration of max-pooling

An example Image Portion
for Max Pooling
Numbers represent
the pixel values

2	3	4	0
1	5	3	2
0	4	2	3
1	0	6	1



What dense layers do

- Say that the last convolutional layer produced (after pooling) a tensor of shape $(B, C_{\text{out}}, N_1, N_2)$
- Just before the first dense layer, we "flatten" (i.e., reshape) that tensor into a matrix $\boldsymbol{A}$ of shape (B, N_{in}), where $N_{\text{in}} = C_{\text{out}} N_1 N_2$
- Then using weights $\boldsymbol{W} \in \mathbb{R}^{N_{\text{out}} \times N_{\text{in}}}$ and biases $\boldsymbol{b} \in \mathbb{R}^{N_{\text{out}}}$, the dense layer computes

$$Z[i,m] = \sum_{n=1}^{N_{\text{in}}} W[m,n] A[i,n] + b[m] \quad \text{for } i = 1 \ldots B \text{ and } m = 1 \ldots N_{\text{out}}$$

- Same as the linear stages of the 2-layer neural network from the last unit!
- Also same as a convolutional layer with a 1×1 kernel when $N_1 = 1 = N_2$

Why use convolution in first few layers?

- Convolutional layers have many fewer parameters to learn!
  - Say input has shape $(B, C_{\text{in}}, N_1, N_2)$ & output has shape $(B, C_{\text{out}}, N_1, N_2)$
    - e.g., AlexNet 2nd layer: $(*, 96, 55, 55) \rightarrow (*, 256, 55, 55)$
  - Convolutional layer: $C_{\text{out}} C_{\text{in}} K_1 K_2$ weights and C_{out} biases
    - e.g., AlexNet 2nd layer: $K_1 = 5 = K_2$, thus 6.1×10^5 weights and 256 biases
  - Fully connected layer: $C_{\text{out}} C_{\text{in}} N_1^2 N_2^2$ weights and $C_{\text{out}} N_1 N_2$ biases
    - e.g., AlexNet 2nd layer: 2.2×10^{11} weights and 7.7×10^5 biases!
- Convolutional layers exploit translational invariance
  - small local patterns could be located anywhere

Outline

- Motivation: ImageNet Large-Scale Visual Recognition Challenge
- Deep Networks and Feature Hierarchies
- 2D Convolution Basics
- Convolutional Neural Networks
- Creating and Visualizing Convolutional Layers in PyTorch
- Training CNNs: Backpropagation, Batch-Norm, Dropout, Etc.
- Transfer Learning from Famous Pre-Trained Networks

Example 1: Edge detection with a grayscale image

- Recall: Can perform edge detection by correlating with the Sobel filters

$$\boldsymbol{G}_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \text{ and } \boldsymbol{G}_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

- We will demonstrate this in PyTorch using the "cameraman" image
- First we convert "`im`" from an `np.ndarray` to a `torch.Tensor`:

```
print("Image shape in NumPy is "+str(im.shape))
nrow, ncol = im.shape

# convert to PyTorch batch
nimage = 1          # number of images in batch = 1 since we only have one image
nchan_in = 1        # number of input channels = 1 since it is a greyscale image
batch_shape = (nimage,nchan_in,nrow,ncol)  # batch shape in PyTorch
x = im.reshape(batch_shape)
x_torch = torch.Tensor(x)
print("Batch shape in PyTorch is "+str(x_torch.size()))
```

```
Image shape in NumPy is (512, 512)
Batch shape in PyTorch is torch.Size([1, 1, 512, 512])
```

Creating convolutional layers in PyTorch

- Next we enter the Sobel filters as NumPy arrays:

```
Gx = np.array([[1,0,-1],[2,0,-2],[1,0,-1]])  # Gradient operator in x-direction
Gy = np.array([[1,2,1],[0,0,0],[-1,-2,-1]])  # Gradient operator in y-direction
```

- Then we create the conv-layer `model` in PyTorch as follows:

```
model = nn.Conv2d(in_channels=1, out_channels=2, kernel_size=Gx.shape)
```

- To load the filters into the conv-layer, we first extract the weights and biases using `model.state_dict()`

```
state_dict = model.state_dict()
W = state_dict['weight']
b = state_dict['bias']
```

- Then we set the weights in each channel to `Gx` and `Gy`, and we set the biases to zero

```
# Extend kernels to tensor
W[0,0,:,:] = torch.Tensor(Gx)
W[1,0,:,:] = torch.Tensor(Gy)
b = torch.zeros(2)

# load into PyTorch model
state_dict['bias'] = b
state_dict['weight'] = W
model.load_state_dict(state_dict)
```

- Then we load these weights and biases back into the model using `model.load_state_dict()`

- And finally we process the data

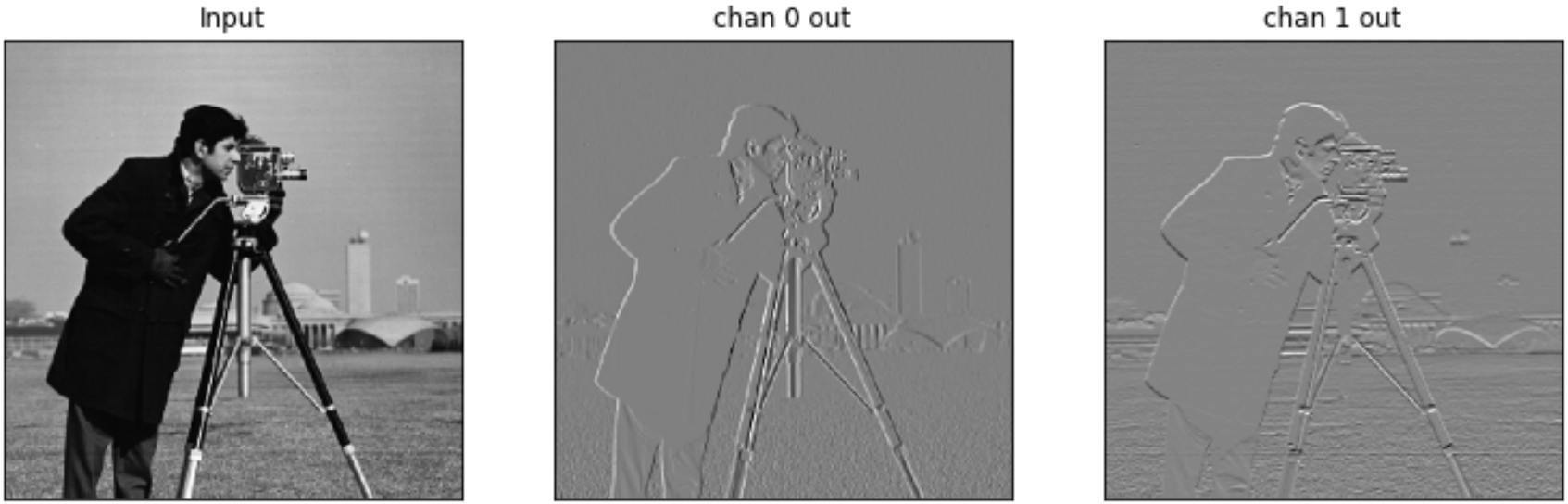
```
# Process data
y_torch = model(x_torch)
y = y_torch.detach().numpy()
```

Edge detection in PyTorch

- The outputs of our CNN can be extracted using

```
chan0=y[0,0,:,:];
chan1=y[0,1,:,:];
```

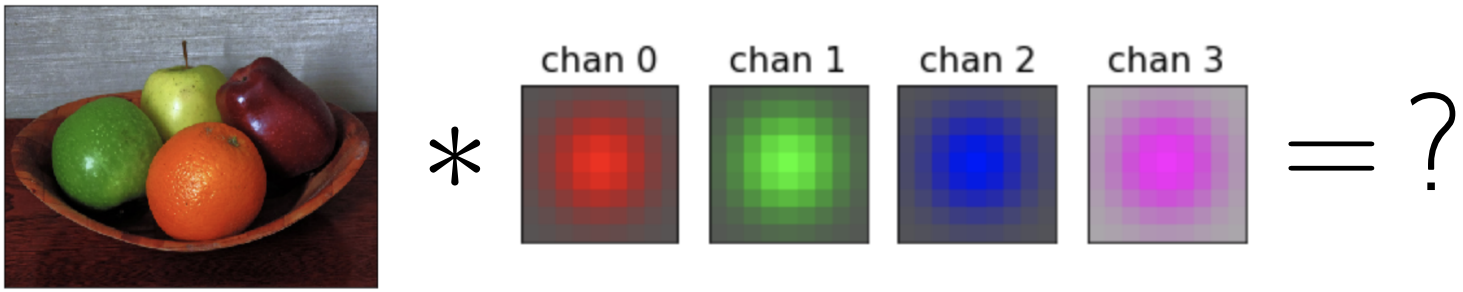
- Plotting these gives the same result that we got before from `scipy.signal.correlate2d`



- As expected, because `torch.nn.Conv2d` performs correlation, not convolution!

