Convolutional Neural Network

Topics

General and biological motivation.

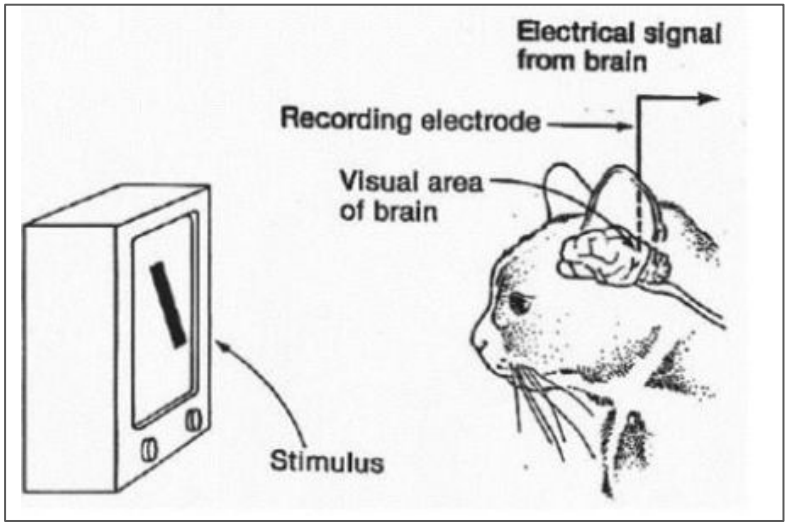
Hand-coded to learnt filters.

CNNs over fully connected networks.

Different layers in architecture (pooling, relu, etc.)

Biological motivation - Mammalian vision system.

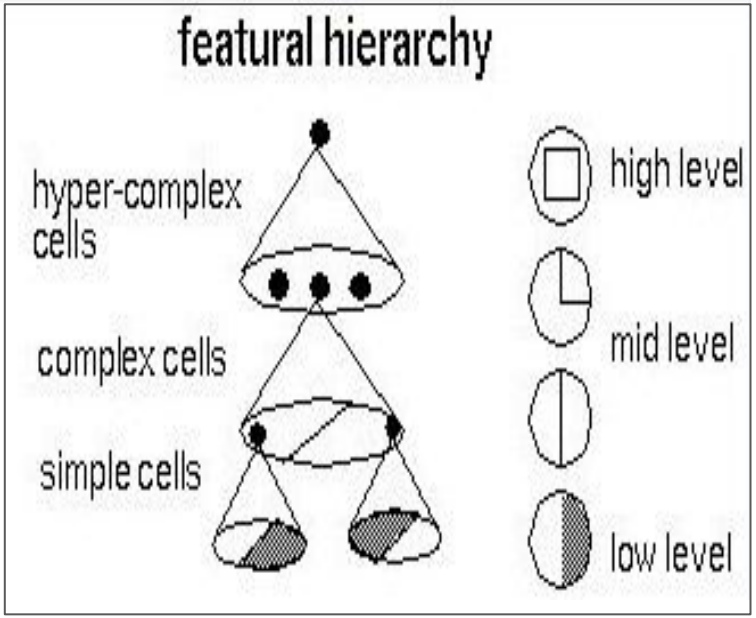
Hubel and Wiesel (1959)



Experimental setup

Suggested a ‘hierarchy’ of feature detectors in the mammalian visual cortex.

Biological motivation - Mammalian vision system.



Simple cells:

1. Activity characterized by a linear function of the image.
2. Operates in a spatially localized (SL) receptive field.
3. Each set responds to edges of different orientation.

Complex cells:

1. Operates in large SL receptive field
2. Receive input from lower level simple cells.

Hyper-complex cells:

1. Larger receptive field
2. Receive input from lower level complex cells.

Biological motivation - Grandmother cell

The grandmother cell is a hypothetical neuron that represents a complex but specific concept or object proposed by cognitive scientist Jerry Letvin in 1969.

Biological motivation - Biological NN to Artificial NN.

Neocognitron [Fukushima, Biological Cybernetics 1980]

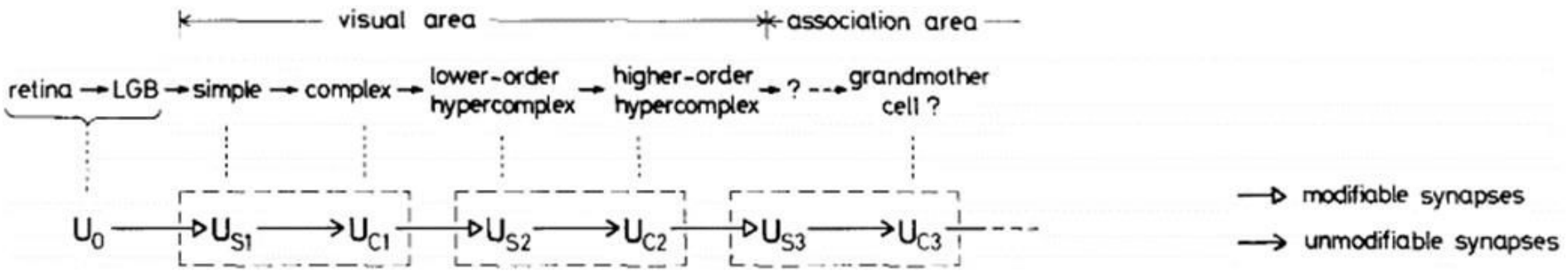
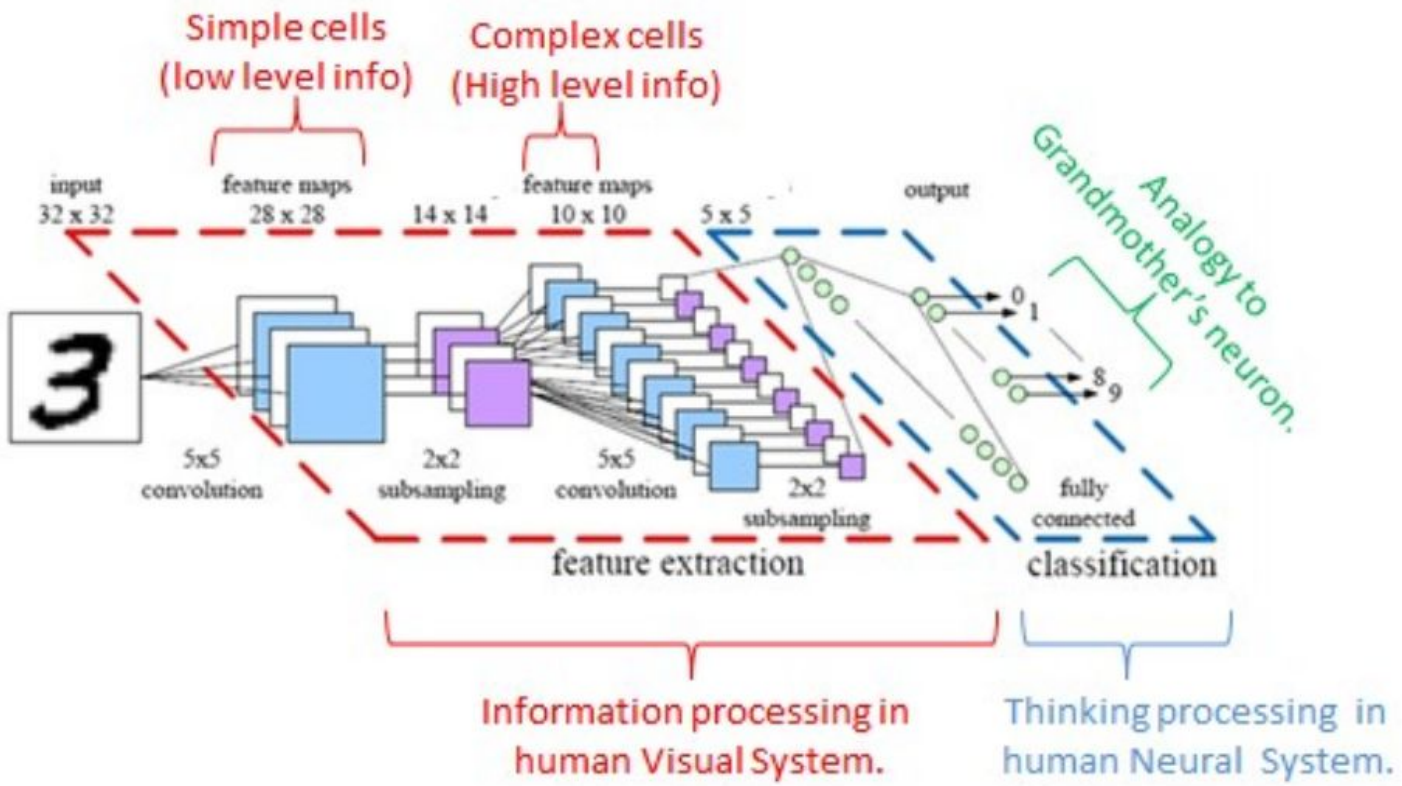


