Unit 10. Convolutional and Deep Neural Networks

EL-GY 6143: INTRODUCTION TO MACHINE LEARNING

PROF. PEI LIU

Learning Objectives

- ❑ Explain the motivation for deep networks
  - Qualitatively describe feature hierarchies
- ❑ Mathematically describe convolutional layer
- ❑ Visualize outputs of convolutional filters on images
- ❑ Implement convolutional layers in Tensorflow
- ❑ Describe and implement algorithmic enhancements for deep networks
  - Dropout, batch normalization and data augmentation
- ❑ Determine when a GPU is needed
- ❑ Describe and implement transfer learning

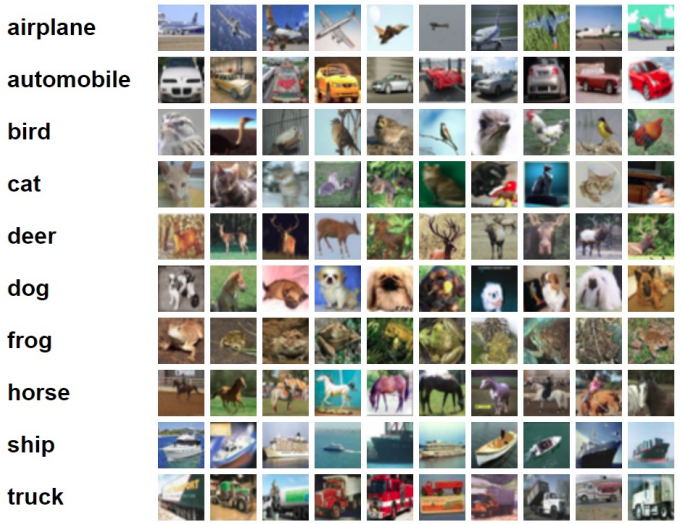
Outline

- ❑ Motivation: ImageNet Large-Scale Visual Recognition Challenge (ILSVR)
- ❑ Deep Networks and Feature Hierarchies
- ❑ 2D convolutions
- ❑ Convolutional neural networks
- ❑ Creating and visualizing convolutional layers in Keras
- ❑ Training CNNs: Backpropagation, Batch-norm, Dropout, etc.
- ❑ Transfer Learning from Famous Pre-trained Networks

Dataset Pre-2009

- Pre-2009, many image recognition systems worked on relatively small datasets
- MNIST:
  - 10 classes, 70,000 examples, 28 x 28 images
- CIFAR 10 (right)
  - 10 classes, 60000 examples, 32x32 color
- CIFAR 100:
  - 100 classes, 600000 examples, 32x32 color
- PASCAL VOC:
  - 20 classes, 11530 examples, variable size images
- Performance saturated
  - Difficult to make significant advancements

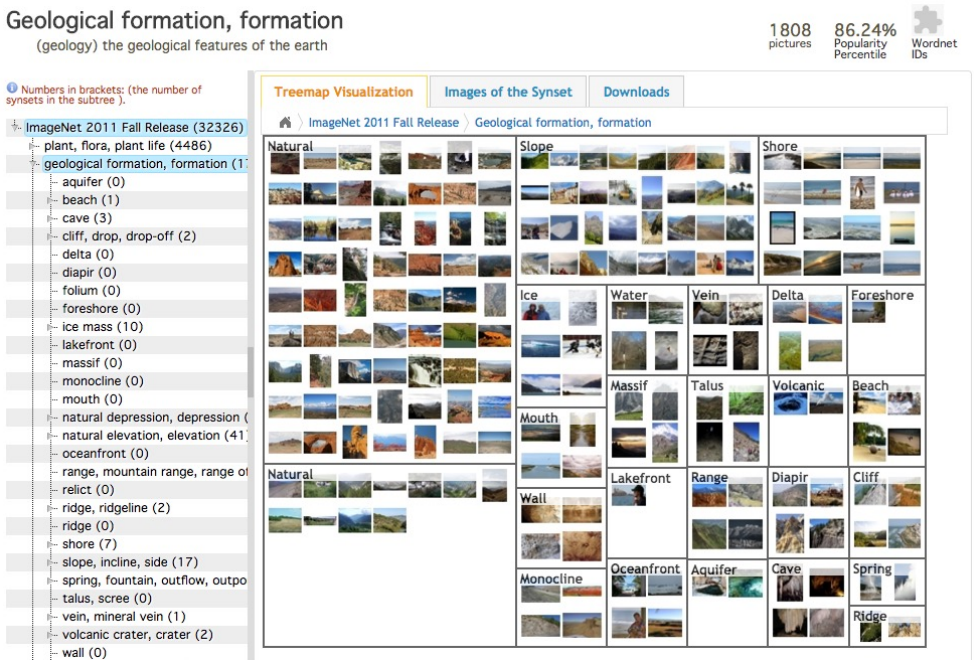
https://www.cs.toronto.edu/~kriz/cifar.html



ImageNet (2009)

- *Better algorithms need better data*

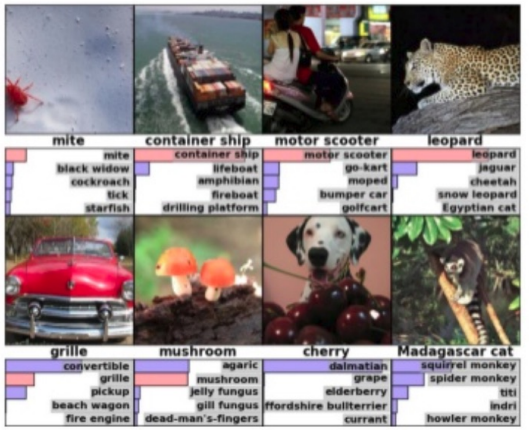
- ImageNet:
  - Goal: “map out the entire world of objects” (see this great article)
  - 3.2 million images
  - Annotated by Amazon mechanical turk
  - Hierarchical categories
  - Details in 2009 CVPR paper



Deng, J., Dong, W., Socher, R., Li, L. J., Li, K., & Fei-Fei, L. (2009, June). Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on* (pp. 248-255). IEEE.

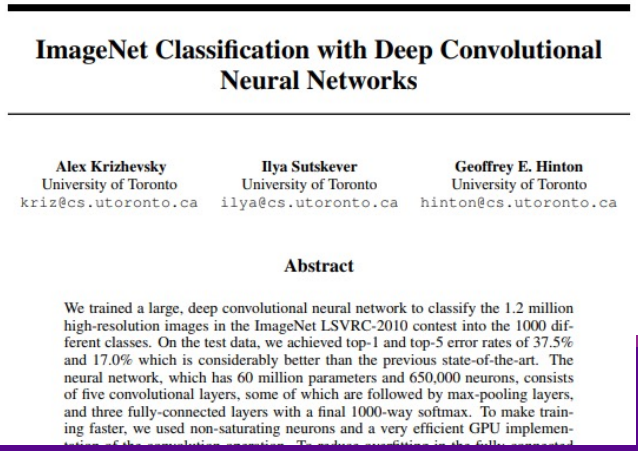
ILSVRC

- ImageNet Large-Scale Visual Recognition Challenge held yearly 2010-2017
- Challenging!
  - Classification: 1000 classes, label 5 objects per image
  - Fine-grained classification: 120 dog categories
  - Objects in many positions, scales, rotations, lightings, occlusions . . .



Deep Networks Enter in 2012

- 2012: Stunning breakthrough by the first deep network
- "AlexNet" from Hinton Group at U Toronto
- Easily won ILSVRC competition
  - Top-5 error rate: 15.3%
  - Second place: 25.6%
- Soon, all competitive methods are deep networks



ImageNet Classification with Deep Convolutional Neural Networks

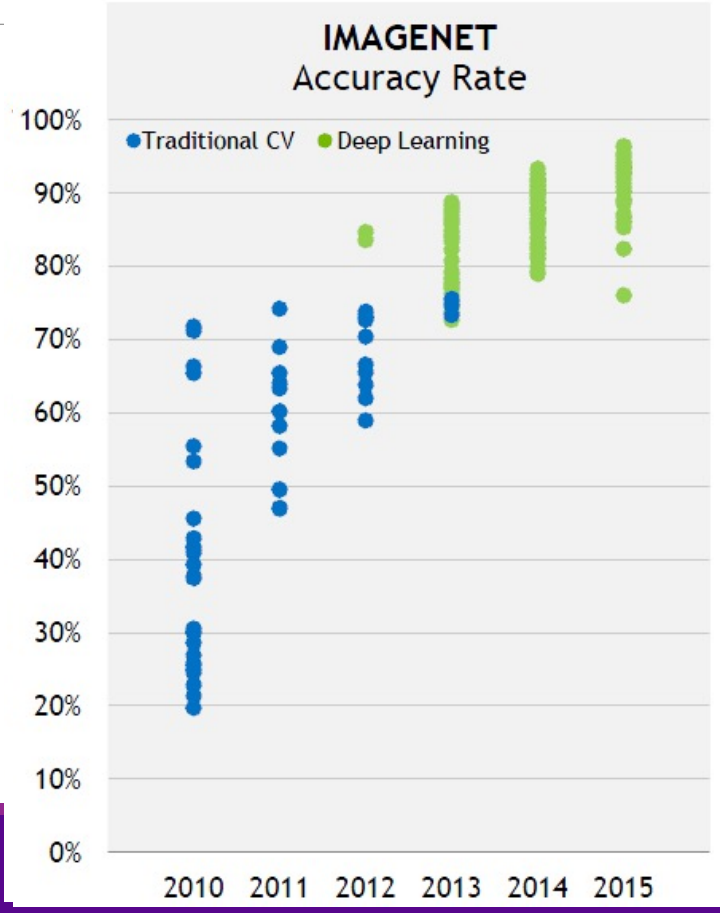
Alex Krizhevsky University of Toronto kriz@cs.utoronto.ca

Ilya Sutskever University of Toronto ilya@cs.utoronto.ca

Geoffrey E. Hinton University of Toronto hinton@cs.utoronto.ca

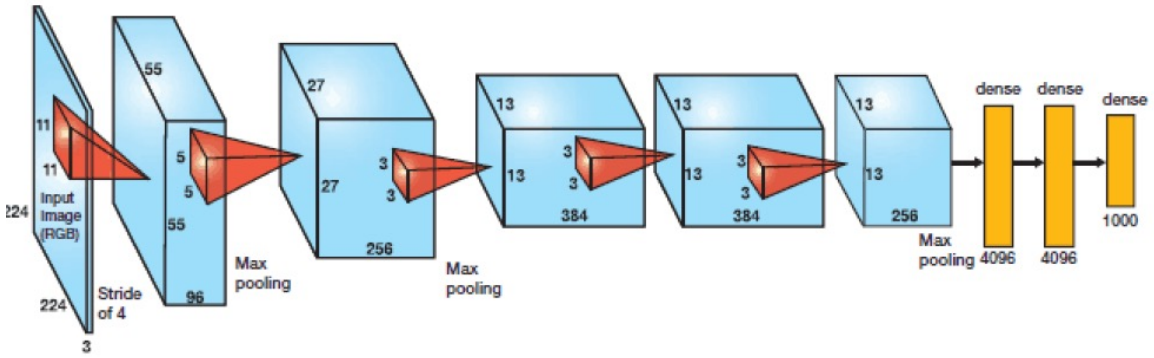
Abstract

We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes. On the test data, we achieved top-1 and top-5 error rates of 37.5% and 17.0% which is considerably better than the previous state-of-the-art. The neural network, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax. To make training faster, we used non-saturating neurons and a very efficient GPU implemen-



Alex Net

- Key idea: Build a very deep neural network
- 60 million parameters, 650000 neurons
- 5 conv layers + 3 FC layers
- Final is 1000-way softmax