Example 2: Color filtering

- Next we correlate a color image with several color filters:



- The color image is a tensor of shape $(368, 487, 3)$ in NumpPy
- We convert that to a tensor $\boldsymbol{X}$ of shape $(1, 3, 368, 487)$ for PyTorch
  - That is, batch size $B = 1$, $C_{\text{in}} = 3$ channels, height $N_1 = 368$, width $N_2 = 487$
- The conv-layer weight matrix $\boldsymbol{W}$ is a tensor of shape $(C_{\text{out}}, C_{\text{in}}, K_1, K_2)$
  - We have $C_{\text{out}} = 4$ filters with $C_{\text{in}} = 3$ colors, height $K_1 = 9$, width $K_2 = 9$
- The conv-layer computes a tensor $\boldsymbol{Z}$ of shape $(B, C_{\text{out}}, N_1, N_2)$ via

$$Z[i, m, j_1, j_2] = \sum_{n=1}^{C_{\text{in}}} \sum_{k_1=1}^{K_1} \sum_{k_2=1}^{K_2} W[m, n, k_1, k_2] X[i, n, j_1+k_1, j_2+k_2] + b[m]$$

Color filtering in PyTorch

- First we convert the NumPy image of shape (368,487,3) into a PyTorch tensor of shape (1,3,368,487)
- Then we create a `model` for the conv-layer using `torch.nn.Conv2d`, just like last time
- Next we extract the weight and bias tensors using `model.state_dict()`

```
nimage = 1
batch_shape = (nimage,nchan_in,nrow,ncol) # batch shap
x = im_color.transpose((2,0,1)).reshape(batch_shape) #
x_torch = torch.Tensor(x)
print("Batch shape in PyTorch is "+str(x_torch.size())
```

```
# Dimensions
nchan_in = 3
nchan_out = 4
kernel_size = (9,9)

# Create network
model = nn.Conv2d(nchan_in,nchan_out,kernel_size)
```

```
state_dict = model.state_dict()
W = state_dict['weight']
b = state_dict['bias']
print(W.shape)
print(b.shape)
```

```
torch.Size([4, 3, 9, 9])
torch.Size([4])
```

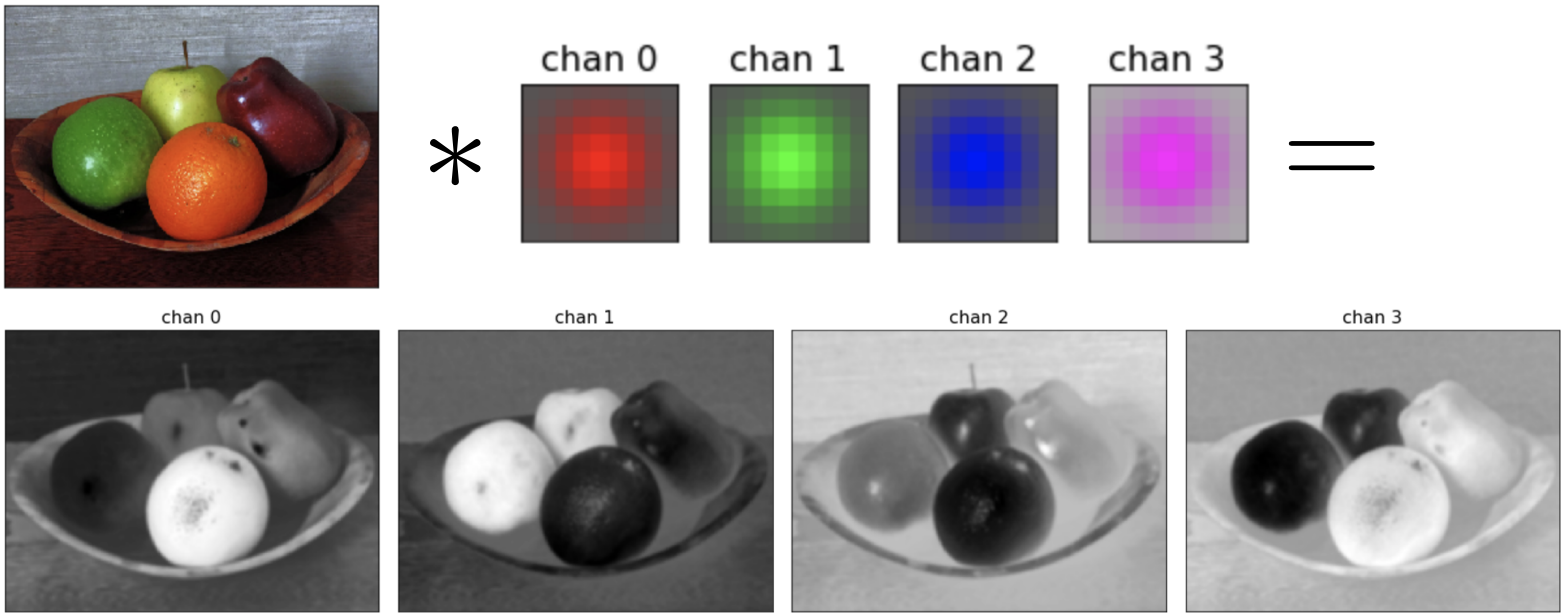
Color filtering in PyTorch

- We make a color weight matrix that maps the C_{in} channels to the C_{out} channels
- Separately, we create a Gaussian convolution kernel
- Then we merge them into a rank-4 tensor `W` and set the bias vector `b` to zero
- The weight & bias are loaded back into the model using `model.load_state_dict()`
- And finally we process the data and convert to NumPy

```
# Color weights
color_wt = np.array([
    [1,    -0.5, -0.5],   # Sensitive to red
    [-0.5,    1, -0.5],   # Sensitive to green
    [-0.5, -0.5,    1],   # Sensitive to blue
    [ 0.5,   -1,  0.5],   # Sensitive to red-blue
])

# Gaussian kernel for convolution over the image
krow, kcol = kernel_size
G = gauss_kernel(krow,kcol,sig=2)

# Extend kernels to tensor using color weights
W = G[None,None,:,:]*color_wt[:,:,None,None]
b = np.zeros(b.shape)

# load into PyTorch model
state_dict['weight'] = torch.Tensor(W)
state_dict['bias'] = torch.Tensor(b)
model.load_state_dict(state_dict)
```

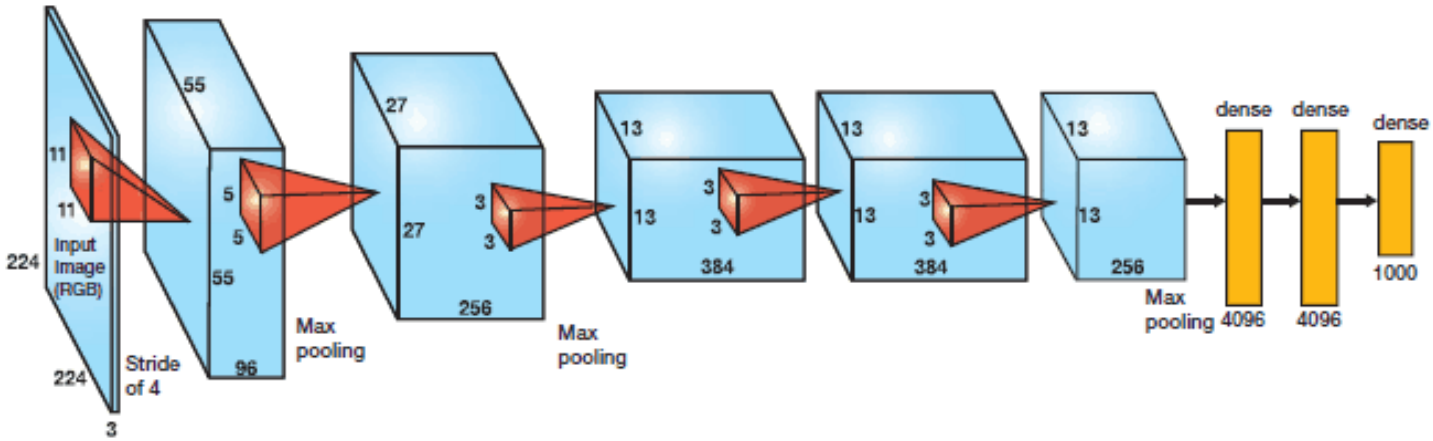
```
y_torch = model(x_torch)
y = y_torch.detach().numpy()
```