Fig. 1. Correspondence between the hierarchy model by Hubel and Wiesel, and the neural network of the neocognitron

1. But neuroscience has told us relatively less about how to train networks.
2. Neocognitron used layer-wise unsupervised clustering algorithm.

Biological motivation - CNN.

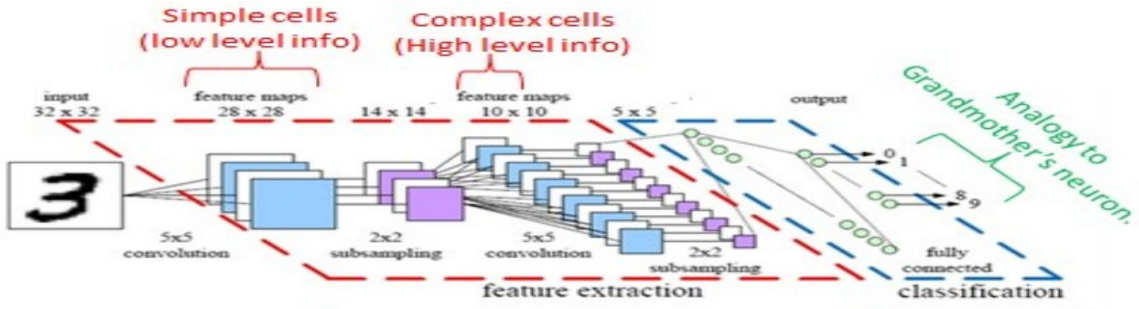
Back-propagation [Lang and Hinton, 1988], and modern CNN [LeCun *et al*., 1989]

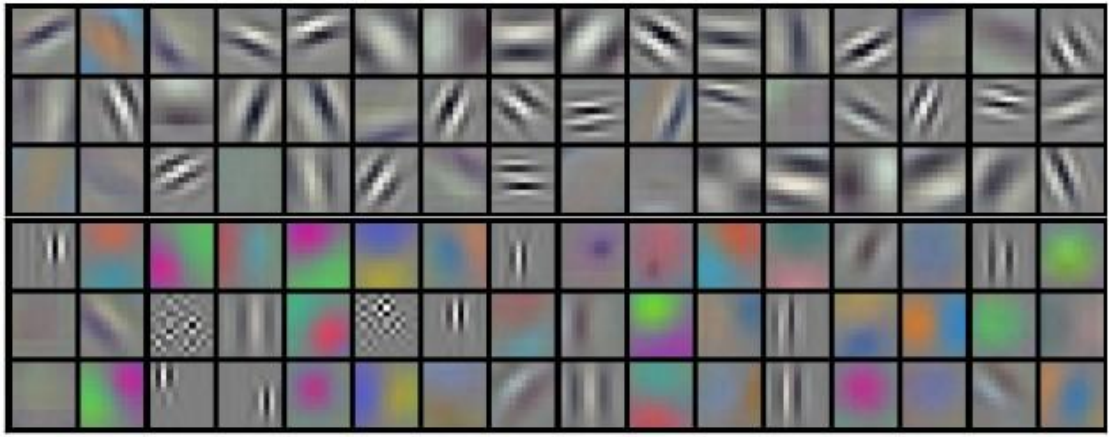


CNN proposed by LeCun *et al.* for document recognition.

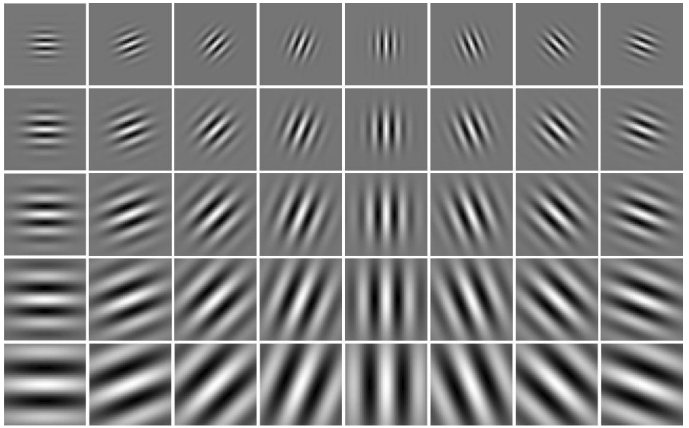
Simple cells and low-level filters in a CNN

Marčelja, S. [1980] suggests that simple cells in visual cortex can be modeled as **gabor filters**.



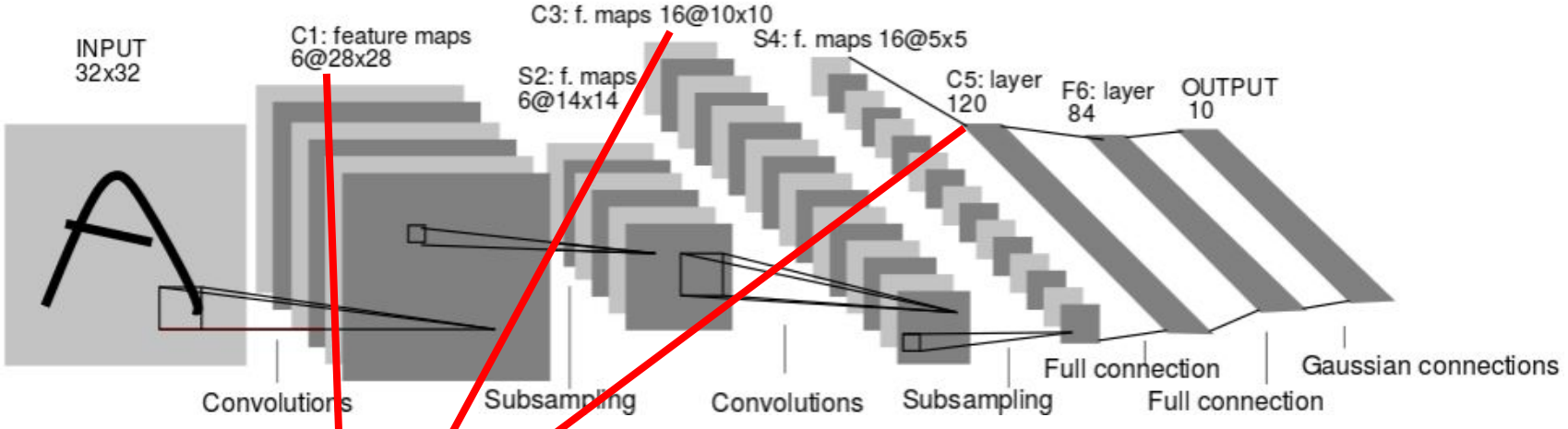


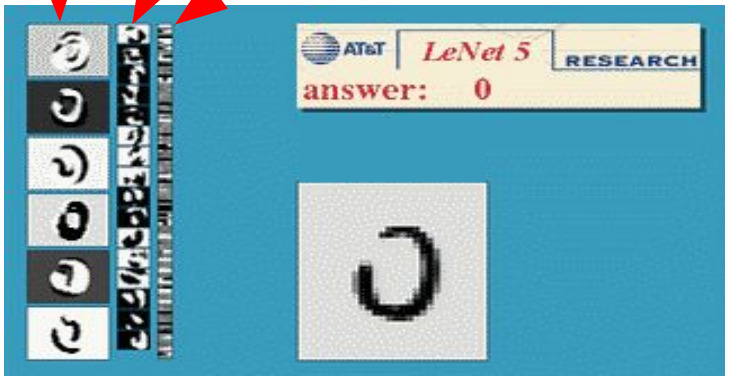
Low-level learnt filters of CNN (from Alexnet, 2012)



Gabor filters

CNN for document recognition [LeCun *et al*., 1989].