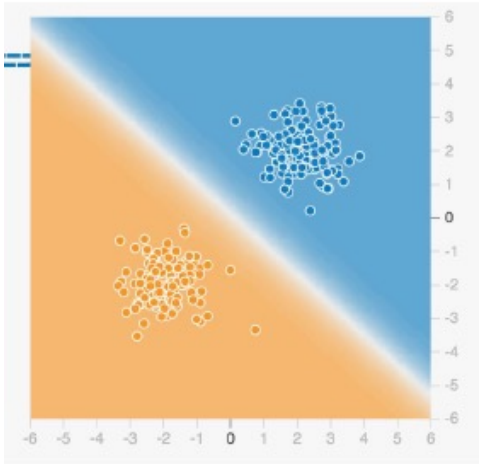
Outline

- ❑Motivation: ImageNet Large-Scale Visual Recognition Challenge (ILSVR)
- ❑Deep Networks and Feature Hierarchies
- ❑2D convolutions
- ❑Convolutional neural networks
- ❑Creating and visualizing convolutional layers in Keras
- ❑Training CNNs: Backpropagation, Batch-norm, Dropout, etc.
- ❑Transfer Learning from Famous Pre-trained Networks

A One Layer Network

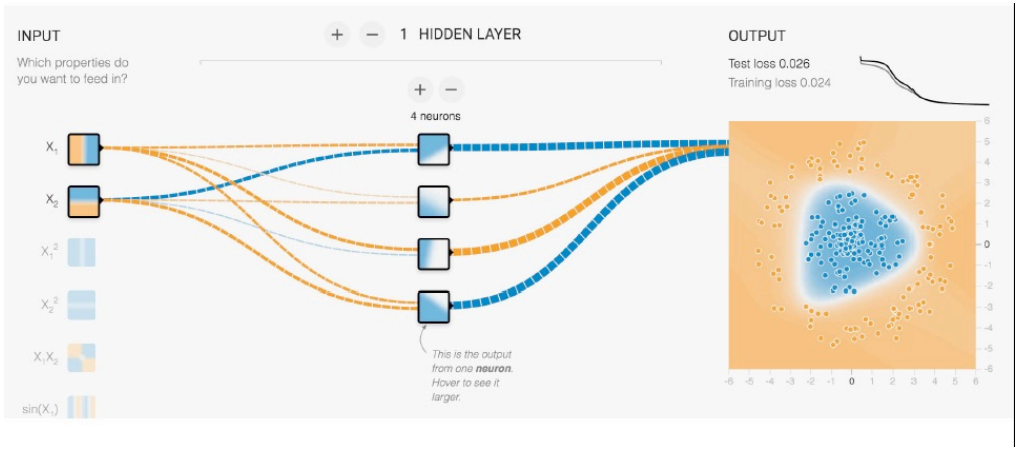
- A 1-layer neural net (e.g. linear classifier) computes an activation

$$\mathrm{u} = g_{act}(z) = \frac{1}{1+e^{-z}} \text{ from linear score } z = \boldsymbol{w}^T\boldsymbol{x} + b$$



- The unit divides input space into two half spaces
  - The half spaces are linearly separated
  - Figure shows $d = 2$ case with $\boldsymbol{x} = [x_1, x_2]$
- We say that output u is tuned to $\boldsymbol{x}$ when output is largest:
  - Output u is largest when z is largest
  - $z = \boldsymbol{w}^T\boldsymbol{x} + b = \|\boldsymbol{w}\|\|\boldsymbol{x}\| \cos\theta + b$
  - So, output z is largest when $\cos\theta = 0 \Rightarrow$ Input $\boldsymbol{x} = C\boldsymbol{w}$
  - Thus, input should be aligned with weights $\boldsymbol{w}$

Two Layer Network

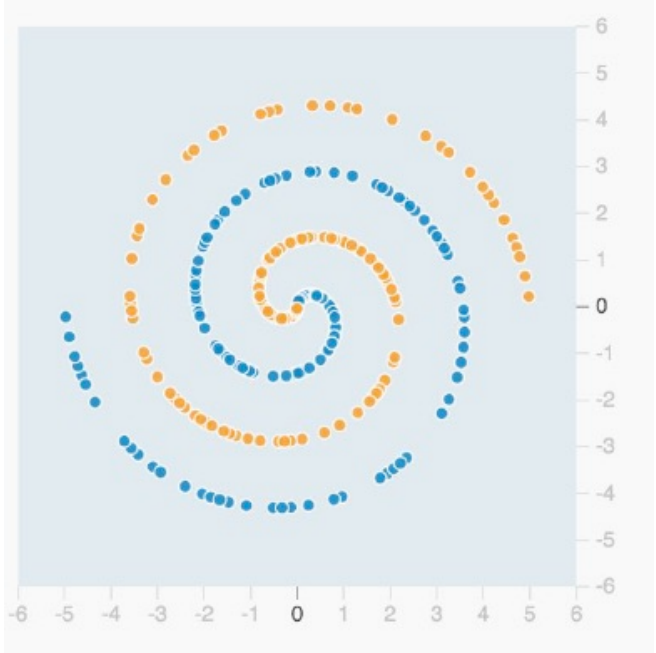


- Two-layer neural networks:
  - Implement nonlinear boundaries
  - Built from intersections of linear regions

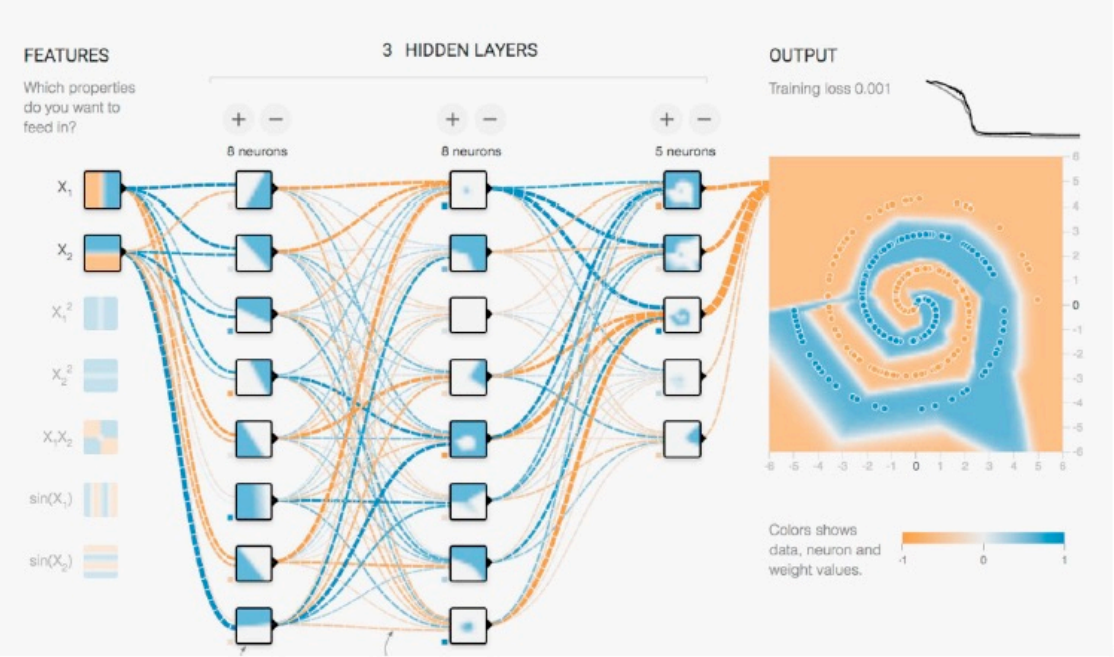
- Picture to left:
  - Output of Tensorboard
  - Tool in TensorFlow
  - Provided for visualizing neural nets

From Kaz Sato, “Google Cloud Platform Empowers TensorFlow and Machine Learning”

What about a More Complicated Region?



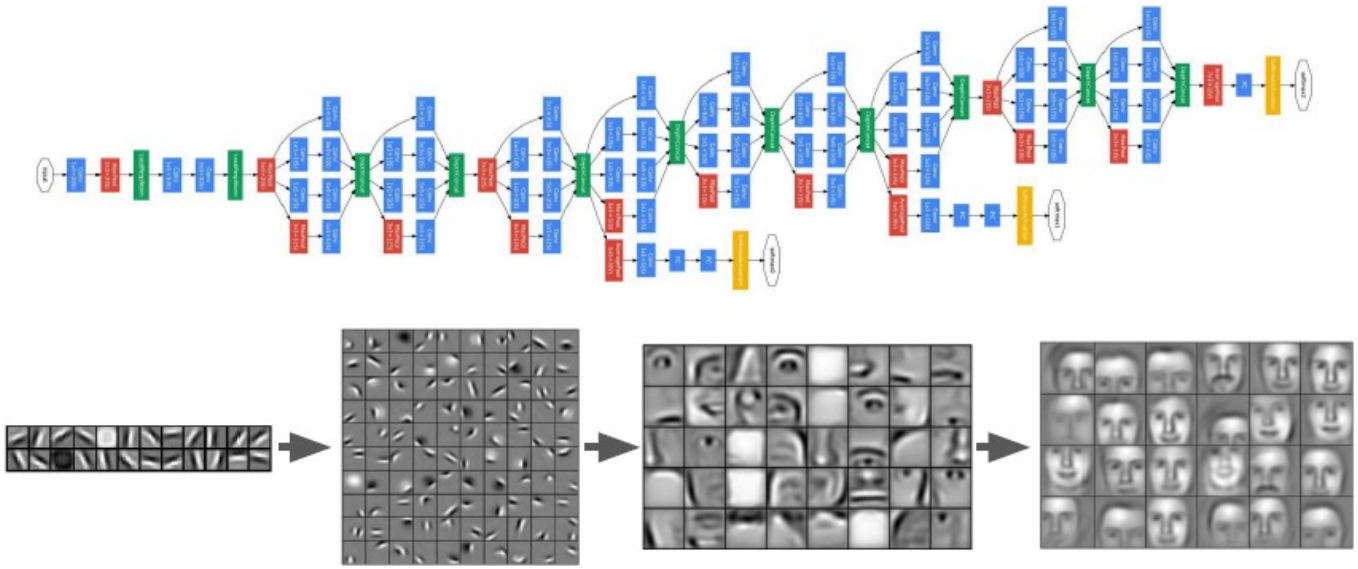
Use Multiple Layers



- More hidden layers
- Hierarchies of features
- Generate very complex shapes

Can you Classify This?

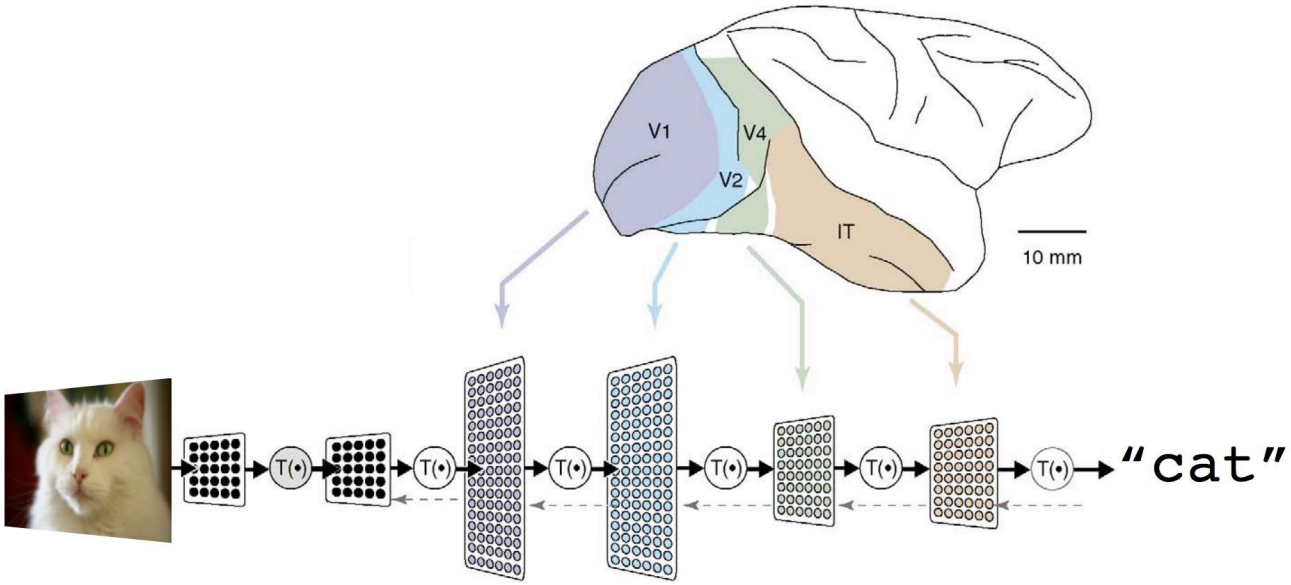
Build a Deep Neural Network



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

Biological Inspiration

- Processing in the brain uses multi-layer processing



History and Why Now?