Color filtering in PyTorch

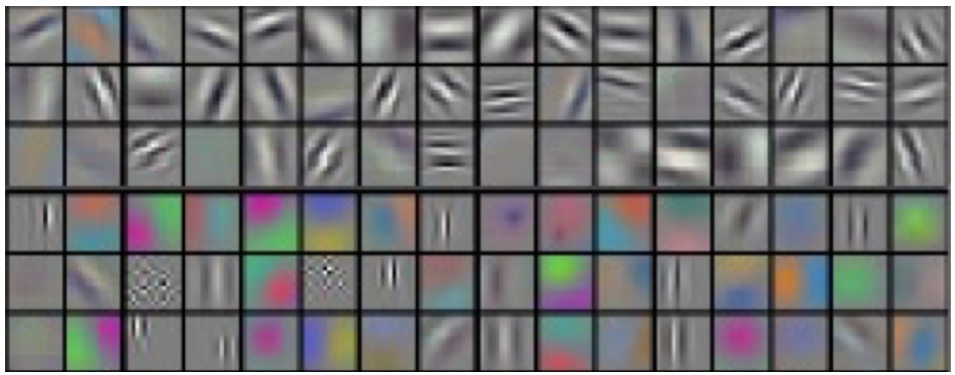


- We now see the effect of the color filtering:
  - The red filter returns a "heatmap" showing where the image has red
  - The green filter returns a "heatmap" showing where the image had green
  - And so on . . .
- These are examples of "feature maps" in machine learning

Learned kernels in AlexNet



- AlexNet's first layer:
  - input: color image, size $224 \times 224 \times 3$ (in NumPy)
  - weights: 96 color filters of size $11 \times 11 \times 3$ (in NumPy), padding='same'

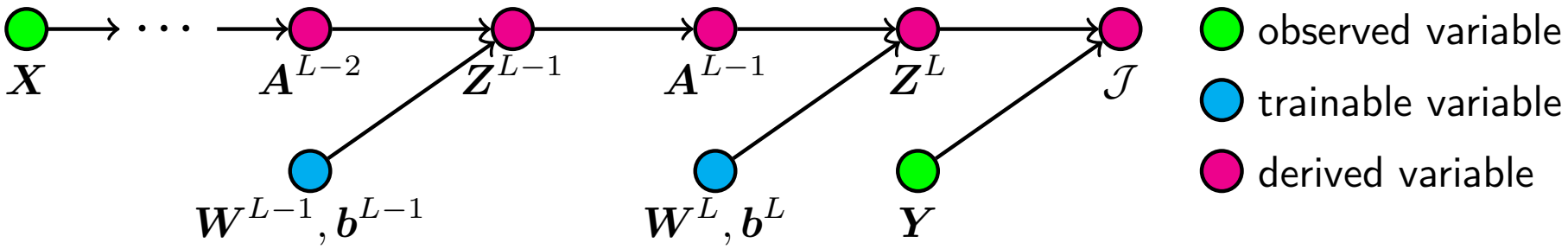


  - sub-sampling with stride 4, yielding output of size $55 \times 55 \times 96$ (in NumPy)

Outline

- Motivation: ImageNet Large-Scale Visual Recognition Challenge
- Deep Networks and Feature Hierarchies
- 2D Convolution Basics
- Convolutional Neural Networks
- Creating and Visualizing Convolutional Layers in PyTorch
- Training CNNs: Backpropagation, Batch-Norm, Dropout, Etc.
- Transfer Learning from Famous Pre-Trained Networks

Backpropagation in CNNs



- We now describe the backpropagation training procedure for an L-layer CNN
- The computation graph is shown above ...
  - starting with the input mini-batch $\boldsymbol{X}$ ...
  - and concluding with the loss $\mathcal{J}$
- Note that the quantities $\boldsymbol{X}, \{\boldsymbol{W}^l, \boldsymbol{b}^l, \boldsymbol{A}^l, \boldsymbol{Z}^l\}_{l=1}^{L}$, and $\boldsymbol{Y}$ are now tensors
  - If there are dense layers, we can treat them as convolutional with kernel size 1×1

The forward pass

- Backprop for deep-CNNs is very similar to that for dense 2-layer networks
- For the forward pass, we first initialize $\boldsymbol{A}^0 = \boldsymbol{X}$.
- Then, for $l = 1, 2, \ldots, L$, we compute

$$Z^l[i,m,j_1,j_2] = \sum_{n=1}^{C_{\text{in}}^l} \sum_{k_1=1}^{K_1^l} \sum_{k_2=1}^{K_2^l} W^l[m,n,k_1,k_2]\, A^{l-1}[i,n,j_1+k_1,j_2+k_2] + b^l[m]$$

$$A^l[i,m,k_1,k_2] = \sum_{j_1=1}^{N_1^l} \sum_{j_2=1}^{N_2^l} P^l[i,m,k_1,k_2,j_1,j_2]\, h^l\big(Z^l[i,m,j_1,j_2]\big)$$

  - $\boldsymbol{P}^l$ represents the subsampling/pooling operation in the lth layer
  - Pooling may be data-dependent, so we index P^l using i and m

- Finally, we compute the loss via

$$\mathcal{J} = \sum_{i=1}^{B} J(\boldsymbol{z}_i^L, \boldsymbol{y}_i) \quad \text{with rank-3 tensor } [\boldsymbol{z}_i^L]_{m,j_1,j_2} = Z^L[i,m,j_1,j_2]$$

The backward pass

- For the backward pass, we initialize by computing

$$\frac{\partial \mathcal{J}}{\partial Z^L[i,m,j_1,j_2]} = \frac{\partial J(\boldsymbol{z}_i^L, \boldsymbol{y}_i)}{\partial Z^L[i,m,j_1,j_2]}$$

  - The details depend on the form of $J(\cdot)$ (e.g., quadratic, cross-entropy, etc.)
  - See examples in the previous lecture packet

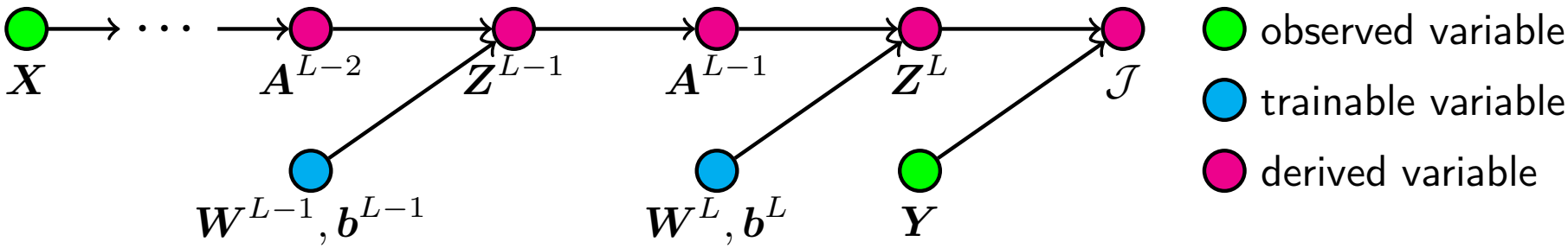
- Then, for $l = L, L-1, \ldots, 1$, we compute

$$\frac{\partial \mathcal{J}}{\partial b^l[m]} = \sum_{i,j_1,j_2} \underbrace{\frac{\partial \mathcal{J}}{\partial Z^l[i,m,j_1,j_2]}}_{\text{from above}} \underbrace{\frac{\partial Z^l[i,m,j_1,j_2]}{\partial b^l[m]}}_{=1}$$

$$\frac{\partial \mathcal{J}}{\partial W^l[m,n,k_1,k_2]} = \sum_{i,j_1,j_2} \underbrace{\frac{\partial \mathcal{J}}{\partial Z^l[i,m,j_1,j_2]}}_{\text{from above}} \underbrace{\frac{\partial Z^l[i,m,j_1,j_2]}{\partial W^l[m,n,k_1,k_2]}}_{=A^{l-1}[i,n,j_1+k_1,j_2+k_2]}$$

  - This is a correlation, just like in a conv-layer!