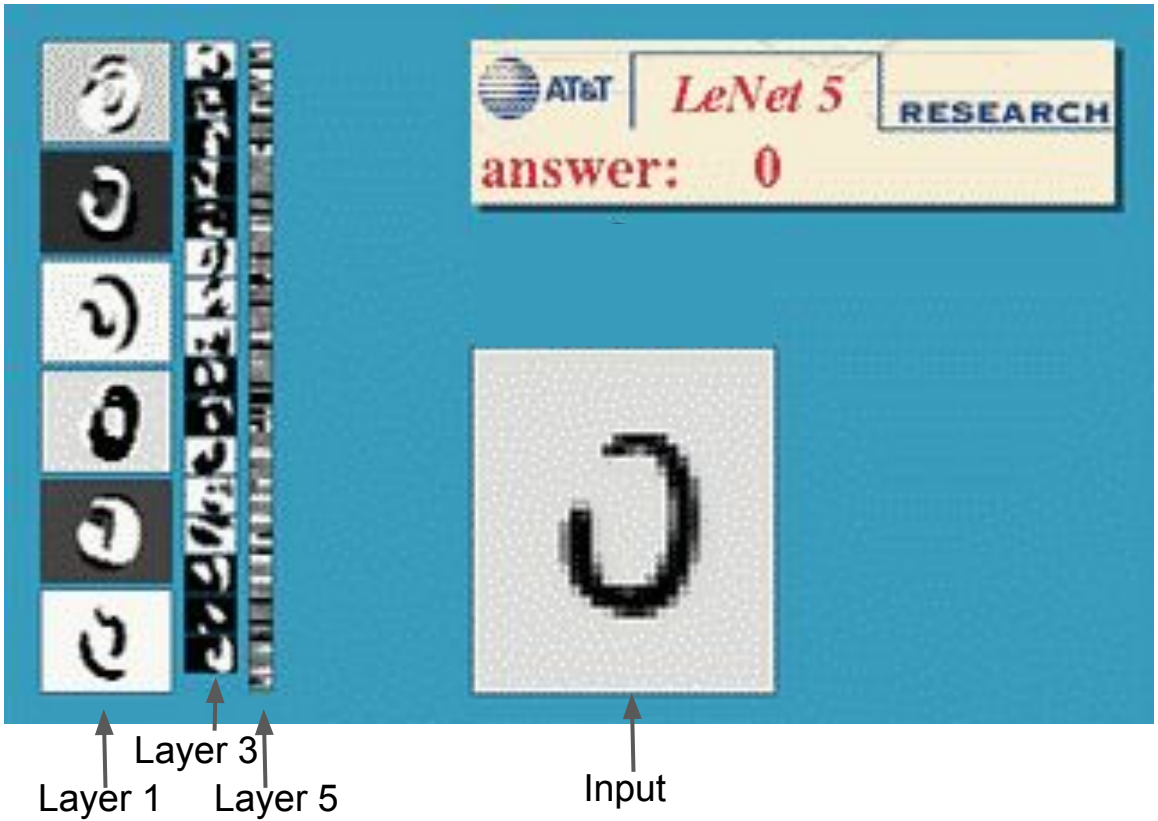
All images are 28x28 grayscale.

60k training examples.

10k test examples

Output value is integer from 0-9

CNN for document recognition [LeCun *et al.*, 1989].



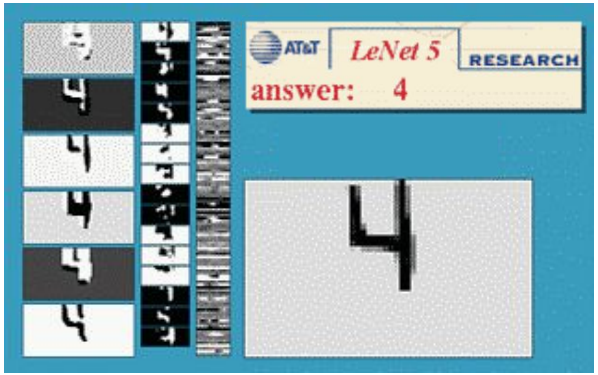
All images are 28x28 grayscale.

60k training examples.

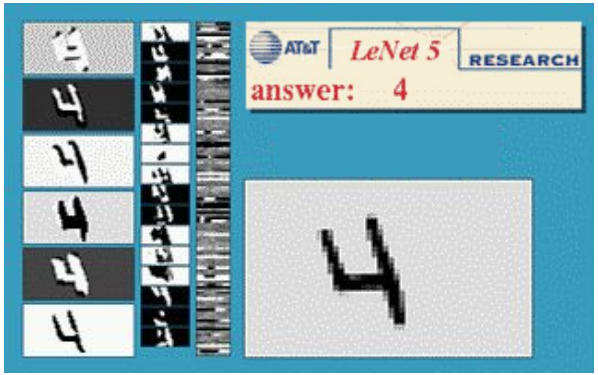
10k test examples

Output value is integer from 0-9

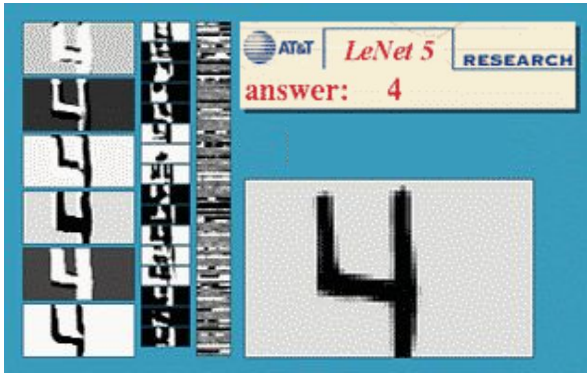
CNN for document recognition [LeCun *et al.*, 1989].