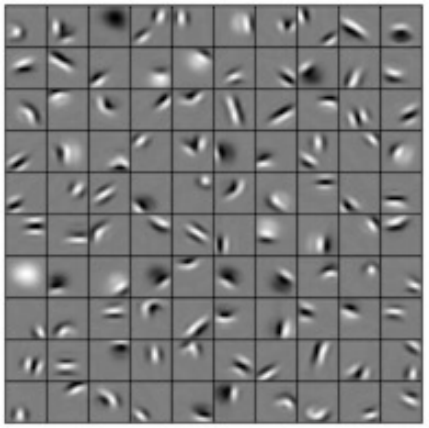
- Early history (see article):
  - Using multiple layers dates to 1965 (Ivakhenko and Lapa)
  - Convolutional networks with pooling (Fukushima, 1980)
  - Back-propagation in deep networks: (Werbos 1982, LeCun, 1989)
- But larger networks were a challenge:
  - Limited training data (Many parameters ⇒ overfitting)
  - Limited Computational power
  - Vanishing gradient: Sigmoids produce gradients ≈ 0 for large inputs. Many gradients vanish for many layers
- AlexNet with ImageNet introduced many solutions:
  - Trained on much larger dataset
  - Dropout, ReLU activations, batch-norm and other tricks…

Outline

- ❑Motivation: ImageNet Large-Scale Visual Recognition Challenge (ILSVR)
- ❑Deep Networks and Feature Hierarchies
- ❑2D Convolutions
- ❑Convolutional neural networks
- ❑Creating and visualizing convolutional layers in Keras
- ❑Training CNNs: Backpropagation, Batch-norm, Dropout, etc.
- ❑Transfer Learning from Famous Pre-trained Networks

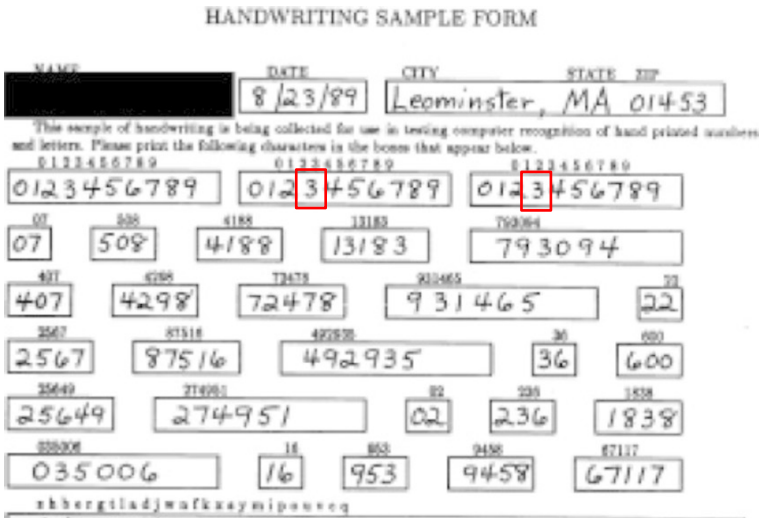
Local Features

- Early layers in deep neural networks often find local features
- Small patterns in larger image
  - Examples: Small lines, curves, edges
- Subsequent layers combine local features to create more complex features
- The features can be located anywhere
- How do we find them?



Local Features

- How do we find local features?
- A localization problem.
- Example: Find the digit “3” in the form



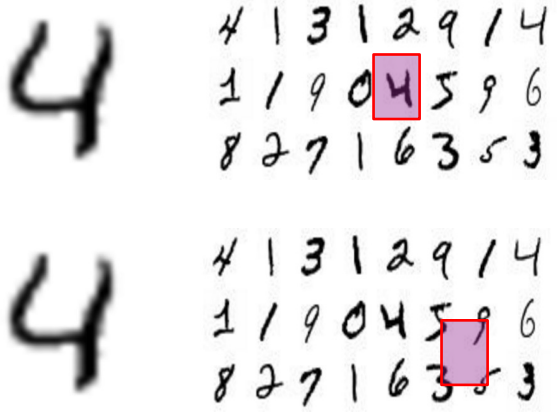
Localization via "Correlation"

- Correlation: Find local feature by sliding window
- Large image: X $N_1 \times N_2$ (e.g. 512 x 512)
- Small filter: W $K_1 \times K_2$ (e.g. 8 x 8)
- At each offset (j_1, j_2) compute:

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

  - Correlation of W with image box starting at (j_1, j_2)
  - $Z[j_1, j_2]$is large if feature is present around (j_1, j_2)

Filter $\boldsymbol{W}$ Image $\boldsymbol{X}$ $Z[j_1, j_2]$



Terminology

- In signal processing and math, convolution includes flipping:

$$z[n_1, n_2] = \sum_{k_2=0}^{K_2-1} \sum_{k_1=0}^{K_2-1} w[k_1, k_2] x[n_1 - k_1, n_2 - k_2]$$

  - Implemented in scipy.signal.convolve2d
  - For this class, we will call this convolution with reversal

- But, in many neural network packages (including Keras), convolution does not include flipping:

$$z[n_1, n_2] = \sum_{k_2=0}^{K_2-1} \sum_{k_1=0}^{K_2-1} w[k_1, k_2] x[n_1 + k_1, n_2 + k_2]$$

  - In signal processing, this is called correlation
  - Implemented in scipy.signal.correlate2d
  - Following use in deep learning, we will call this convolution

Boundary Conditions

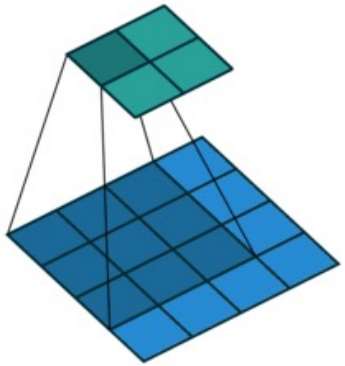
- Suppose inputs are
  - x, size $N_1 \times N_2$, w: size $K_1 \times K_2$, $K_1 \leq N_1, K_2 \leq N_2$
  - $z = x * w$ (without reversal)

$$z[n_1, n_2] = \sum_{k_2=0}^{K_2-1} \sum_{k_1=0}^{K_2-1} w[k_1, k_2] x[n_1 + k_1, n_2 + k_2]$$

- Different ways to define outputs
- Valid mode: $0 \leq n_1 < N_1 - K_1 + 1,\ 0 \leq n_2 < N_2 - K_2 + 1$
  - Requires no zero padding
- Same mode: Output size $N_1 \times N_2$
  - Usually use zero padding for neural networks
- Full mode: Output size $(N_1 + K_1 - 1) \times (N_2 + K_2 - 1)$
  - Not used often in neural networks

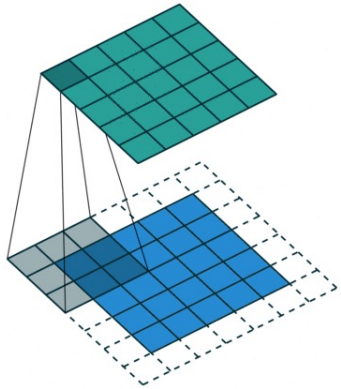
Boundary Conditions Illustrated

- There are three ways to handle boundary conditions
- See excellent github with animated gifs



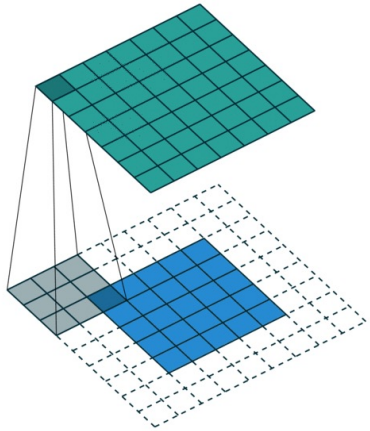
Valid mode
(no zero padding)

Output shape:
$(N_1 - K_1 + 1) \times (N_2 - K_2 + 1)$



Same mode
(half zero padding)

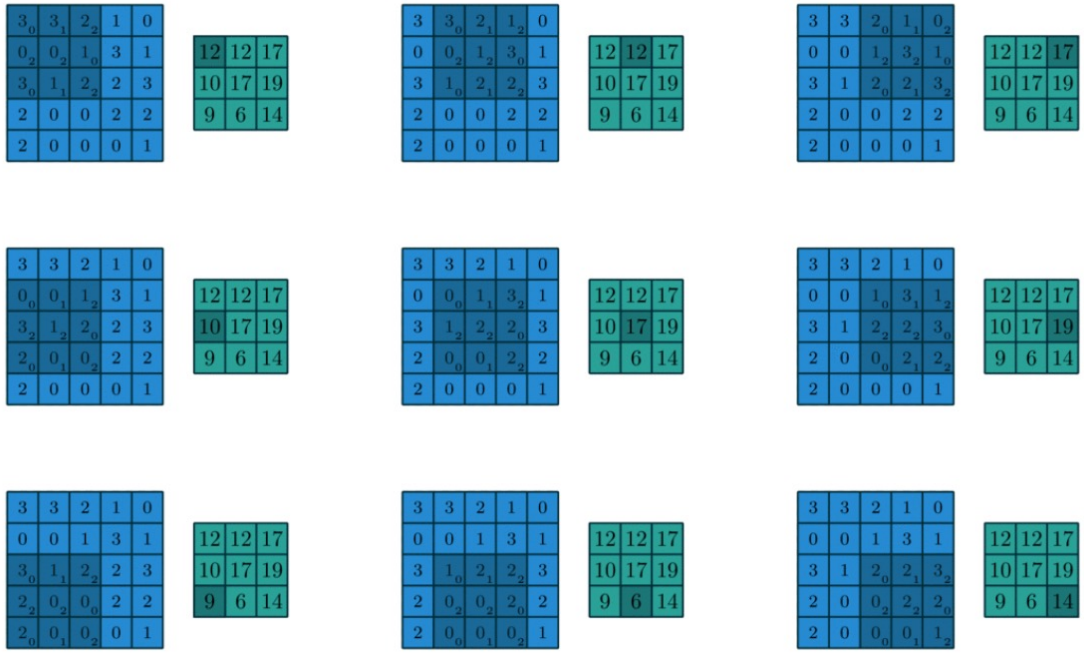
Output shape:
$N_1 \times N_2$



Full mode
(full zero padding)

Output shape:
$(N_1 + K_1 - 1) \times (N_2 + K_2 - 1)$

2D Convolution Example with Valid Mode



Kernel

$$W = \begin{bmatrix} 0 & 1 & 2 \\ 2 & 2 & 0 \\ 0 & 1 & 2 \end{bmatrix}$$

Example Convolution in Python

- We now demonstrate 2D convolution in Python
- We use the following packages:
  - skimage: Image processing
  - scipy signal: For signal processing
- We experiment with "cameraman" image
  - Famous image in image processing
  - Built-in image skimage.data
- We examine two convolutions:
  - Local averaging / blurring
  - Edge detection