The backward pass (continued)



- Continuing, we compute

$$\frac{\partial \mathcal{J}}{\partial A^{l-1}[i,n,k_1,k_2]} = \sum_{m,j_1,j_2} \underbrace{\frac{\partial \mathcal{J}}{\partial Z^{l}[i,m,j_1,j_2]}}_{\text{from above}} \underbrace{\frac{\partial Z^{l}[i,m,j_1,j_2]}{\partial A^{l-1}[i,n,k_1,k_2]}}_{=W^{l}[m,n,k_1-j_1,k_2-j_2]}$$

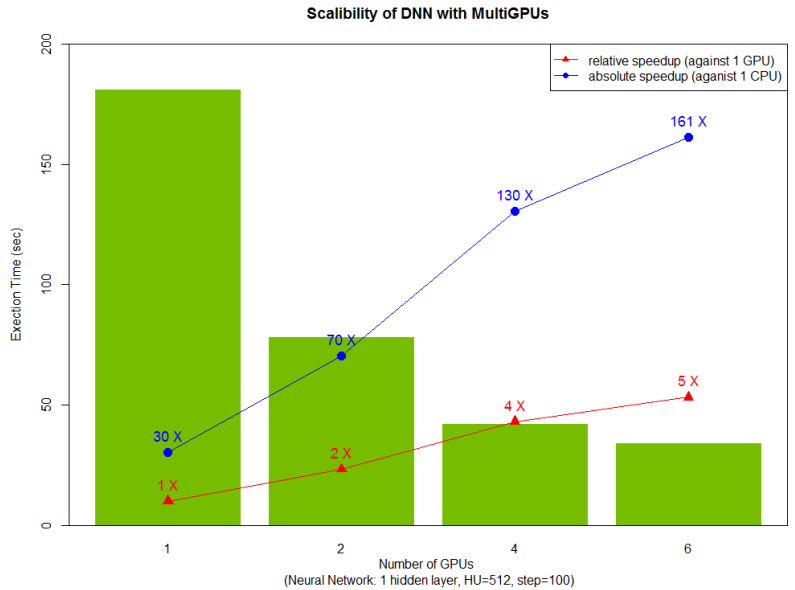
  - Note convolution!

$$\frac{\partial \mathcal{J}}{\partial Z^{l-1}[i,m,j_1,j_2]} = \sum_{k_1,k_2} \underbrace{\frac{\partial \mathcal{J}}{\partial A^{l-1}[i,m,k_1,k_2]}}_{\text{from above}} \underbrace{\frac{\partial A^{l-1}[i,m,k_1,k_2]}{\partial Z^{l-1}[i,m,j_1,j_2]}}_{= P^{l-1}[i,m,k_1,k_2,j_1,j_2] \times h^{(l-1)'}(Z^{l-1}[i,m,j_1,j_2])}$$

- and then repeat with $l \leftarrow l-1$

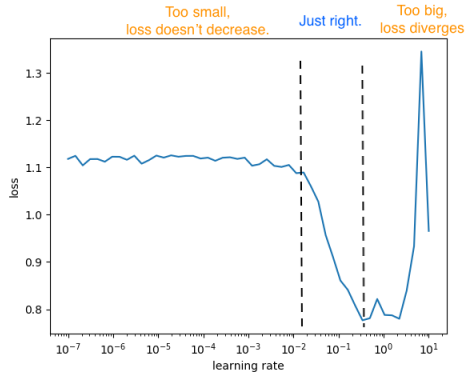
GPUs

- State-of-the-art CNNs involve millions of parameters
- and require huge datasets for training!
- Conventional processors cannot train in reasonable time
- So people often use Graphics Processing Units (GPUs)
  - originally designed for video graphics
  - now essential to deep learning
- Not all GPUs work well for deep learning!
  - can buy a GPU workstation for ∼$3k
  - can buy a many-GPU cluster for ∼$30k
  - can rent GPUs in the cloud for ∼$1/hr (e.g., Google CoLab and many others)

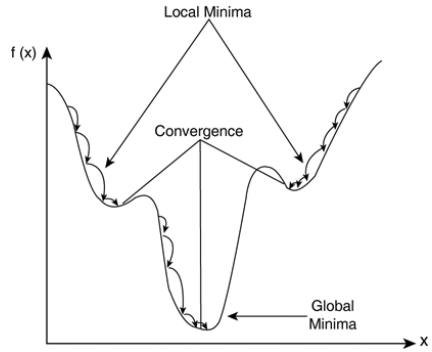


Learning-rate scheduling

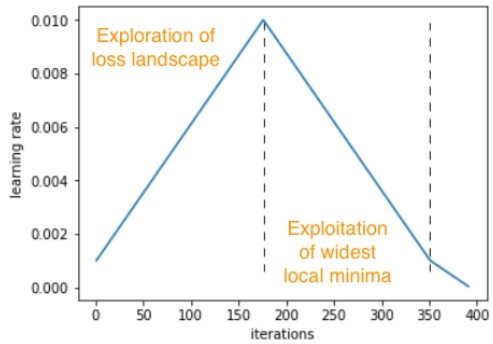
- Fixed learning-rate:
  - With convex optimization, small learning-rates hurt in speed but not accuracy
  - With CNNs, small learning-rates cause convergence to bad local minima!



- Scheduled learning-rate:
  - First use large learning-rate to escape bad local minima
  - Then small learning-rate to converge to good local minimum
- Some algs also use momentum to escape bad local minima



- Learning-rate schedules:
  - Many implemented in PyTorch: multiplicative, exponential, stepwise, cosine, cyclic, one-cycle, etc.
  - Excellent results have been attained using a one-cycle schedule configured by a learning-rate range test



Batch normalization

- Another way to speed up training is to use batch normalization
  - see general discussion here, and CNN-specific discussion here
- Why? Recall that it is good practice to standardize the raw features $\boldsymbol{X}$
  - that is, for each feature, make mean = 0 and variance = 1 over each minibatch
- With batch norm, we first standardize the linear outputs $\boldsymbol{Z}$ *in each channel and layer*, and then learn an optimal mean & variance for them:

$$\overline{Z}^l[m] = \frac{1}{BN_1N_2} \sum_{i=1}^{B} \sum_{j_1=1}^{N_1} \sum_{j_2=1}^{N_2} Z^l[i,m,j_1,j_2] \qquad \text{(sample mean)}$$

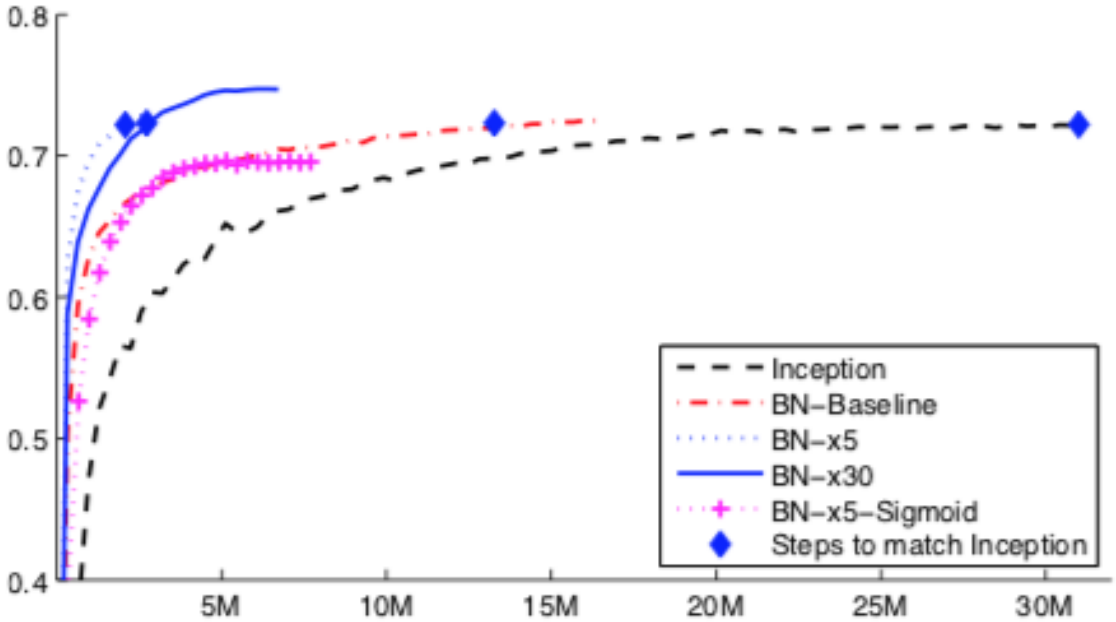
$$s_Z^l[m] = \sqrt{\epsilon + \frac{1}{BN_1N_2} \sum_{i=1}^{B} \sum_{j_1=1}^{N_1} \sum_{j_2=1}^{N_2} \left(Z^l[i,m,j_1,j_2] - \overline{Z}^l[m]\right)^2} \qquad \text{(sample stdv)}$$

$$\widehat{Z}^l[i,m,j_1,j_2] = \gamma^l[m] \frac{Z^l[i,m,j_1,j_2] - \overline{Z}^l[m]}{s_Z^l[m]} + \beta^l[m]$$

where $\gamma^l[m]$ and $\beta^l[m]$ are the learned parameters and $\epsilon > 0$ is small

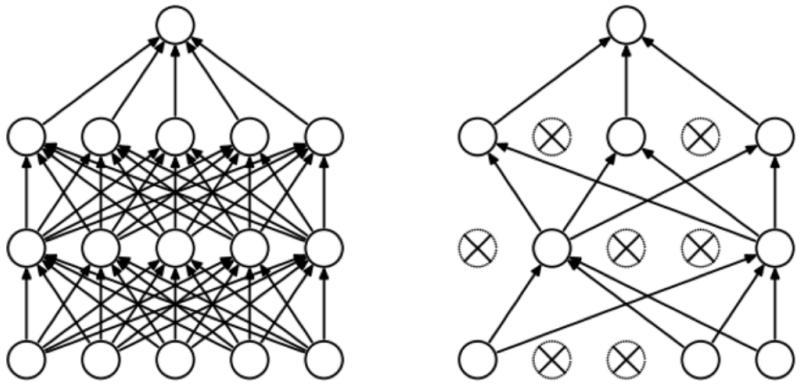
Batch normalization

- Batch norm allows higher learning rates, thus reduced training times, because
  - it reduces variation across channels and minibatches
  - it helps to decouple layers
  - it makes the cost function smoother
- Below is an example from the original paper:
  - the "$\times 5$" and "$\times 30$" refer to learning-rate increases over the baseline



Dropout

- Dropout is a well-known technique to reduce overfitting during training
- Idea: temporarily remove a random subset of neurons in each mini-batch
  - Trains an *ensemble* of simpler networks! (More about ensembles in next unit...)