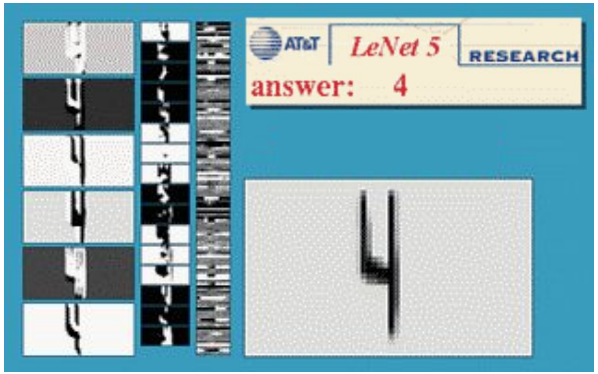
Translation invariance



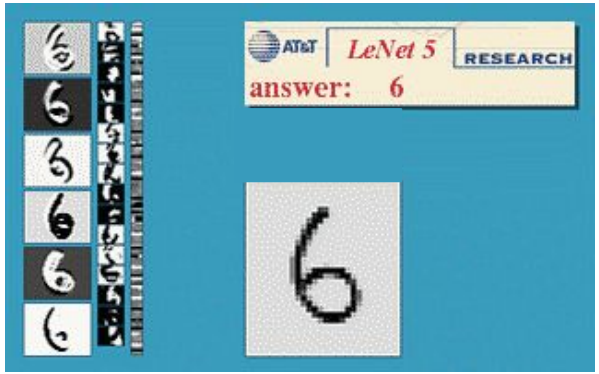
Rotation invariance



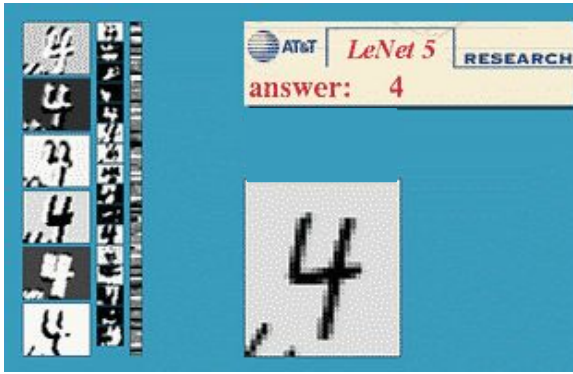
Scale invariance



Squeeze invariance



Stroke-width invariance



Noise invariance

Then why DL didn’t take-off in 90’s?

1. ***Limited*** *big data availability*
2. ***Limited*** *computational power to crunch data*

Why DL is trending now?

Big data availability

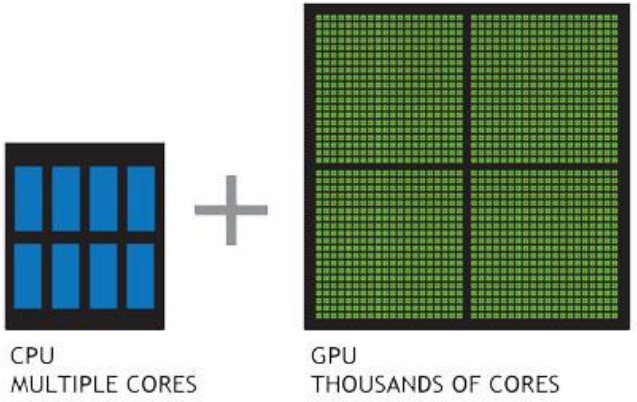
One trillion images.

350 million images uploaded **per day**.

100 hrs of video uploaded **per minute**.

2.5 Petabytes data **every minute**.

Computational power to crunch data



Parallel processing units - GPUs

When/how was deep-learning reclaimed?

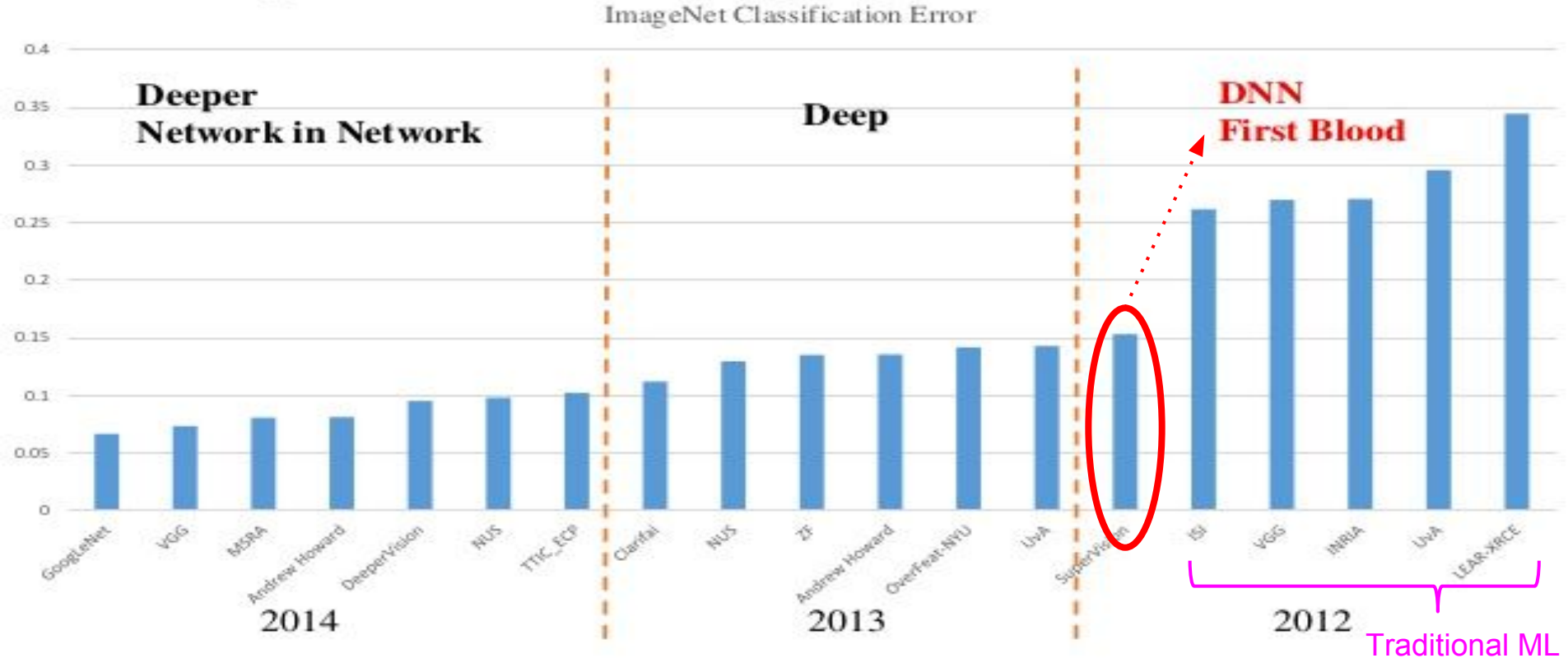
IMAGENET

- 1,000 object classes (categories).
- Images:
  - 1.2 M train
  - 100k test.

When/how was deep-learning reclaimed?

ImageNet Classification

- **1000** categories and **1.2** million training images



Li Fei-Fei: ImageNet Large Scale Visual Recognition Challenge, 2014 http://image-net.org/

Topics

General and biological motivation.

Hand-coded to learnt features.

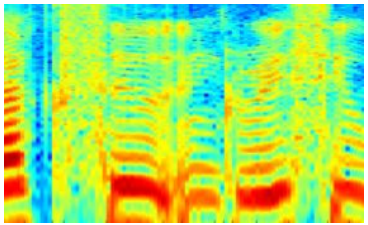
CNNs over fully connected networks.

Different layers in architecture (pooling, relu, etc.)

Traditional machine learning

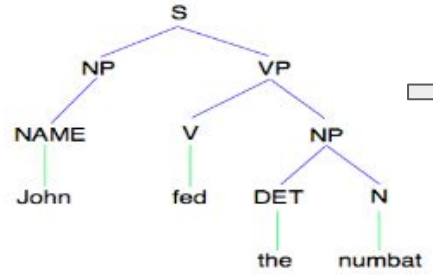
Raw data | Feature extraction | Classifier/detector | Result

SVM or clustering or Shallow neural n/w or HMM, etc.