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

Ex 1. Convolution for Local Averaging

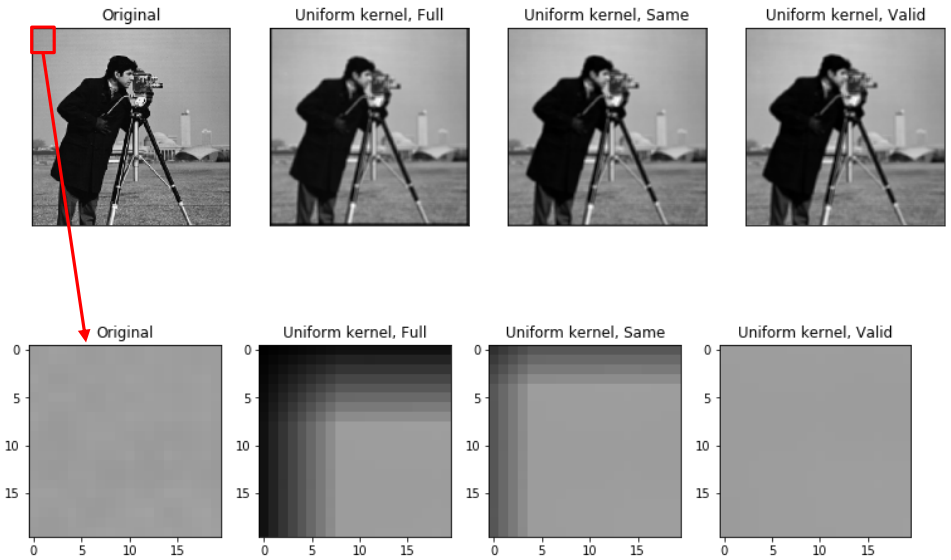
- First, consider convolving with a uniform kernel
- Each output is a local average of the input
- Visually, this blurs the image
- Amount of blurring increases with kernel size

$$\boldsymbol{G} = \frac{1}{K_x K_y}\begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix} \begin{matrix} \\ \\ \updownarrow K_y \end{matrix}$$

$$\underset{\leftrightarrow K_x}{\phantom{\frac{1}{K_x K_y}\begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}}}$$


```
kx = 9
ky = 9
sig = 3
G_unif = np.ones((kx,ky))/(kx*ky)
im_unif_full = scipy.signal.correlate2d(im, G_unif, mode='full')
im_unif_same = scipy.signal.correlate2d(im, G_unif, mode='same')
im_unif_valid = scipy.signal.correlate2d(im, G_unif, mode='valid')
```

Illustration of Boundary Conditions




```
kx = 9
ky = 9
sig = 3
G_unif = np.ones((kx,ky))/(kx*ky)
im_unif_full = scipy.signal.correlate2d(im, G_unif, mode='full')
im_unif_same = scipy.signal.correlate2d(im, G_unif, mode='same')
im_unif_valid = scipy.signal.correlate2d(im, G_unif, mode='valid')
```

```
1 print("Input shape = " + str(im.shape))
2 print("Output shape (Full) = " + str(im_unif_full.shape))
3 print("Output shape (Same) = " + str(im_unif_same.shape))
4 print("Output shape (valid) = " + str(im_unif_valid.shape))
```

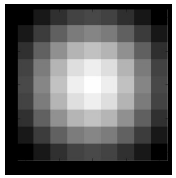
```
Input shape = (512, 512)
Output shape (Full) = (520, 520)
Output shape (Same) = (512, 512)
Output shape (valid) = (504, 504)
```

- Notice the black pixels at the boundaries when using same or full mode
- These arise from zero padding

Averaging vs. Gaussian Filtering

- Previously, we used a uniform kernel
- Now we try a Gaussian kernel
  - Standard deviation σ controls effective width
  - Choose window size $K \geq 2\sigma + 1$
- Both blur images, but differently

9x9 Gaussian blur kernel



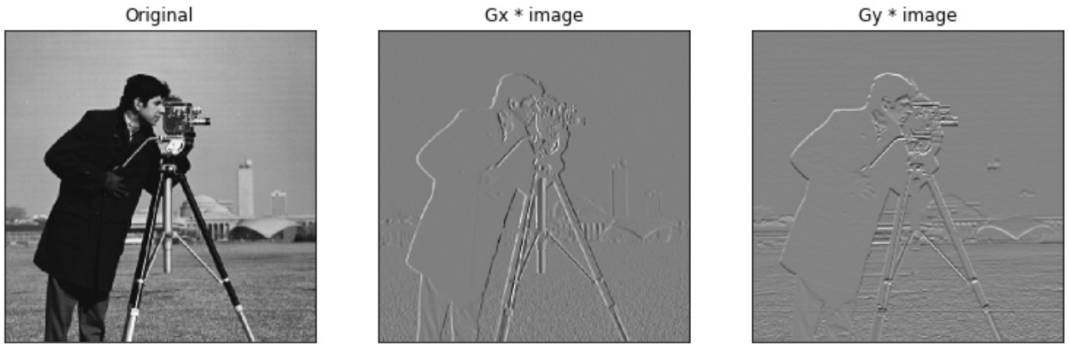
Ex 2. Convolution for Edge Detection

- We can do edge detection by convolving with a gradient filter
- For example, use Sobel filters:

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \qquad G_x = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

- Define $Z_x = G_x * X, \;\; Z_y = G_y * X$ (without reversal)
- Called gradient filters since:
  - $Z_x[i,j] = Z_y[i,j] = 0$ in areas where $X[i,j]$ is constant
  - $Z_x[i,j] =$ large positive on strong decrease in x-direction = vertical edge white → black
  - $Z_x[i,j] =$ large negative on strong increase in x-direction = vertical edge black → white
  - $Z_y[i,j]$ is similarly sensitive to horizontal edges

Edge Detection using Sobel Filter




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

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

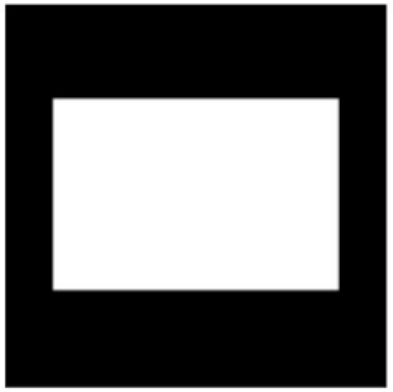
In-Class Exercise

In-Class Exercise

In this exercise you will try to build your filter.

- Display the image of `X` constructed below. If you display it correctly, you should see a white rectangle.
- Design a `4 x 4` filter, `G` such that if `Z = G*X`, `Z[i,j]=0` in the regions where `X` is constant and `Z[i,j]` takes it maximum value on the upper right corner of the rectangle.
- Display `Z = G*X` to confirm that it takes it maximum value on the top right corner.

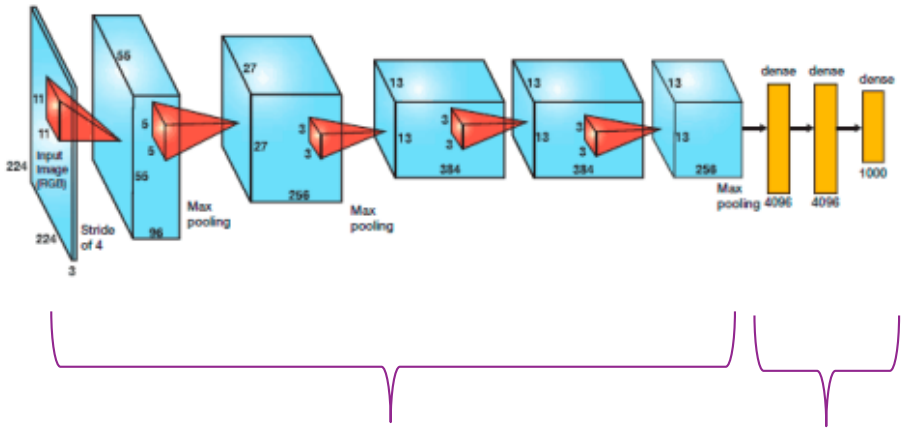
```
1 imshape = (128,128)
2 bx = 16
3 by = 32
4 X = np.zeros(imshape)
5 X[by:-by,bx:-bx] = 1
```



Outline

- ❑Motivation: ImageNet Large-Scale Visual Recognition Challenge (ILSVR)
- ❑Deep Networks and Feature Hierarchies
- ❑2D Convolutions
- ❑Convolutional Neural Networks
- ❑Creating and visualizing convolutional layers in Keras
- ❑Training CNNs: Backpropagation, Batch-norm, Dropout, etc.
- ❑Transfer Learning from Famous Pre-trained Networks

Classic CNN Structure



Convolutional layers

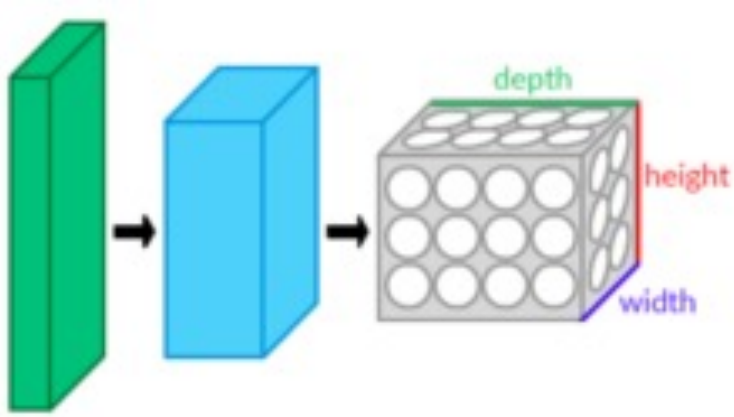
2D convolution with Activation and pooling / sub-sampling

Fully connected layers

Matrix multiplication & activation

- Starts with convolutional layers. Each layer does
  - 2D convolution with several kernels
  - Activation (e.g., ReLU)
  - Sub-sampling or pooling
- Finish with fully connected (or dense) layers. Each layer does . . .
  - Matrix multiplication
  - Activation

Tensors



- Input and output of each layer is a tensor
  - A multidimensional array
- Examples of tensors
  - Grayscale image: $(Height, Width)$
  - Color image: $(Height, Width, Chan)$
    $Chan \in \{R, G, B\}$
  - Batch of images: $(Sample, Height, Width, Chan)$
- Example: A batch of 100 color images with 256×384 pixels has shape: $(100, 256, 384, 3)$
- The number of dimensions is called the order or rank
  - Note that rank has a different meaning in linear algebra
  - So, we will use order

What Do Convolutional Layers Do?

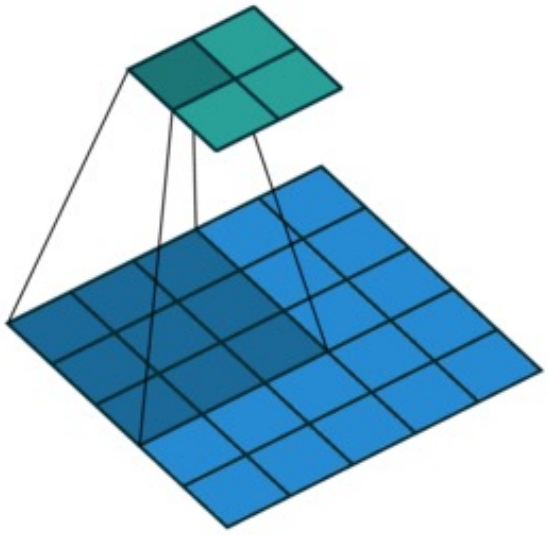
- Each convolutional layer has:
  - Weight tensor: W size $(K_1, K_2, N_{in}, N_{out})$
  - Bias vector, b: size N_{out}
- Takes input tensor U creates output tensor
- Convolutions performed over space and added over channels

$$Z[i_1, i_2, m] = \sum_{k_1=0}^{K_1-1} \sum_{k_2=0}^{K_2-1} \sum_{n=0}^{N_{in}-1} W[k_1, k_2, n, m] U[i_1 + k_1, i_2 + k_2, n] + b[m]$$

- For each output channel m, input channel n
  - Computes 2D convolution with $W[:, :, n, m]$ (2D filters of size $K_1 \times K_2$)
  - Sums results over n
  - Different 2D filter for each input channel and output channel pair

Subsampling and Pooling

- After convolution and activation, there is often a data-reduction
- There are many options here. Some popular ones are . . .
- Subsampling:
  - keep the top-left pixel from 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 the largest value in each $K \times K$ region
  - Shift the region by stride S horizontally & vertically
- Average pooling:
  - Keep the average value in 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