(a) Standard Neural Net (b) After applying dropout.

- Basic implementation (used before dense layers):
  - Multiply each $A^l[i,m]$ by a Bernoulli random variable $D^l[i,m] \in \{0, 1/p\}$
    - The "on" probability $p \triangleq \Pr\{D^l[i,m] \neq 0\}$ is a tuning parameter (usually $p = 0.5$)
    - So, we either zero the activation or scale it by $1/p$ to preserve the average value
  - This is only done during training; full network is used during testing phase
- Comments:
  - Dropout tends to increase # of training epochs
  - Dropout reduces the capacity of the network (so may need to increase # chans)
  - For convolutional layers, other versions of dropout are more appropriate

Data augmentation

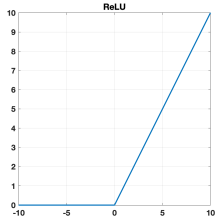
- Often our training dataset is smaller than we'd like
- Idea: Augment the dataset
  - Ex: generate additional images by flipping, shifting, scaling, rotating, and cropping the original images

  - Other approaches and applications of augmentation are discussed here
- In PyTorch, data augmentation is performed by the `DataLoader`
  - See the tutorial here

Improvements on ReLU activation

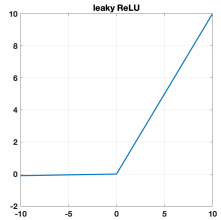
- Deep networks use ReLU hidden activations to prevent vanishing gradients
  - Problem: a subset of nodes can get stuck outputting zero ("dead ReLU")

$$h_{\textbf{ReLU}}(z) = \begin{cases} z & z \geq 0 \\ 0 & z < 0 \end{cases},$$



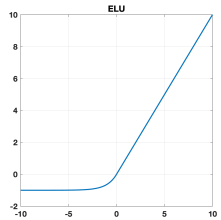
- To avoid dead nodes, Leaky ReLU uses slope $a = 0.01$ with negative inputs
  - Parametric ReLU (PReLU) goes further, allowing the left slope a to be *learned*

$$h_{\textbf{PReLU}}(z) = \begin{cases} z & z \geq 0 \\ az & z < 0 \end{cases}, \quad a > 0$$



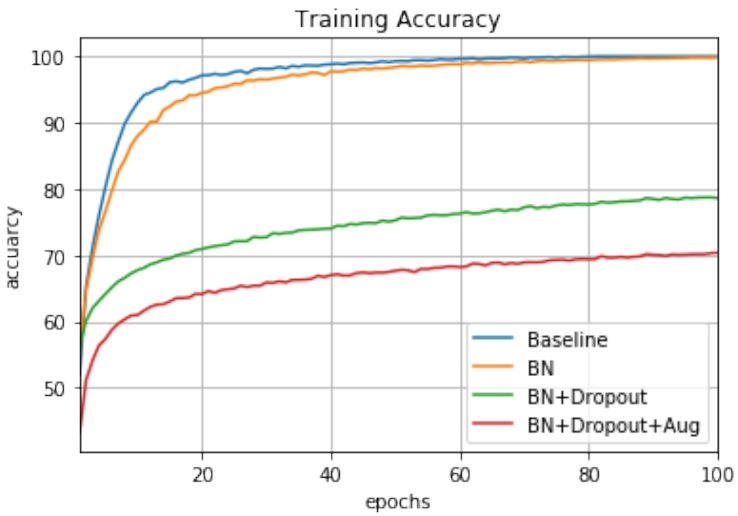
- Another way to avoid dead nodes is to use the exponential linear unit (ELU)
  - Reduces "bias shift," which speeds up learning

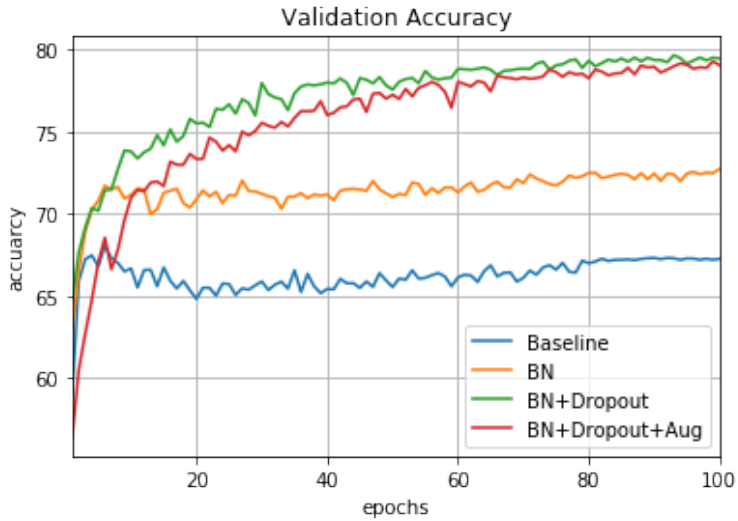
$$h_{\textbf{ELU}}(z) = \begin{cases} z & z \geq 0 \\ \alpha(e^z - 1) & z < 0 \end{cases}, \quad \text{with } \alpha > 0 \text{ (usually } \alpha = 1)$$



PyTorch CNN demo

- We explore these training mods in `demo09b_cnn_classifier.ipynb`:
  - Goal: classify CIFAR-10 (i.e., 32×32 color images, 10 classes)
  - a simple CNN: 2 conv layers with ReLU & max pooling, followed by 2 dense layers
- The above CNN achieves 67% test accuracy (@ 100 epochs)
  - adding batch norm: 72.5% test accuracy
  - adding dropout: 79.5% test accuracy
  - adding data augmentation: 79% (doesn't help; we have enough data)





- Another good PyTorch CNN tutorial/demo can be found here

Outline

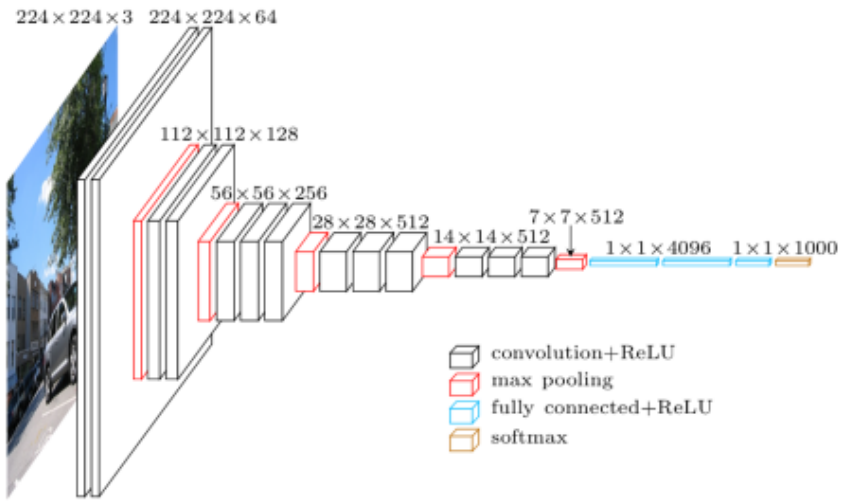
- Motivation: ImageNet Large-Scale Visual Recognition Challenge
- Deep Networks and Feature Hierarchies
- 2D Convolution Basics
- Convolutional Neural Networks
- Creating and Visualizing Convolutional Layers in PyTorch
- Training CNNs: Backpropagation, Batch-Norm, Dropout, Etc.
- Transfer Learning from Famous Pre-Trained Networks

Transfer learning from a pretrained network

- State-of-the-art networks take enormous resources to train
  - millions of parameters $\Rightarrow$ weeks of training on GPU clusters
- Idea: "transfer learning" = adapt a pre-trained network to a new task
  - early (convolutional) layers are universal: keep them fixed!
  - retrain the later (dense) layers to perform well on new data or a new task
- Example applications:
  - You have only enough data to train a few layers (not whole network)
  - You use the early layers to generate features for a different learning task
  - Other applications are discussed here
- PyTorch's `torchvision.models` package has many pretrained nets!
  - e.g., AlexNet, VGG, GoogLeNet, ResNet, Inception-v3, DenseNet, SqueezeNet
  - We will go through `demo09d_cnn_vgg16` later, to show how this is done