”

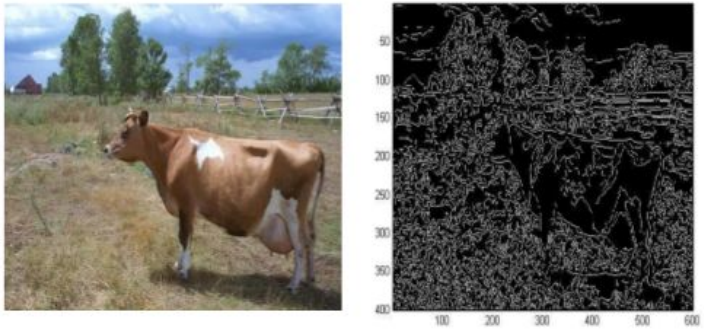
Speaker id, speech translate



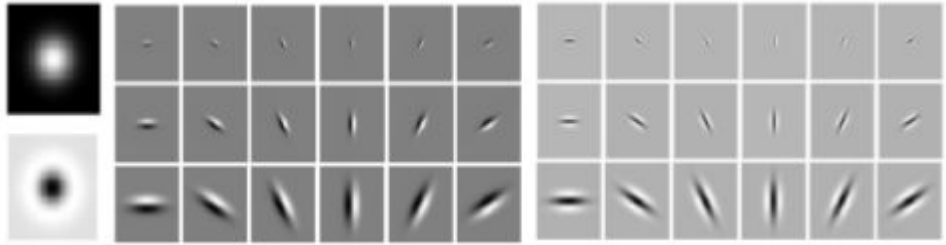
”

Machine translation

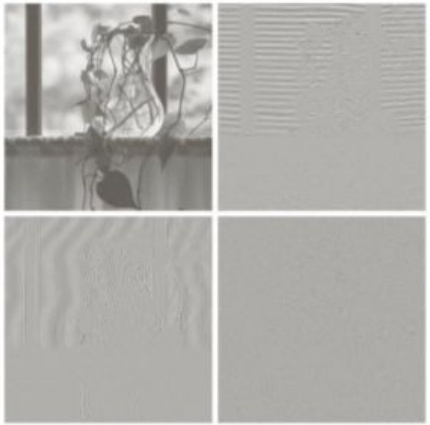
Features: Classical



Edges and Corners: Sobel, LoG and Canny

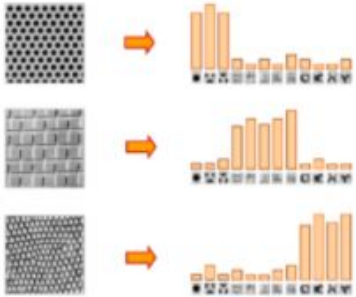


Filter banks



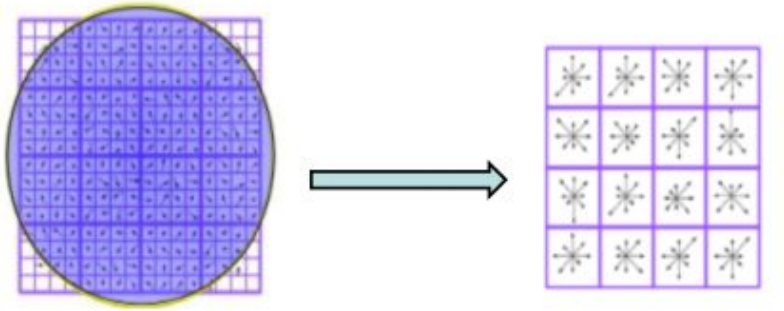
Different transforms (Fourier/Wavelet)

PCA/Subspaces

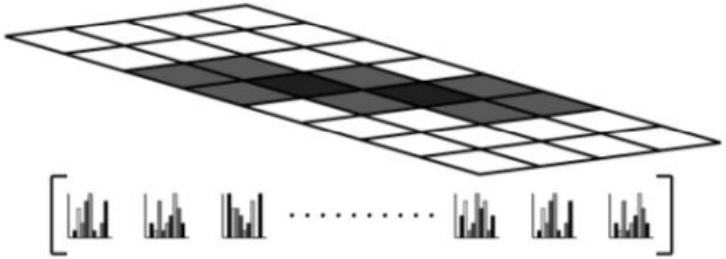


Histogram of responses

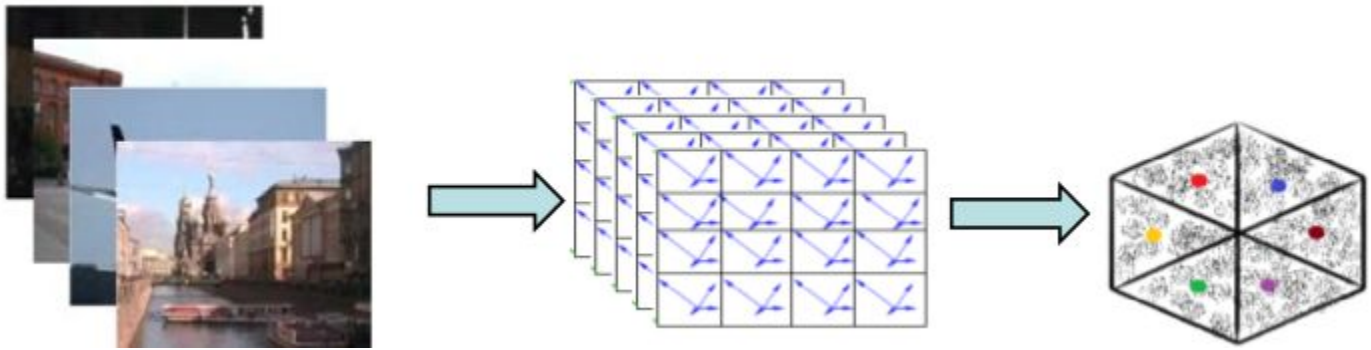
Well Engineered Features



SIFT (Lowe 1999, 2004)



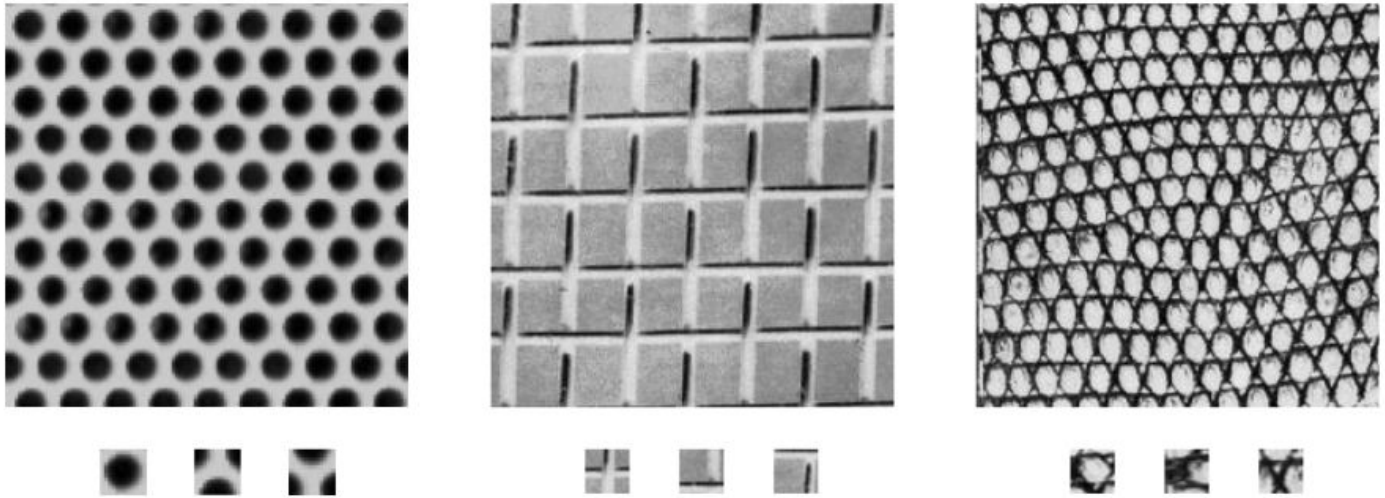
HOG (Dalal and Triggs 2005)