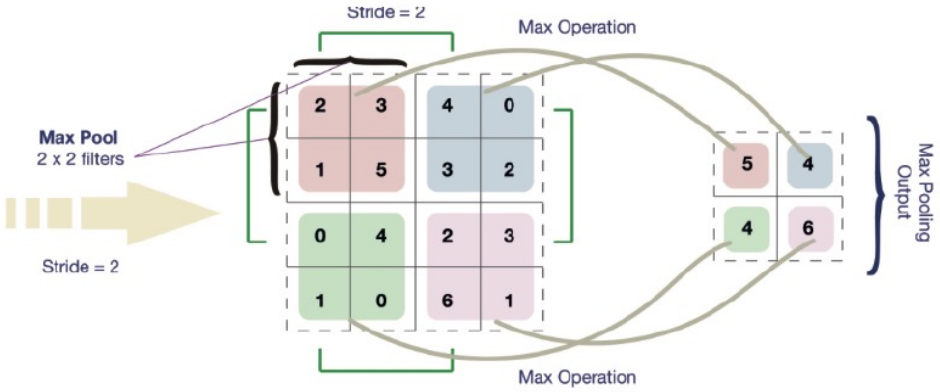


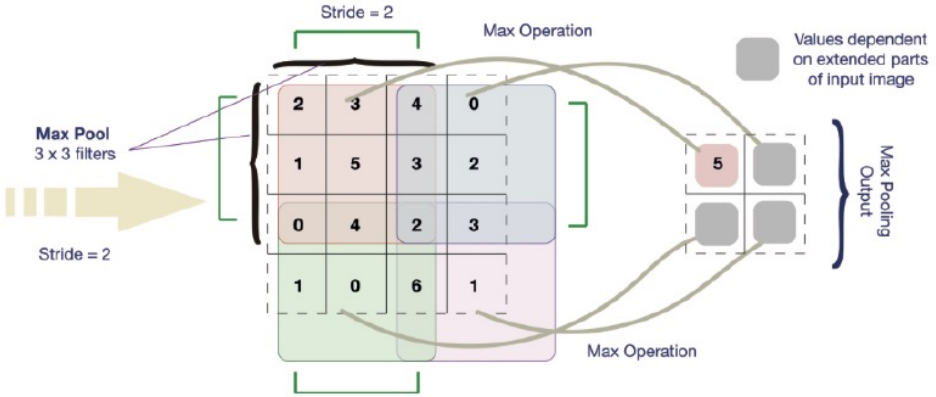
Stride S=2

Max Pooling Illustrated

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, N_1, N_2, C)
- Just before the first dense layer, we flatten (i.e., reshape) into matrix $\boldsymbol{U}$
  - Shape is (B, N_{in}), $N_{in} = N_1 N_2 C$
- Then output is performed with matrix multiplication:

$$Z[i,k] = \sum_{j=1}^{N_{in}} W[j,k]\, U[i,j] + b[k], \qquad k = 0, \ldots, N_{out}$$

  - Weights W: shape (N_{in}, N_{out})
  - Bias b: Shape $(N_{out},)$
- Same as the linear stages of the 2-layer neural network from the last unit!

Convolution vs Fully Connected

- Using convolution layers greatly reduces number of parameters
- Ex: Suppose input is $(*, N_1, N_2, N_{in})$ output is $(*, M_1, M_2, N_{out})$
  - Ex: AlexNet 2nd layer $(*, 55, 55, 96) \rightarrow (*, 55, 55, 256)$
- Convolutional network with (K_1, K_2) size filters
  - Requires $K_1 K_2 N_{in} N_{out}$ weights and N_{out} biases
  - Example: AlexNet 2nd layer with $K_1 = K_2 = 5$ filters has $6.1(10)^5$ weights and 256 biases
- But a fully-connected layer with same size inputs and outputs:
  - Would require $N_1 N_2 N_{in} M_1 M_2 N_{out}$ weights and $N_1 N_2 N_{out}$ biases
  - Example: AlexNet 2nd layer would need $2.2(10)^{11}$ weights and $7.7(10)^5$ biases
- Convolutional layers exploit translation invariance
  - Local features are small and could be located

Outline

- ❑Motivation: ImageNet Large-Scale Visual Recognition Challenge (ILSVR)
- ❑Deep Networks and Feature Hierarchies
- ❑2D convolutions
- ❑Convolutional neural networks
- ❑Creating and visualizing convolutional layers in Keras
- ❑Training CNNs: Backpropagation, Batch-norm, Dropout, etc.
- ❑Transfer Learning from Famous Pre-trained Networks

Creating Convolutional Layers in Keras

- Done easily with Conv2d
  - Specify input_shape (if first layer), kernel size and number of output channels

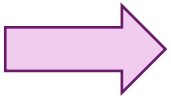
- To illustrate:
  - We create a network with a single convolutional layer
  - Set the weights and biases (normally these would be learned)
  - Run input through the layer (using the predict command)
  - Look at the output

```
# Create network
K.clear_session()
model = Sequential()
model.add(Conv2D(input_shape=input_shape,filters=nchan_out,
                 kernel_size=kernel_size,name='conv2d'))
```

Example 1: Gradients of a BW image

- Create simple convolutional layer
- Input: BW image, $N_{in} = 1$ input channel
- Two output channels: x- and y-gradient, $N_{out} = 2$

$$* \quad G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \qquad G_x = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$



Input.
One channel
Shape = (512,512,1)

Filters
Two gradient

Create a Layer in Keras

```
K.clear_session()
model = Sequential()
kernel_size = Gx.shape
nchan_out = 2
model.add(Conv2D(input_shape=input_shape,filters=nchan_out,
                 kernel_size=kernel_size,name='conv2d'))
```

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 510, 510, 2)       20
=================================================================
Total params: 20
Trainable params: 20
Non-trainable params: 0
_________________________________________________________________
```

- Create a single layer model
- Use the Conv2D layer
- Specify
  - Kernel size
  - Number of output channels
  - Input shape
- Why do we have 20 parameters?

Set the Weights

```
layer = model.get_layer('conv2d')
W, b = layer.get_weights()
print("W shape = " + str(W.shape))
print("b shape = " + str(b.shape))
```

```
W shape = (3, 3, 1, 2)
b shape = (2,)
```

- Read the weights and the shapes

```
W[:,:,0,0] = Gx
W[:,:,0,1] = Gy
b = np.zeros(nchan_out)
layer.set_weights((W,b))
```

- Set the weights to the two filters
- Normally, these would be trained

```
x = im.reshape(batch_shape)
y = model.predict(x)
```

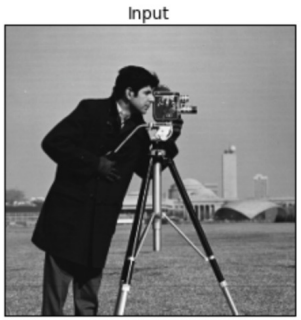
- Run the input through the network

Perform Convolution in Keras

- Create input x
  - Need to reshape
- Use predict command to compute output
- Generates two output channels y

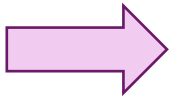
```
x = im.reshape(batch_shape)
y = model.predict(x)
```

Input x

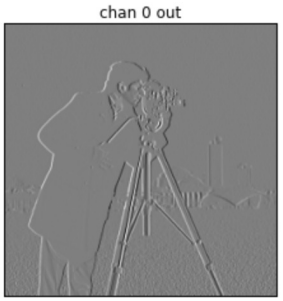


*

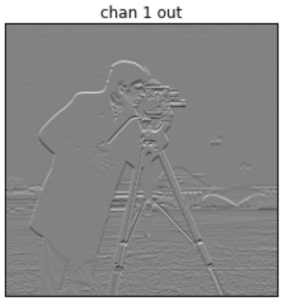
Filters
Two gradients



y[:,:,0]



y[:,:,1]

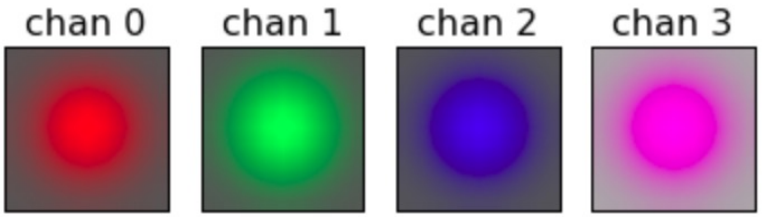


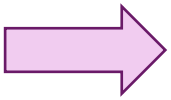
Example 2: Color Input

- Input: Single color input
  - $N_{in} = 3$ input channels
  - Input size per sample = 368 x 487 x 3
- Output: Filter with four different color filters
  - Each kernel is 9 x 9
  - $N_{out} = 4$ output channels

```
Image shape is (368, 487, 3)
```

*





Create the Layer in Keras

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

# Create network
K.clear_session()
model = Sequential()
model.add(Conv2D(input_shape=input_shape,filters=nchan_out,
                 kernel_size=kernel_size,name='conv2d'))
```

```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 360, 479, 4)       976
=================================================================
Total params: 976
Trainable params: 976
Non-trainable params: 0
_________________________________________________________________
```

- Model with single layer
- Input shape = (368, 488,3)
- Output shape = (360,479,4)
- What the number of parameters is 976?

Set the Weights

```
# 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 mix
])

# Gaussian kernel over space
krow, kcol = kernel_size
G = gauss_kernel(krow,kcol,sig=2)

# Multiply by weigthing color
W = G[:,:,None,None]*color_wt.T[None,None,:,:]
b = np.zeros(b.shape)
layer.set_weights((W,b))
```

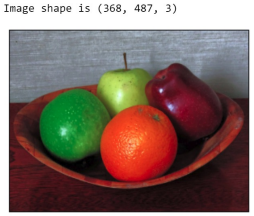
- Consider weight of the form:

$$W[i,j,k,\ell] = G[i,j]C[\ell,k]$$

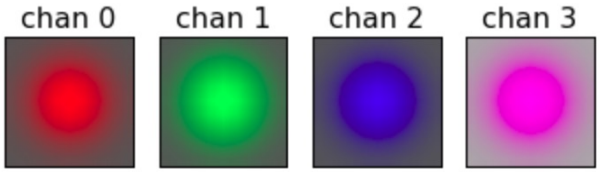
- $G[i,j]$= filter over space
  - Use Gaussian blur
- $C[\ell,k]$ = filter over channel
  - Weighting of color k in output channel ℓ
- Each filter:
  - Average over space and selects color
- Again, normally we would train the weights

Perform Convolution

Input, x
(3 channels)

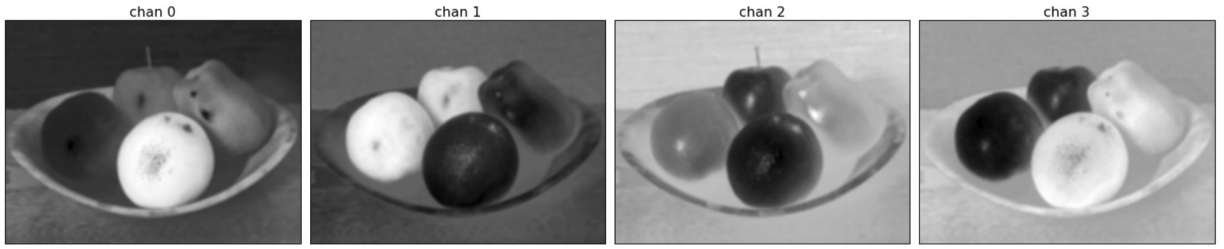


*



Filters, W
(9,9,3,4)