VGG

- VGG
  - Idea: Use very small kernels (e.g., 3×3) but more layers (e.g., 16)
    - Why? Multiple small convolutions have the "receptive field" of one big convolution
    - Ex: 5 conv layers with 3×3 kernels $\Leftrightarrow$ 1 conv layer with an 11×11 kernel
    - But five (nonlinear) layers are more powerful than one
  - Several depths: VGG11, VGG13, VGG16, VGG19
  - Won some contests in ImageNet ILSVRC-2014
    - VGG16 achieved top-5 error of 7.5% (AlexNet got 15.3%)
- VGG16 remains a respected network
  - Lower layers are often used as feature extractors for other vision tasks
  - Thus good for transfer learning
  - $3\times$ number of parameters in AlexNet

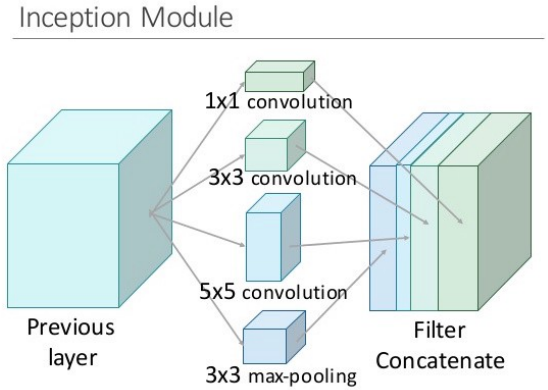


GoogleLeNet / Inception-v1

- From 2012 (AlexNet) to 2014 (VGG16), performance got better as networks got deeper
- But, as networks got deeper, the number of parameters was getting too large (e.g., 60 $\rightarrow$ 180 million!)
- So, Google researchers worked to design a network that was deep but much more efficient

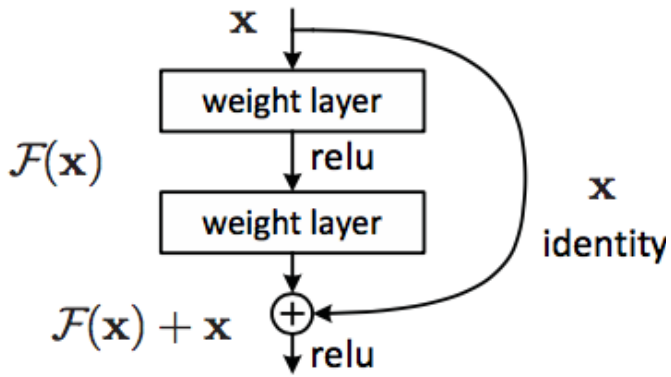
From the film *Inception* (2010)

- Their GoogLeNet (also called Inception-v1) used only 5 million parameters!
  - Won some contests in ImageNet ILSVRC-2014
  - Achieved top-5 error of 6.7% (VGG16 got 7.5%)
  - Used 22 layers with "inception modules"
    - Several conv layers in parallel, with different kernel sizes
    - Key idea: for bigger kernels, use fewer channels
    - Multi-size kernels also help make scale-invariant
    - Good overview here



Residual networks

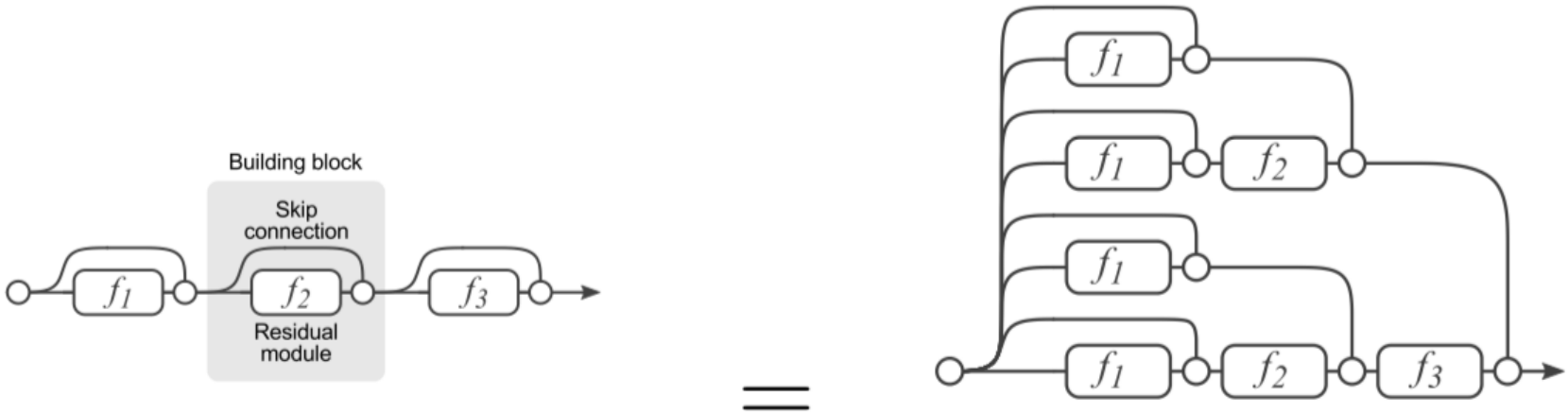
- Through 2015, the best networks got deeper. But deep nets are hard to train:
  - vanishing gradients (largely solved by ReLU & batchnorm)
  - degradation problem: as networks get very deep, even *training* error increases!
- Idea: If network layers can act like identity functions, then deeper networks *should* be able to perform at least as well as shallower networks
  - But difficult to make layers act like identity!
  - Idea: Facilitate identity via residual blocks:
  - Handle dimension changes w/ learned dense layer



- ResNet, introduced by Microsoft Research in Dec 2015
  - Solves degradation problem; allows super deep networks! (100s of layers)
  - Won ImageNet ILSVRC-2015 with 3.6% error rate! (GoogLeNet got 6.7%)
  - Slightly lower complexity than VGG16

Alternative view of residual networks

- An L-layer ResNet can be unraveled into an ensemble of 2^L parallel networks:



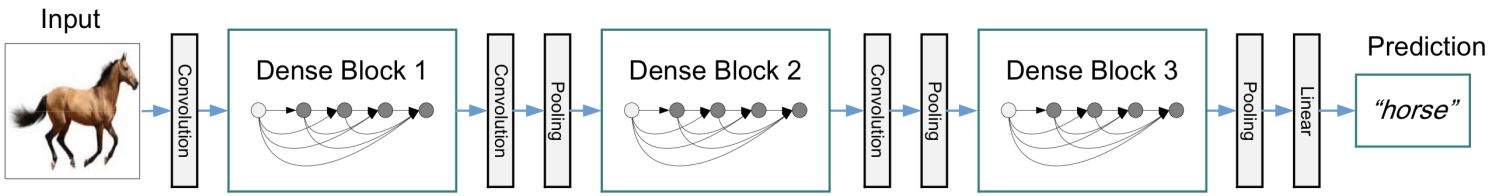
(a) Conventional 3-block residual network

(b) Unraveled view of (a)

- Key idea: Most paths $\approx L/2$ length, and longer paths contribute little gradient
  - Ex: In a 110-layer network, only paths with length 10-34 are meaningful
  - So, ResNets are really ensembles of shallow networks!
- Can remove ResNet layers without affecting the overall performance much!
  - Idea: Apply dropout to layers: "Deep network with stochastic depth"
  - Speeds up ResNet training and reduces overfitting

Other deep convolutional networks

- Inception-v2, Inception-v3 (2015), Inception-v4, Inception-ResNet (2016):
  - Evolutions with more efficient inception modules, auxiliary outputs, etc. (see here)
  - Inception-v3 took 2nd place in ILSVRC-2015
- DenseNet (2016):
  - Each layer connected to all previous layers. Outperforms ResNet (see overview)



- SqueezeNet (2016):
  - Motivated by limited-memory (e.g., mobile) implementations
  - Achieved AlexNet-like performance with 50x fewer parameters
- MobileNet (2017):
  - Very efficient in memory & computation, yet high accuracy (see overview)

Demonstration of pre-trained networks in PyTorch

- VGG16 is one of many pre-trained models available in TorchVision
- After loading, examine its structure:
  1. features section:
     - 13 conv layers with ReLU & maxpool
  2. avgpool section:
     - ensures 7x7 passed to classifer; handles different image sizes
  3. classifier section:
     - 3 dense layers with ReLU & dropout

```
# Load appropriate packages
import torchvision.models as models
```

```
model = models.vgg16(pretrained=True)
```

```
print(str(model))
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=F
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    .
    .
    .
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

Getting images

- We need some test images to put through the network
- A great source is Flickr
  - First, need to get some API keys
  - Then can download images by keyword (e.g., "elephants")
  - Details in `demo09c_cnn_flickr`
- Storing the images:
  - Use separate `test/` and `train/` directories
  - Use separate class subdirectories (e.g., `test/elephants/`)

Loading images

- Use PyTorch's ImageFolder function to create a `DataSet` object
  - Supports normalization/resizing/augmentation transforms
  - Note: We don't need the images to be 224 x 224 (due to AdaptiveAvgPool)
- Use PyTorch's DataLoader to draw minibatches

```
from torchvision.datasets import ImageFolder
import torchvision.transforms as transforms

# We'll first create a transformation that
# 1) crops to a desired size (we'll do 224x224)
# 2) converts to a Torch.Tensor (do this before normalization!)
# 3) normalizes using the standard ImageNet scalings
crop = transforms.CenterCrop(224)
normalize = transforms.Normalize(
   mean=[0.485, 0.456, 0.406],
   std=[0.229, 0.224, 0.225]
)
transform = transforms.Compose([crop, transforms.ToTensor(), normalize])

# Create a transformed DataSet in the `demo` folder
dataset = ImageFolder(root='demo', transform=transform)
# Create a DataLoader with a given batch size
batch_size = 10
loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size)
```

```
x, y = next(iter(loader))
```

Classifying the test images

- We first put the input `x` through the `model` to get the linear scores
- Then we put the scores through a softmax layer
- Then we extract the top 3 classes
- Then we print the results as a pandas dataframe
- The network incorrectly classified image 7 (but a human might have too!)