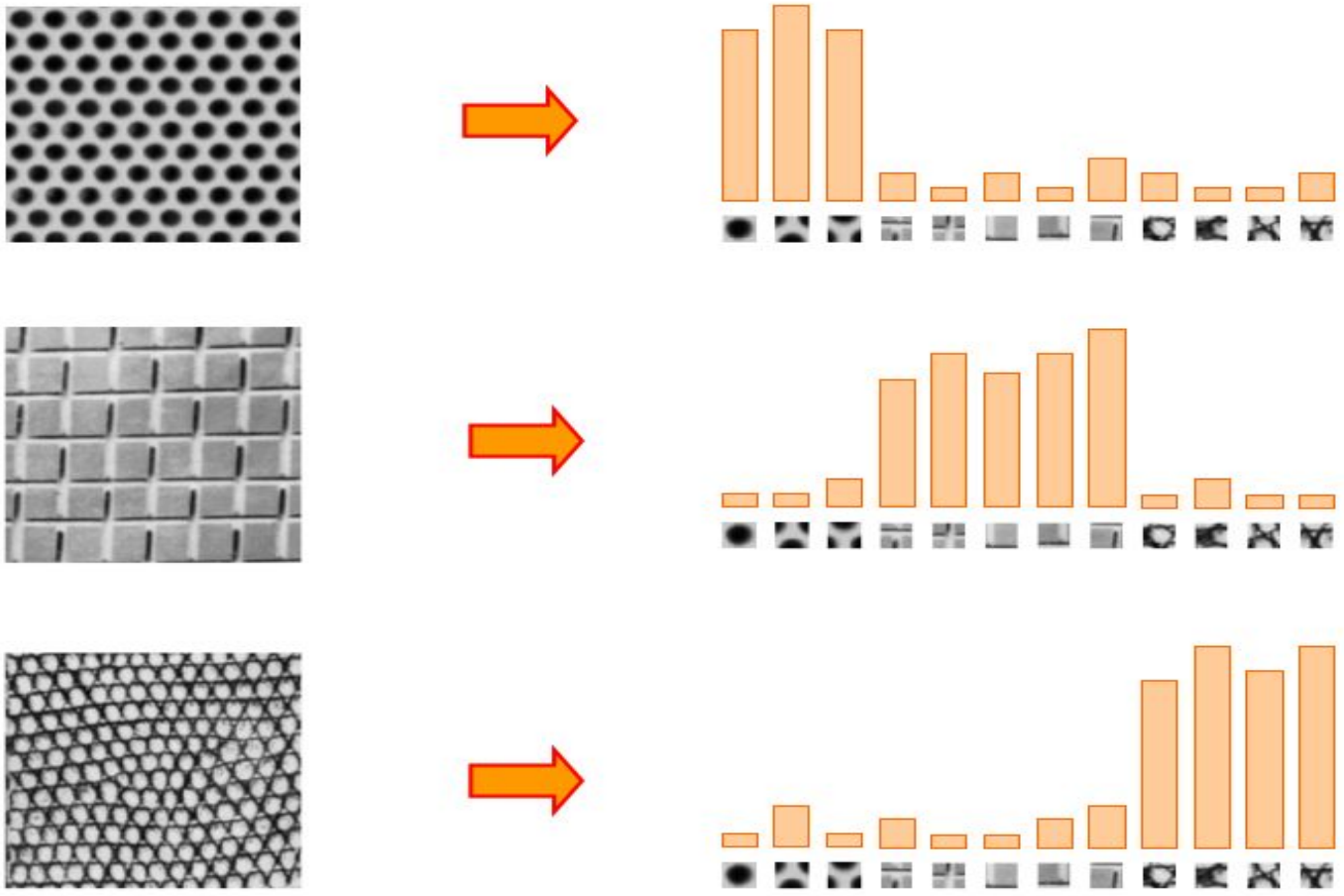
Bag of Words (Sivic and Zisserman 2003)

Example: Texture recognition

1. Texture is characterized by the repetition of basic elements or textons.
2. For stochastic textures, it is the identity of the textons, not their spatial arrangement, that matters.

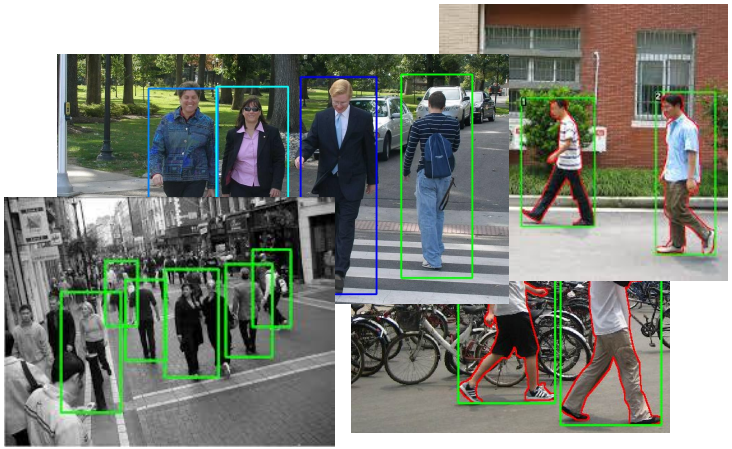


Example: Texture recognition

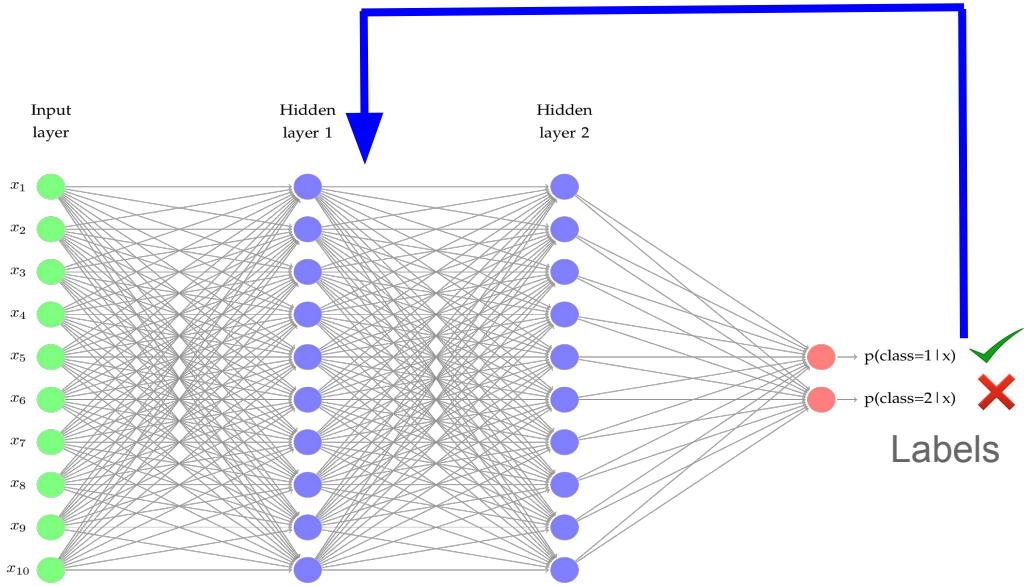


Deep learning

Training phase



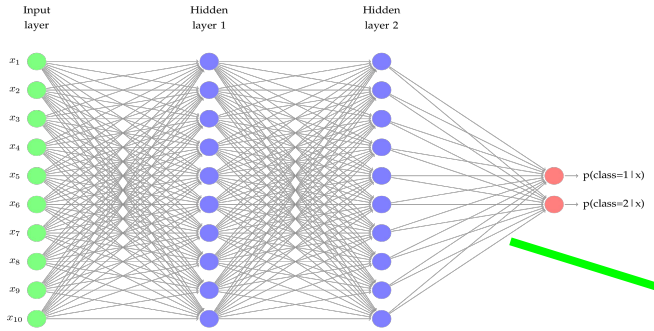
Labelled dataset (usually in millions)



Network (with unoptimized weights)

Deep learning

Deployment



Network (with trained weights)

Pedestrian detection (for automatic braking)

Deep learning benefits over traditional ML

Robust

1. No need to design the features ahead of time – features are automatically learned to be optimal for the task at hand.
2. Robustness to natural variations in the data is automatically learned.