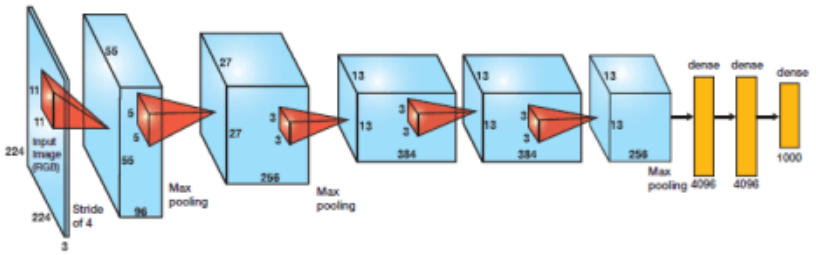
```
y = model.predict(x)
```

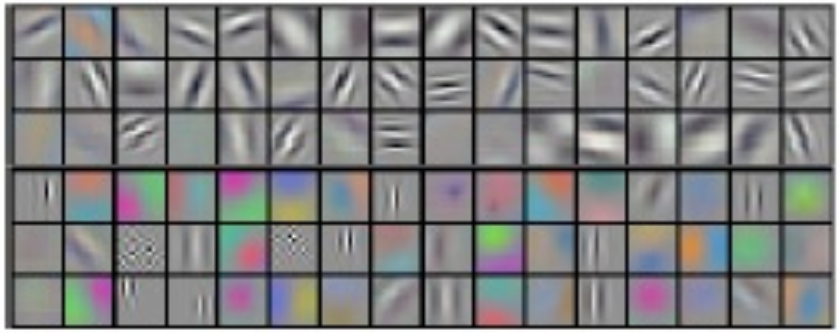


Output y
4 channels

`y[:,:,0]` `y[:,:,1]` `y[:,:,2]` `y[:,:,3]`

First Layer Filters in AlexNet





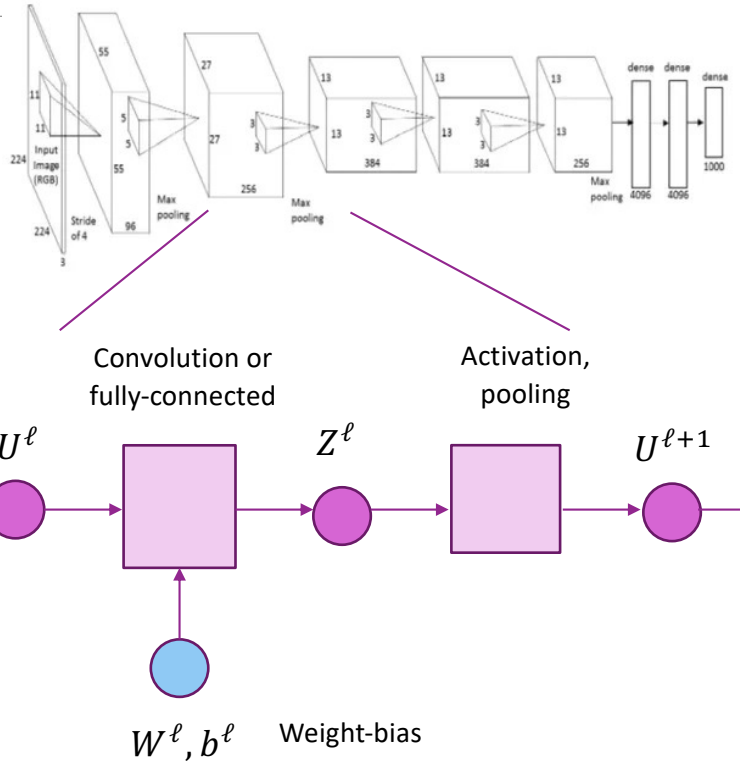
- AlexNet first layer
  - 96 filters
  - Size 11 x 11 x 3
  - Applied to image of 224 x 224 x 3
- Each filter can be viewed as a 11x11x3 image
- Selective to basic low-level features
  - Curves, edges, color transitions, …

Outline

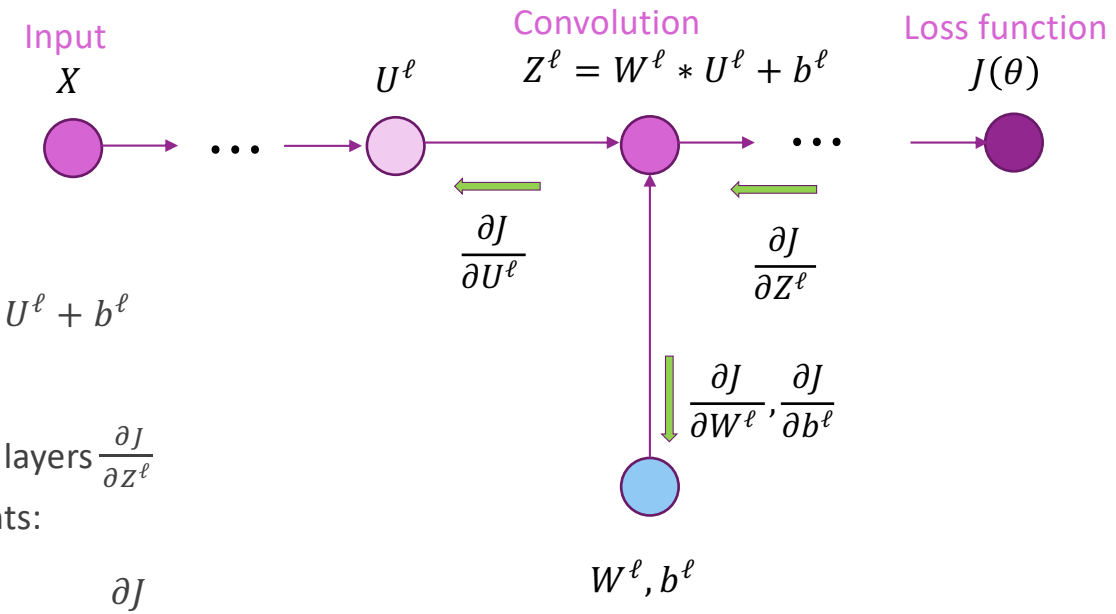
- ❑Motivation: ImageNet Large-Scale Visual Recognition Challenge (ILSVR)
- ❑Deep Networks and Feature Hierarchies
- ❑2D convolutions
- ❑Convolutional neural networks
- ❑Creating and visualizing convolutional layers in Keras
- ❑Training CNNs: Backpropagation, Batch-norm, Dropout, etc.
- ❑Transfer Learning from Famous Pre-trained Networks

Indexing Multi-Layer Networks

- Similar to single layer NNs
  - But must keep track of layers
- Consider batch of image inputs:
  - $X[i, j_1, j_2, n]$, (sample,row,col,channel)
- Input tensor at layer ℓ:
  - $U^{\ell}[i, j_1, j_2, n]$ for convolutional layer
  - $U^{\ell}[i, n]$ for fully connected layer
- Output tensor from linear transform:
  - $Z^{\ell}[i, j_1, j_2, , n]$ or $Z^{\ell}[i, n]$
- Output tensor after activation / pooling:
  - $U^{\ell+1}[i, j_1, j_2, , n]$ or $U^{\ell+1}[i, n]$



Back-Propagation in Convolutional Layers



- Convolutional layer in forward path

$$Z^\ell = W^\ell * U^\ell + b^\ell$$

- During back-propagation:
  - Obtain gradient tensor from upstream layers $\frac{\partial J}{\partial Z^\ell}$
  - Need to compute downstream gradients:

$$\frac{\partial J}{\partial W^\ell}, \quad \frac{\partial J}{\partial b^\ell}, \quad \frac{\partial J}{\partial U^\ell}$$

Gradient Details

- Write convolution as:

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

  - Drop layer index ℓ and sample index i

- In backpropagation, we receive gradient tensor: $\frac{\partial J}{\partial Z[i, j_1, j_2, m]}$

- First compute gradient wrt weights: $\frac{\partial J}{\partial W[k_1, k_2, n, m]}$

Gradient With Respect to Weights

- Forward layer:

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

- Gradient with respect to weights:

$$\frac{\partial Z[i,j_1,j_2,m]}{\partial W[k_1,k_2,n,m]}=U[i,j_1+k_1,j_2+k_2,n]$$

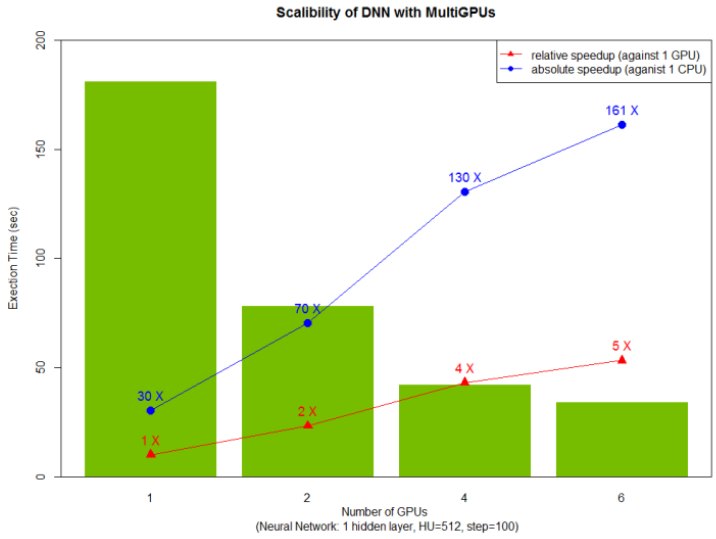
- By chain rule:

$$\begin{aligned}\frac{\partial J}{\partial W[k_1,k_2,n,m]}&=\sum_{i=0}^{B-1}\sum_{j_1=0}^{N_1-1}\sum_{j_2=0}^{N_2-1}\frac{\partial Z[i,j_1,j_2,m]}{\partial W[k_1,k_2,n,m]}\frac{\partial J}{\partial Z[i,j_1,j_2,m]}\\&=\sum_{i=0}^{B-1}\sum_{j_1=0}^{N_1-1}\sum_{j_2=0}^{N_2-1}U[i,j_1+k_1,j_2+k_2,n]\frac{\partial J}{\partial Z[i,j_1,j_2,m]}\end{aligned}$$

- Gradient wrt weights can be computed via convolution
  - Convolve input U with gradient tensor $\frac{\partial J}{\partial Z[i,j_1,j_2,m]}$
- Similar computations for gradients with respect to $\frac{\partial J}{\partial b}$, $\frac{\partial J}{\partial U}$
  - See homework

GPUs

- State-of-the-art networks involve millions of parameters
- Require enormous datasets
- Conventional processors cannot train in reasonable time
- Use Graphics Processor Units
  - Originally for graphics acceleration
  - Now essential for deep learning
- Cannot use the GPU on your laptop
- But can:
  - Rent GPU instances in cloud (~$1.20 / hour)
  - Purchase GPU workstation (~$2000)



Batch Normalization

- Important speed up method
  - When the parameters of a layer change, so does the distribution of inputs to subsequent layers.
- Recall good practice: Standardize the raw features X
- that is, for each feature:
  - Make mean = 0 and variance = 1 over each minibatch
- Batch normalization:
  - We first standarize the outputs in each channel and layer, and
  - Then learn an optimal mean & variance for them:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

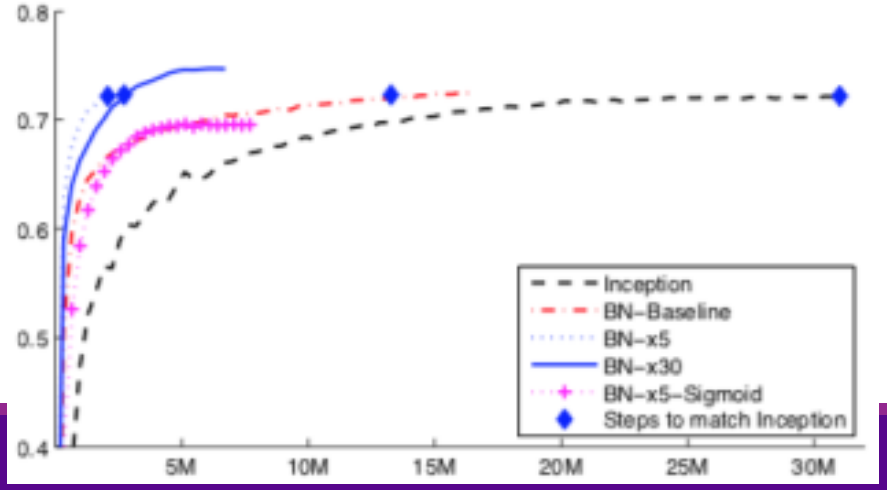
$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Sergey Ioffe, Christian Szegedy: **Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.** https://arxiv.org/pdf/1502.03167v3.pdf, 2015

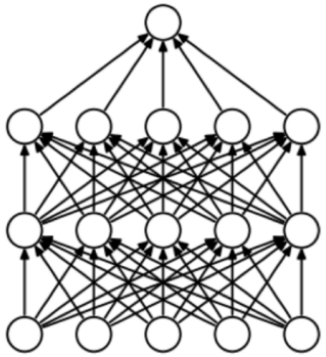
Batch Normalization

- 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 “5” and “30” refer to learning-rate increases over the baseline

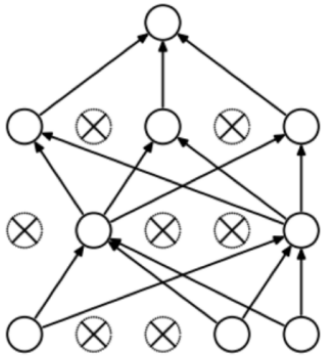


DropOut

- Dropout: Technique to reduce overfitting during training
  - Idea: temporarily remove a random subset of neurons
  - Use in fully-connected layers only!
- Motivation: Prevent co-adaptation;
  - Neurons should work well on their own
- Details:
  - Multiply each input by a Bernoulli random variable $D \in \{0, 1/p\}$
  - $P(D \neq 0) = p$ is a design parameter (usually use $p = 0.5$)
  - During test time, don't remove any units