```
import torch.nn as nn

# compute model predictions for images in batch x
model.eval()
with torch.no_grad():
    output = model(x)
    output = nn.functional.softmax(output, dim=1)
```

```
# Compute top 3 categories
ntop = 3
top3_pct, top3_idx = output.topk(ntop)
top3_pct = top3_pct.numpy()
top3_idx = top3_idx.numpy()
```

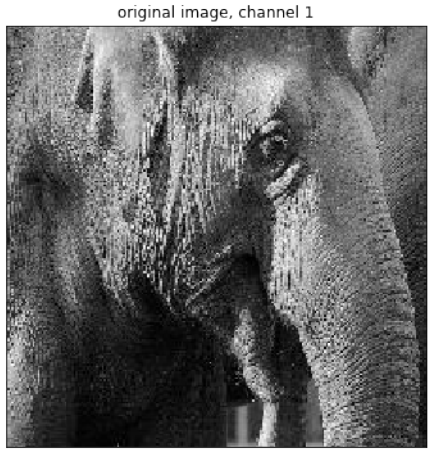
	class 0	prob 0	class 1	prob 1	class 2	prob 2
0	African elephan	0.699629	tusker	0.281839	Indian elephant	0.018526
1	African elephan	0.590090	tusker	0.349729	Indian elephant	0.060181
2	tusker	0.544234	African elephan	0.321651	Indian elephant	0.134114
3	Indian elephant	0.790688	African elephan	0.141189	tusker	0.067665
4	African elephan	0.752693	tusker	0.194113	Indian elephant	0.041240
5	African elephan	0.775054	tusker	0.172309	Indian elephant	0.050715
6	tusker	0.454303	Indian elephant	0.368875	African elephan	0.176821
7	Indian elephant	0.726010	African elephan	0.218296	tusker	0.055628
8	African elephan	0.944679	tusker	0.038309	Indian elephant	0.016993
9	African elephan	0.866949	tusker	0.117329	Indian elephant	0.015721

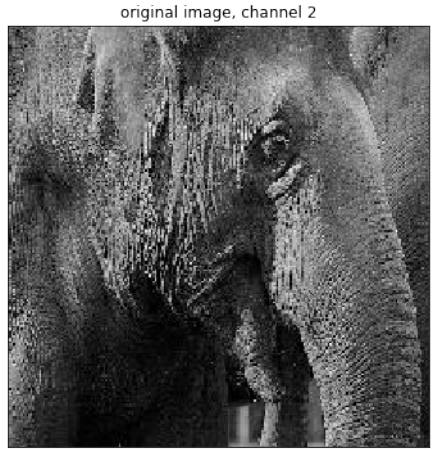
Extracting internal variables

```
class FeatureExtractor(nn.Module):
    def __init__(self, model, feature_layers, classifier_layers):
        super(FeatureExtractor,self).__init__()
        self.model = model
        self.feature_layers = feature_layers
        self.classifier_layers = classifier_layers

    def forward(self, x):
        feature_outputs = []
        classifier_outputs = []
        # feature modules
        for name, module in self.model.features._modules.items():
            x = module(x)
            if name in self.feature_layers:
                feature_outputs += [x]

        # avgpool module (used to deal with image size different from 224x224 pixels)
        x = self.model.avgpool(x)
        x = x.view(x.size(0),-1) # flatten

        # classifier modules
        for name, module in self.model.classifier._modules.items():
            x = module(x)
            if name in self.classifier_layers:
                classifier_outputs += [x]