Generalizable

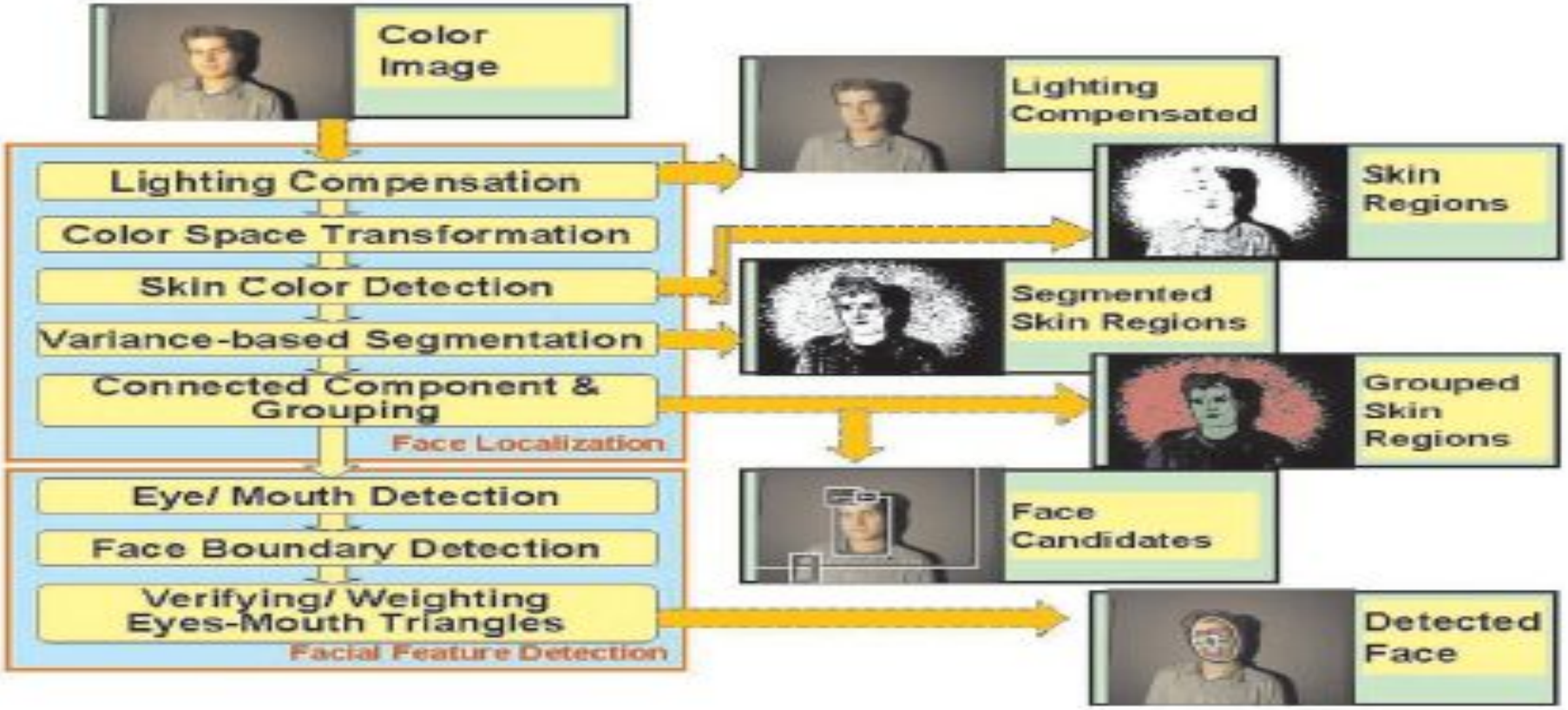
1. The same neural net approach can be used for many different applications and data types (e.g., hand-crafted face features cannot be used for pedestrian detection, whereas the same CNN architecture can be used for both).