(a) Standard Neural Net



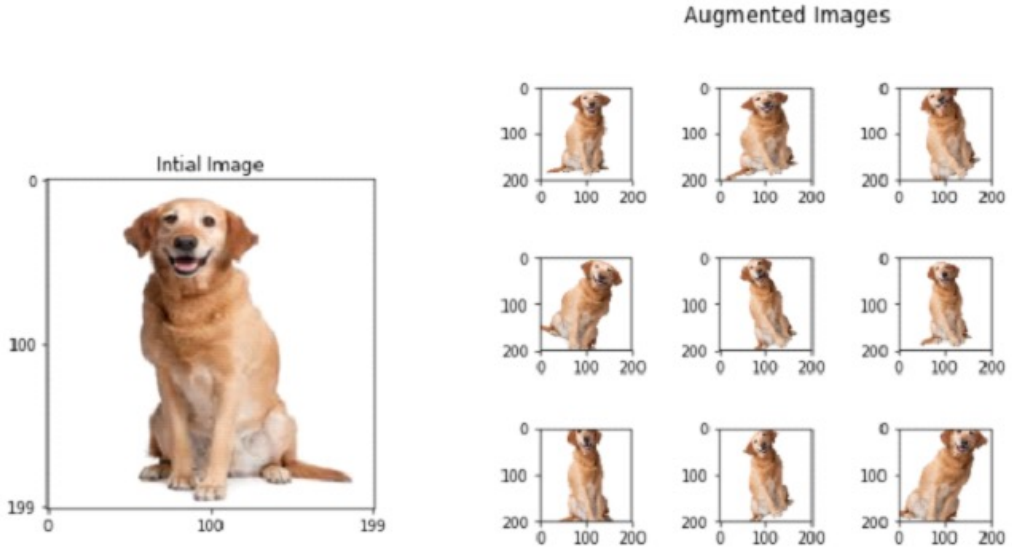
(b) After applying dropout.

Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting", JMLR 2014

- Note: increases # of training epochs, but should decrease complexity per epoch
  - First introduced in 2012, dropout seems to be getting less popular now
  - Networks are becoming deeper, and less dense layers are being usedl

Data Augmentation

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



CIFAR10 Demo

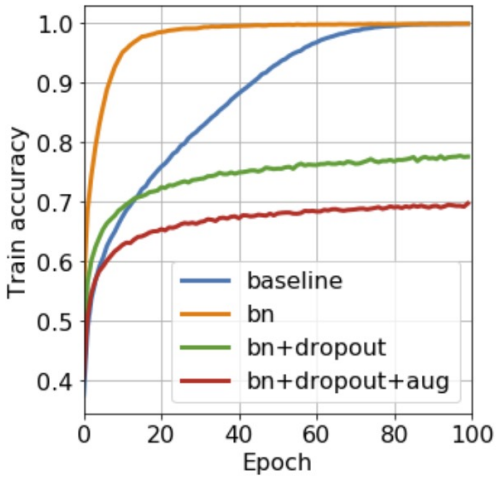
- CIFAR10 problem:
  - Classify 32x32 color images
  - 10 classes
- Test four classifiers:
  - Baseline
  - Batch norm
  - Batch norm + dropout
  - Batch norm + dropout + data aug
- Use tensorflow pre-built layers
- Full demo on github

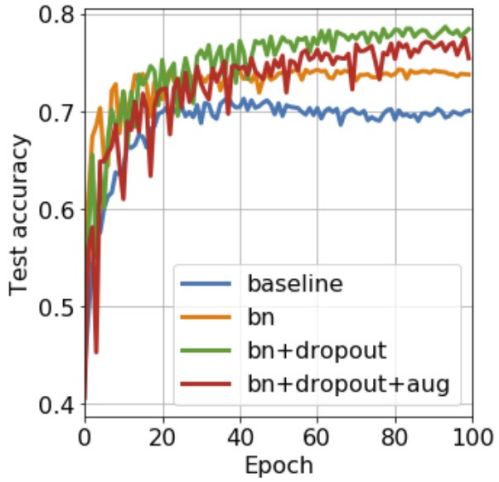
```
def create_mod(use_dropout=False, use_bn=False):
    num_classes = 10
    model = Sequential()
    model.add(Conv2D(32, (3, 3),
                     padding='valid', activation='relu',
                     input_shape=Xtr.shape[1:]))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    if use_bn:
        model.add(BatchNormalization())
    model.add(Conv2D(32, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    if use_bn:
        model.add(BatchNormalization())
    if use_dropout:
        model.add(Dropout(0.5))
    model.add(Dense(512, activation = 'relu'))
    if use_bn:
        model.add(BatchNormalization())
    if use_dropout:
        model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation='softmax'))

    return model
```

CIFAR10 Demo

- CIFAR10 problem:
  - Classify 32x32 color images
  - 10 classes
- Use simple network
  - 2 conv layers + 2 FC layers
  - RMS prop with lr=10^{-3}, decay=10^{-4}
  - Batch size=32
  - 100 epochs
- Note:
  - Exact accuracy varies with optim parameters





Method	Final test accuracy
Baseline	~70%
Batch norm	~74%
Batch norm+dropout	~78% (dropout on FC layers)
Batch norm+dropout + data augmentation	~76% (enough data)

Outline

- ❑Motivation: ImageNet Large-Scale Visual Recognition Challenge (ILSVR)
- ❑Deep Networks and Feature Hierarchies
- ❑2D convolutions
- ❑Convolutional neural networks
- ❑Creating and visualizing convolutional layers in Keras
- ❑Training CNNs: Backpropagation, Batch-norm, Dropout, etc.
- ❑Transfer Learning from Famous Pre-trained Networks

Pre-Trained Networks

- State-of-the-art networks take enormous resources to train
  - Millions of parameters
  - Often days of training, clusters of GPUs
  - Extremely expensive
- Pre-trained networks in Keras
  - Load network architecture and weights
  - Models available for many state-of-the-art networks
- Can be used for:
  - Making predictions
  - Building new, powerful networks (see lab)

Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.715	0.901	138,357,544	23
VGG19	549 MB	0.727	0.910	143,667,240	26
ResNet50	99 MB	0.759	0.929	25,636,712	168
InceptionV3	92 MB	0.788	0.944	23,851,784	159
InceptionResNetV2	215 MB	0.804	0.953	55,873,736	572
MobileNet	17 MB	0.665	0.871	4,253,864	88

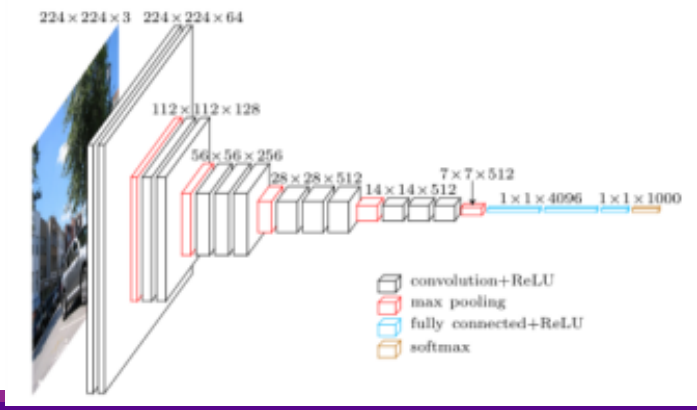
https://keras.io/applications/

VGG16

- From the Visual Geometry Group
  - Oxford, UK
- Won ImageNet ILSVRC-2014
- Based on small early filters
  - But more layers
- Remains a repectable network
- Lower lower layers are often used as feature extraction layers for other tasks

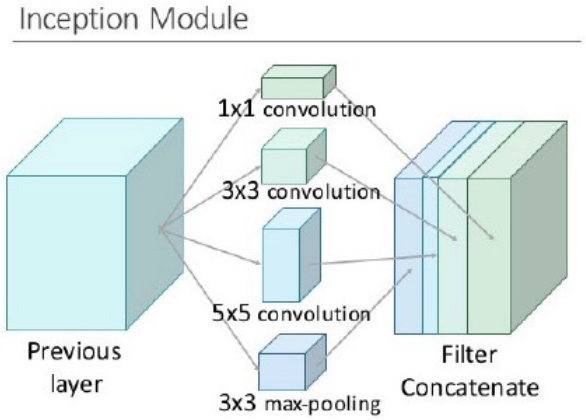
Model	top-5 classification error on ILSVRC-2012 (%)	
	validation set	test set
16-layer	7.5%	7.4%
19-layer	7.5%	7.3%
model fusion	7.1%	7.0%

K. Simonyan, A. Zisserman
Very Deep Convolutional Networks for Large-Scale Image Recognition
arXiv technical report, 2014



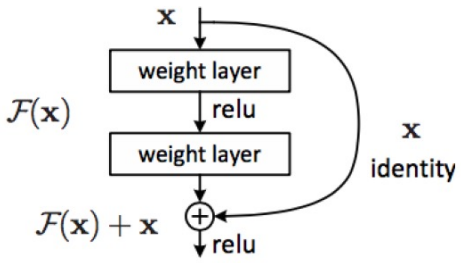
GoogleLeNet / Inception-v1

- From 2012 (AlexNet) to 2014 (VGG-16), performance got better as networks got deeper
- But the number of parameters was getting out of hand (e.g., 60 to 180 million!)
- So, Google researchers worked to design a network that was deeper, but much more efficient
  - From the film Inception (2010)
  - Used only 5 million parameters!
- Won some contests in ImageNet ILSVRC-2014
  - Achieved top-5 error of 6.7% (VGG got 7.3%)
  - 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



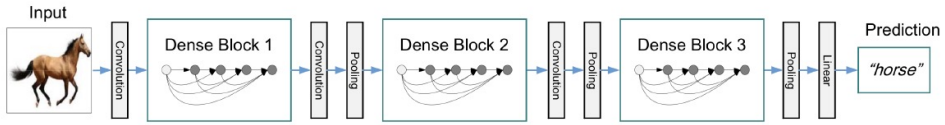
Residual Networks

- 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 deeper, even training error increase!
- A deeper network should be able to perform at least as good as a shallower network,
  - Making some layers act like identity
  - But difficult to make layers act like identity!
- Idea: Make identity easy via residual blocks:
  - Handle dimension changes w/ learned dense layer
- ResNet, introduced by Microsoft Research, 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 VGG



Other Networks in the Deep Learning Zoo

- 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)

Loading the Pre-Trained Network

```
# Load appropriate packages
from tensorflow.keras.applications.xception import Xception
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.preprocessing import image
```

```
model = Xception(weights='imagenet',input_shape=(299,299,3))
```

- Load the packages
- Create the model
  - Downloads the h5 file
  - First time, may be a while (500 MB file)

Display the Network

```
: model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         (None, 224, 224, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
fc1 (Dense)                  (None, 4096)              102764544
_________________________________________________________________
fc2 (Dense)                  (None, 4096)              16781312
_________________________________________________________________
predictions (Dense)          (None, 1000)              4097000
=================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
```

- Very deep: 16 layers (Do not count pooling layers)
- 130 million parameters!