        return x, feature_outputs, classifier_outputs
```

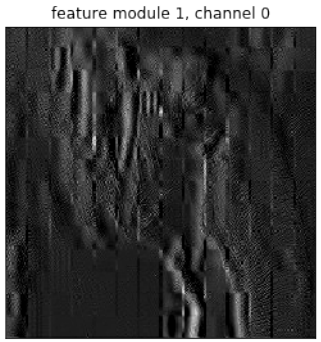
Plotting internal variables

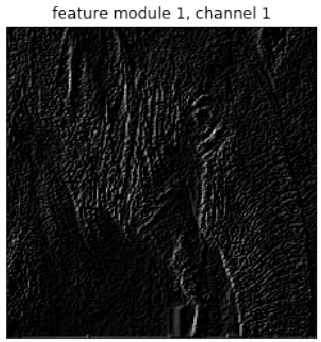
Original image channels:

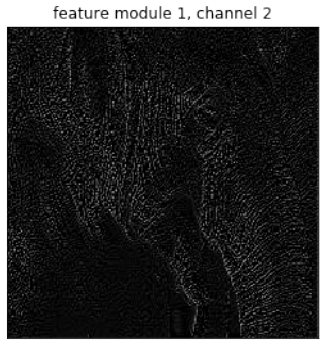


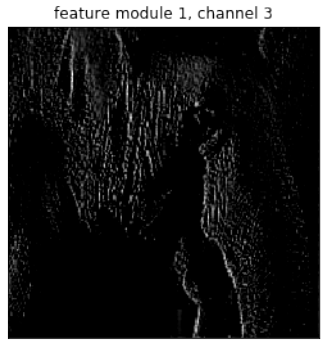


Layer-1 activations:

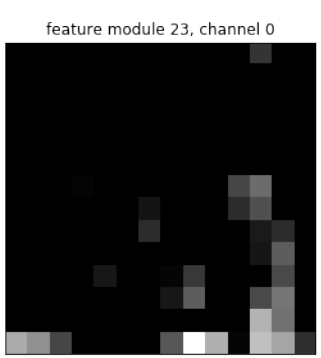


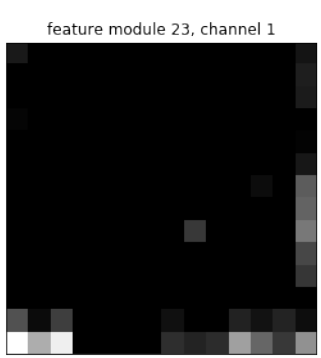


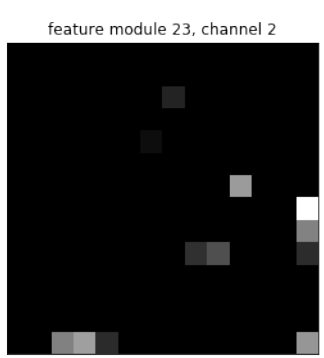


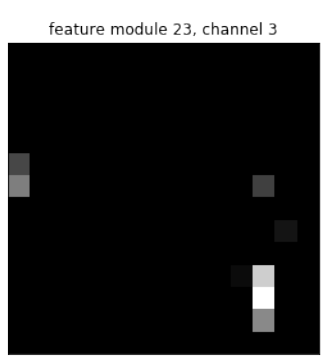


Layer-10 activations:



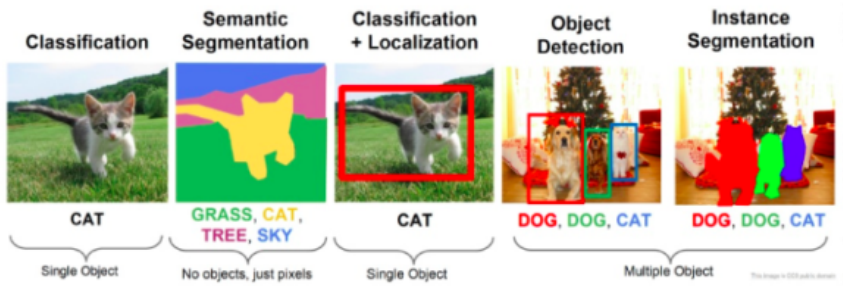






Other applications of deep learning

- Keep in mind that the previous networks all target image classification
- Deep nets have been applied to many other imaging problems, e.g.,
  - object detection
  - image segmentation
  - image compression
  - image denoising, deblocking, deblurring
  - image recovery: super-resolution, phase retrieval, MR, CT, inpainting/completion
  - image generation

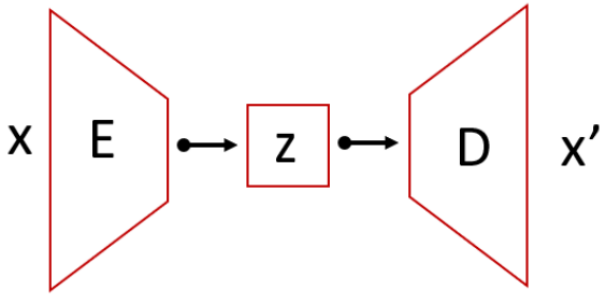


- And deep nets have applied to *many many* applications outside imaging!
- Sometimes the networks look similar to what we have seen
  - But sometimes they look very different!
  - Examples: U-Net, recurrent neural nets (RNNs), LSTMs, transformers, etc.

Autoencoders & VAEs

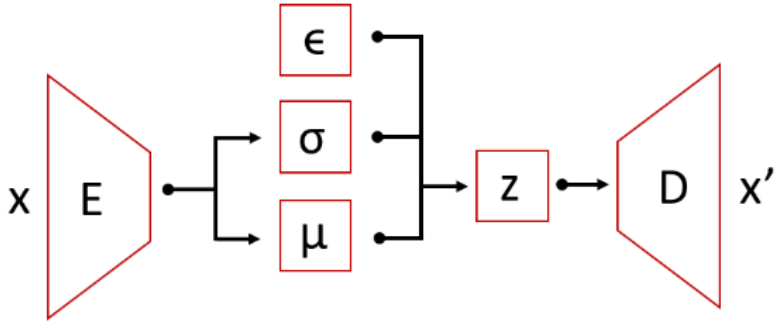
Autoencoders:

- Design an encoder/decoder network that compresses $\{\boldsymbol{x}_i\}_{i=1}^n$ to latent vectors $\{\boldsymbol{z}_i\}_{i=1}^n$ from which one can approximately recover $\{\boldsymbol{x}_i\}_{i=1}^n$



Variational autoencoders (VAEs):

- The encoder learns the parameters of a distribution (e.g., mean and variance of a Gaussian) and the decoder takes in samples $\{\boldsymbol{z}_i\}$ from that distribution



- Training cost:

$$J(\boldsymbol{\theta},\boldsymbol{\phi}) = \sum_{i=1}^{n} \mathbb{E}\left\{\|\boldsymbol{x}_i - D_{\boldsymbol{\phi}}(\overbrace{\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}_i) + \boldsymbol{\sigma}_{\boldsymbol{\theta}}(\boldsymbol{x}_i) \odot \boldsymbol{\epsilon}}^{\boldsymbol{z}_i})\|_2^2\right\} + \lambda \mathsf{KL}\big(\mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{\sigma}_{\boldsymbol{\theta}}^2(\boldsymbol{x}_i)), \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})\big)$$

- Applications:
  - dimensionality reduction, denoising, anomaly detection, generative modeling

Generative Adversarial Networks (GANs)

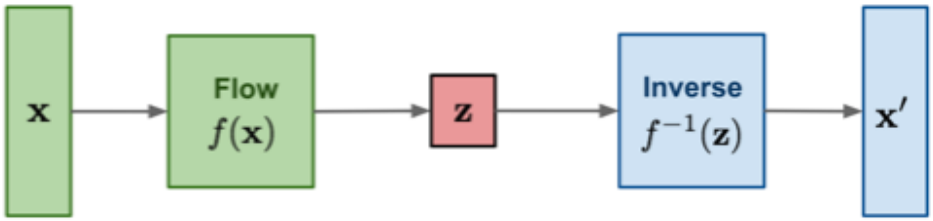
- Goal: Given a training set $\{\boldsymbol{x}_i\}_{i=1}^n$, generate new $\{\boldsymbol{x}\}$ similar to training data
  - This is an example of "unsupervised learning": there are no targets y_i

- Approach:
  1. Design a generator $G(\boldsymbol{z})$ to map a latent vector $\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$ to $\boldsymbol{x}$ domain
  2. Design a discriminator $D(\boldsymbol{x})$ to classify whether an $\boldsymbol{x}$ is real $\uparrow$ or fake $\downarrow$
  - Jointly train G and D: the G tries to fool D, while the D tries to distinguish G's samples from the training samples

$$J_G(\boldsymbol{\phi}) = -\mathbb{E}\left\{\ln D(G_{\boldsymbol{\phi}}(\boldsymbol{z})\right\}, \quad J_D(\boldsymbol{\theta}) = -\tfrac{1}{n}\textstyle\sum_{i=1}^n \ln D_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - \mathbb{E}\left\{\ln(1 - D_{\boldsymbol{\theta}}(G(\boldsymbol{z})))\right\}$$

- Versions:
  - Original, DCGAN, ProgressiveGAN, WassersteinGAN, StyleGAN(v2)(v3)

Normalizing Flows



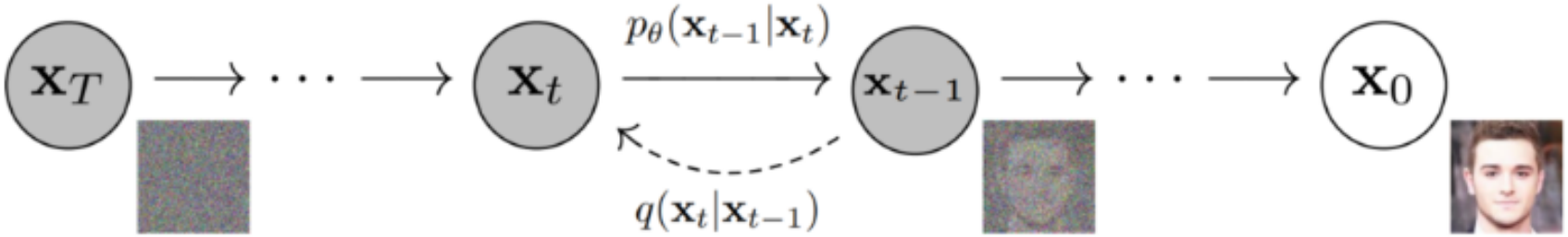
- A normalizing flow is an invertible neural network $f_{\boldsymbol{\theta}}(\cdot)$ that maps a training set $\{\boldsymbol{x}_i\}_{i=1}^n$ to $\{\boldsymbol{z}_i\}_{i=1}^n$ that resemble Gaussian samples
- Once trained, the inverse $f_{\boldsymbol{\theta}}^{-1}(\cdot)$ can be used to map Gaussian $\{\boldsymbol{z}\}$ to samples $\{\boldsymbol{x}\}$ resembling the training set
- The flow can be trained using maximum likelihood (much simpler than GAN!):

$$J(\boldsymbol{\theta}) = \ln \prod_{i=1}^{n} \mathcal{N}(f_{\boldsymbol{\theta}}^{-1}(\boldsymbol{x}_i); \mathbf{0}, \boldsymbol{I}) \cdot \left| \det \left(\frac{\partial}{\partial \boldsymbol{x}} f_{\boldsymbol{\theta}}^{-1}(\boldsymbol{x}_i) \right) \right|$$

  but, for efficient training, $\det \left(\frac{\partial}{\partial \boldsymbol{x}} f^{-1} \right)$ must be easy to compute

- Well-known approaches: NICE, RealNVP, Glow

Diffusion Models



- Main ideas:
  - Forward diffusion: start with original image $\boldsymbol{x}_0$ and gradually add noise over T steps: $q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) = \mathcal{N}(\boldsymbol{x}_t; \sqrt{1-\beta_t}\boldsymbol{x}_{t-1}, \beta_t \boldsymbol{I})$
  - Backward diffusion: start with random $\boldsymbol{x}_T \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$ and gradually remove noise via learned $p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) \approx \mathcal{N}(\boldsymbol{x}_{t-1}; \boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t), \boldsymbol{\sigma}^2_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t))$
  - Landmark papers: DPMs, NCSN, DDPM
- Can generalize above method to condition on a provided piece of information, and can work jointly with images and text: Dall-E, Imagen, Stable Diffusion
- Images generated by Imagen, taken from this overview on diffusion models:

"A photo of a Shiba Inu dog with a backpack riding a bike. It is wearing sunglasses and a beach hat."

"A blue jay standing on a large basket of rainbow macarons"

"A brain riding a rocketship heading towards the moon"

Learning objectives

- Recognize CNNs as learning and exploiting hierarchical patterns
- 2D convolution:
  - Understand as local pattern matching
  - Understand boundary conditions and relation between convolution and correlation
  - Know how to implement convolution with scipy.signal and PyTorch
- Convolutional neural networks:
  - Understand convolutional layers, dense layers, subsampling, pooling
  - Understand backpropagation training
  - Recognize training tricks like batch-norm, dropout, data augmentation
- Transfer learning and pre-trained networks:
  - Be familiar with AlexNet, VGG, Inception, ResNet, and other famous networks
  - Know how to work with pre-trained networks in PyTorch