Get Some Test Images

- Get images from the web of some category (e.g. elephants)
- Many possible sources.
  - Example: Flickr API (see Demo in github)
- Re-size / pad images so that they match expected input of VGG16
  - Input shape (224, 224, 3)




Make Predictions

```
x = preprocess_input(x)
```

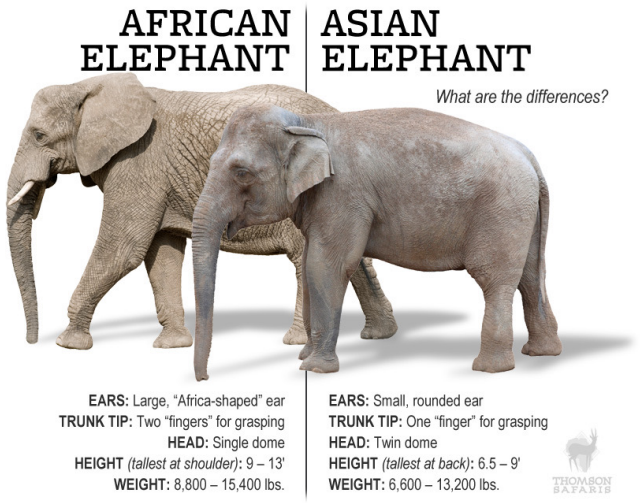
```
preds = model.predict(x)
preds_decoded = decode_predictions(preds, top=3)
```

	class 0	class 1	class 2	prob 0	prob 1	prob 2
0	Indian_elephant	African_elephant	tusker	0.776757	0.196798	0.026314
1	African_elephant	tusker	Indian_elephant	0.514596	0.414825	0.057157
2	tusker	Indian_elephant	African_elephant	0.682218	0.217784	0.099942
3	African_elephant	tusker	Indian_elephant	0.736568	0.228160	0.035263
4	African_elephant	tusker	Indian_elephant	0.409717	0.301944	0.287880
5	water_buffalo	African_elephant	warthog	0.737919	0.129731	0.037343
6	African_elephant	tusker	Indian_elephant	0.745698	0.140428	0.103136
7	Indian_elephant	tusker	African_elephant	0.970890	0.026875	0.002234
8	African_elephant	tusker	Indian_elephant	0.819497	0.108567	0.067853
9	tusker	African_elephant	Indian_elephant	0.499149	0.338156	0.162537

- Pre-process
- Predict
  - Runs input through network
- Decode predictions
  - Creates data structure for outputs

ImageNet Classification can be Hard

- Some categories differences are subtle



	class 0	class 1	class 2	prob 0	prob 1	prob 2
0	Indian_elephant	African_elephant	tusker	0.776757	0.196798	0.026314
1	African_elephant	tusker	Indian_elephant	0.514596	0.414825	0.057157
2	tusker	Indian_elephant	African_elephant	0.682218	0.217784	0.099942
3	African_elephant	tusker	Indian_elephant	0.736568	0.228160	0.035263
4	African_elephant	tusker	Indian_elephant	0.409717	0.301944	0.287880
5	water_buffalo	African_elephant	warthog	0.737919	0.129731	0.037343
6	African_elephant	tusker	Indian_elephant	0.745698	0.140428	0.103136
7	Indian_elephant	tusker	African_elephant	0.970890	0.026875	0.002234
8	African_elephant	tusker	Indian_elephant	0.819497	0.108567	0.067853
9	tusker	African_elephant	Indian_elephant	0.499149	0.338156	0.162537

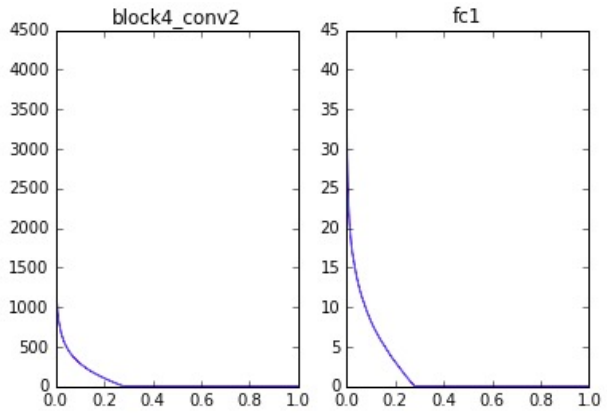
Intermediate Layers

```
from keras.models import Model

# Construct list of layers
layer_names = ['block4_conv2', 'fc1']
out_list = []
for name in layer_names:
    out_list.append(model.get_layer(name).output)

# Create the model with the intermediate layers
model_int = Model(inputs=model.input, outputs=out_list)
```

```
y = model_int.predict(x)
```



- Often need outputs of hidden layers
- Provides "latent" representation of image
  - Can be useful for other tasks
  - See lab
- In Keras, create new model
  - Specify output layers
- Predict with new model to extract hidden outputs
- Hidden layers see high level of sparsity
  - Many coefficients are zero

Output of first few convolution layers

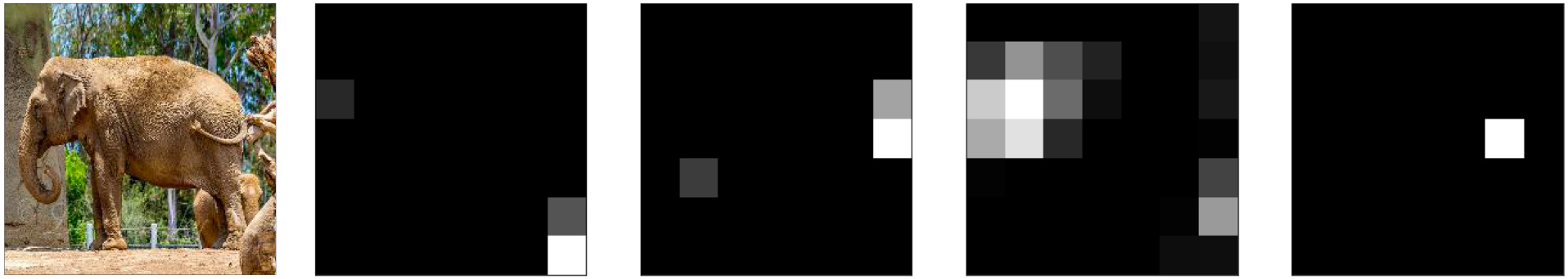
- Output of first convolution layers (first 4 channels) for a given image

First, third, and fourth channels looks like gradient images in different directions

In general, first few layers extract low level features!

Output of higher layers

- Output of last layer before the flatten layer (first 4 channels)



- These carry “high level semantic information” about object category
- See demo for outputs of other layers

Try It Yourself!

In-Class Excerise

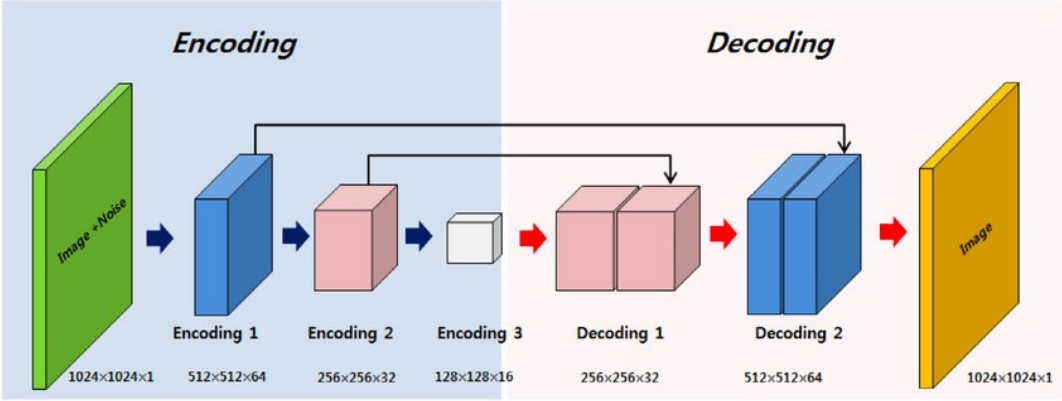
Find any image of your choice and use the pre-trained network to make a prediction.

- Download the image to your directory
- Load it into an image batch
- Predict the class label and decode the predictions

= ?

Autoencoder

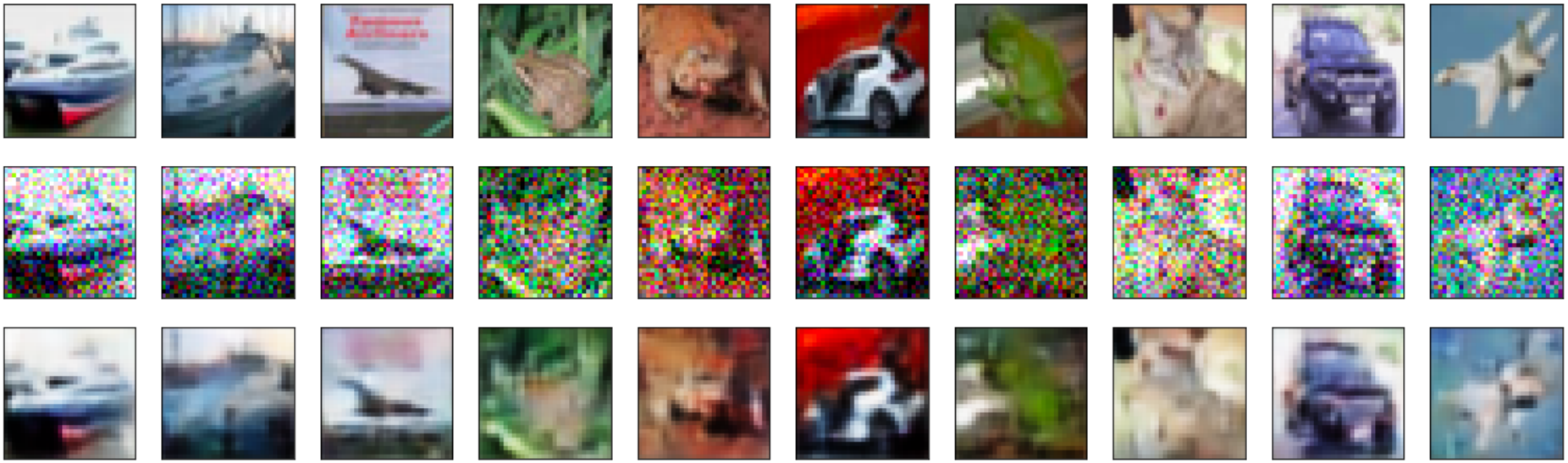
- CNN is not limited for classification!
- When all the layers are convolution, the output can have the same shape as the input (speech->speech, image->image)
- Autoencoder= Encoder+Decoder
- Encoder: image-> features;
- Decoder: features -> image



Symbol	Description	Parameters
➡	Convolution and ReLu activation	Kernel size = 3 × 3 Stride = 2 × 2
➡	Deconvolution and ReLu activation	Kernel size = 3 × 3 Stride = 2 × 2
→	Concatenate	Concatenate Encoding 1, Decoding 2 Concatenate Encoding 2, Decoding 1

Demo: Autoencoder for image denoising

- Using a subset of training samples in the cifar10 dataset
- 2 encoding layers producing 32 (or 16) channels, 3 decoding layers



Other Applications of autoencoders

- Image processing applications:
  - Image compression
  - Image segmentation
  - Saliency detection
- Other applications
  - Unsupervised feature extraction
  - Speech denoising …
  - Language translation
  - …

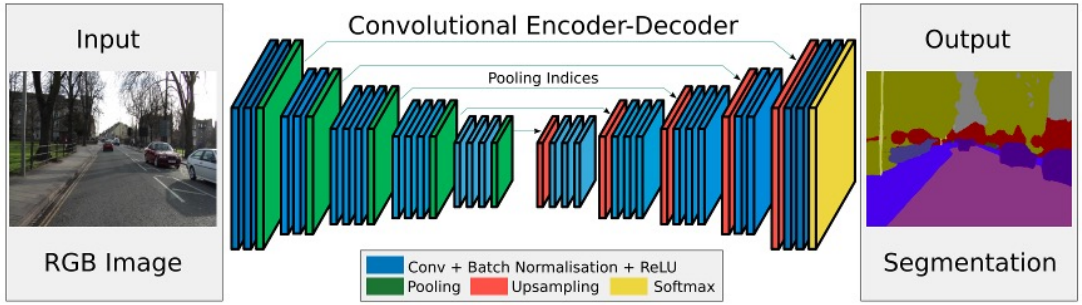


Fig. 2. An illustration of the SegNet architecture. There are no fully connected layers and hence it is only convolutional. A decoder upsamples its input using the transferred pool indices from its encoder to produce a sparse feature map(s). It then performs convolution with a trainable filter bank to densify the feature map. The final decoder output feature maps are fed to a soft-max classifier for pixel-wise classification.

https://github.com/arahusky/Tensorflow-Segmentation

- Autoencoder loss depends on the underlying application
- Using adversarial loss can help to make the output look more like the target output (beyond this class)

Lab: Transfer Learning

- For image classification or other applications, training from scratch takes tremendous resources
- Instead, can refine the VGG or other well trained networks for Task A (e.g. imagenet)
- Build a new network for Task B
  - Use early layers from first network
  - Freeze those parameters
  - Only train small number of parameters at end
- Greatly reduces number of parameters for Task B
- Can be trained on ~1000 images