Scalable

1. Performance improves with more data, and can be leveraged by massive parallelization of GPUs.

Traditional ML vs Deep learning: Face detection

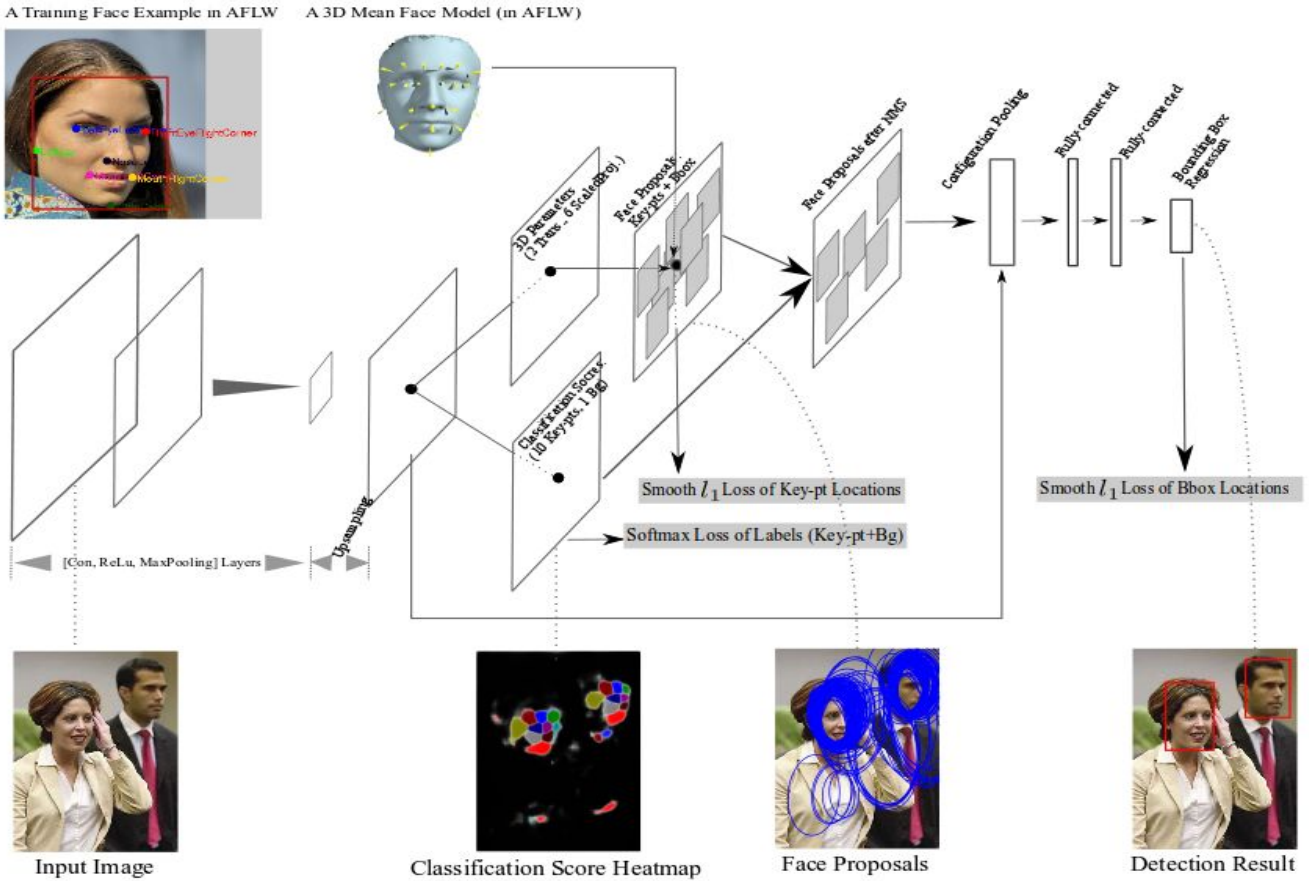
Traditional machine learning



Deep learning: Face detection results

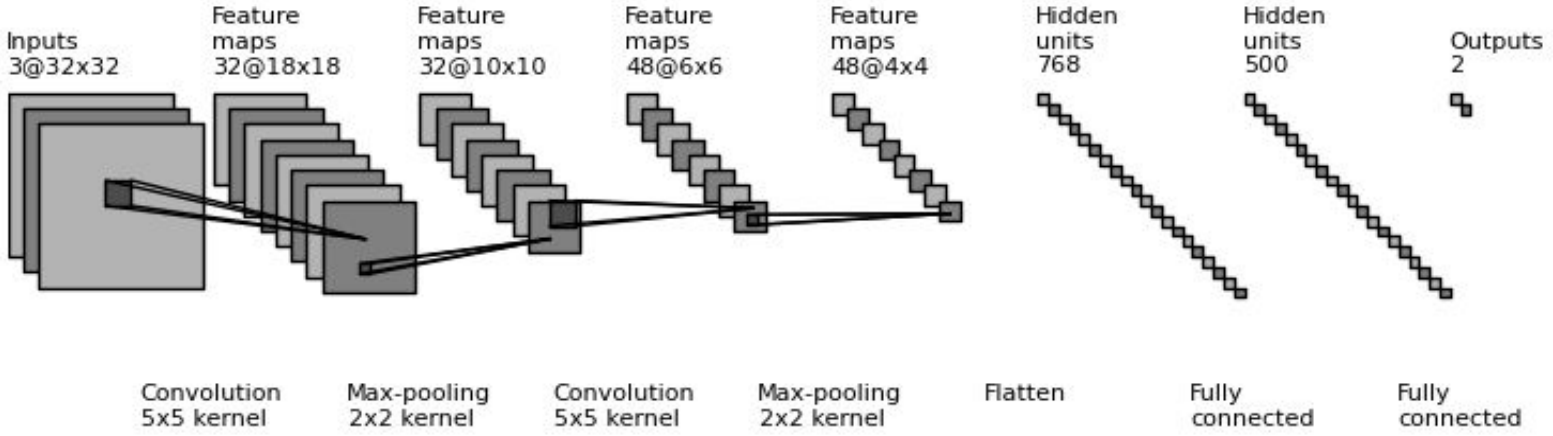
Face Detection with End-to-End Integration of a ConvNet and a 3D Model, Li et al, ECCV 2016

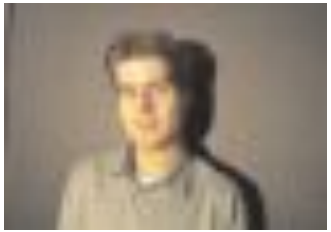
Deep learning: Face detection



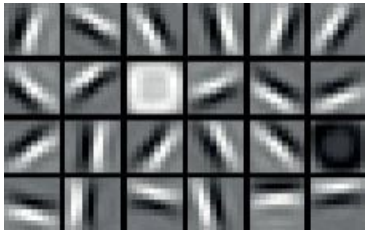
Traditional ML vs Deep learning: Face detection

Deep learning

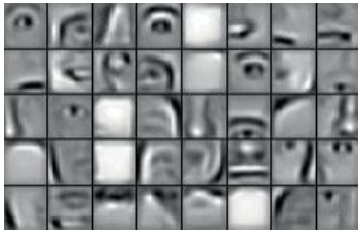




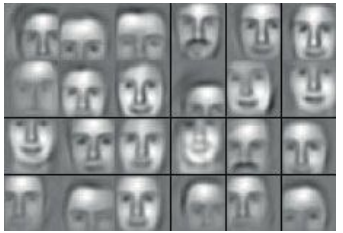
Input



Low-level features



Mid-level features



High-level features

Face $p = 0.94$

Output node

Topics

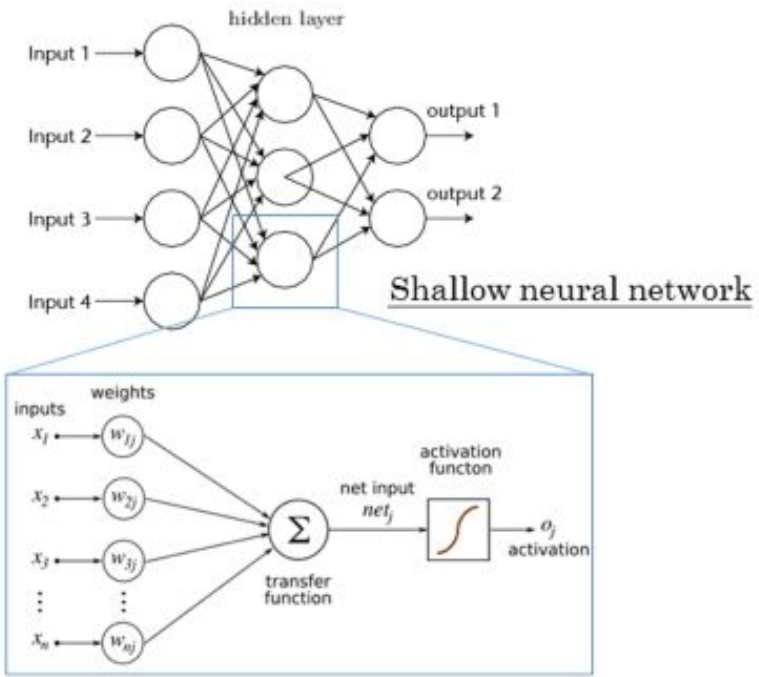
General and biological motivation.

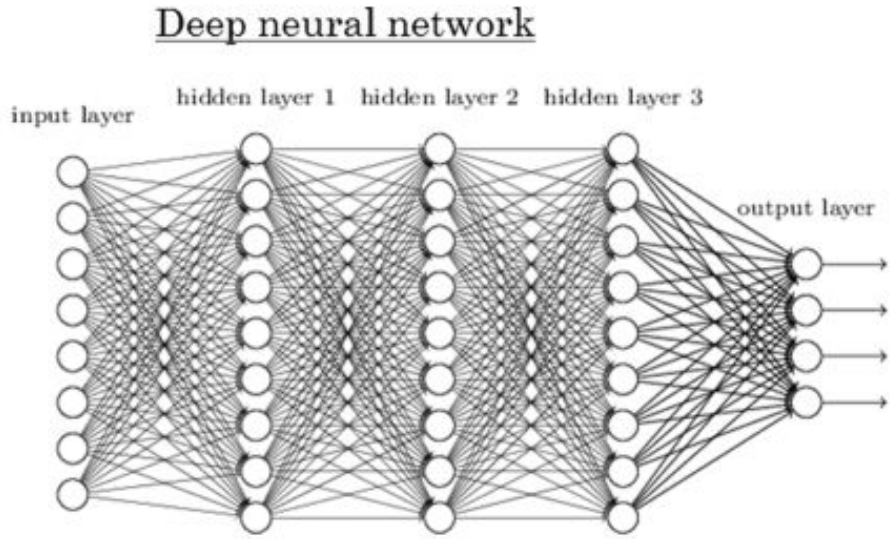
Hand-coded to learnt features.

CNNs over multi-layer neural networks.

Different layers in architecture (pooling, relu, etc.)

CNNs over Multi-layer neural networks (MLNN)



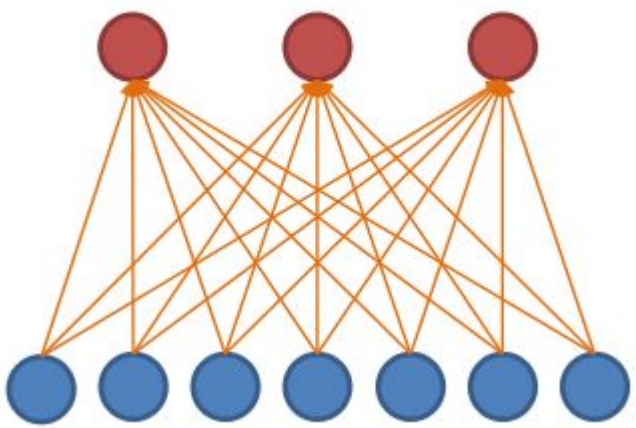


Multi-layer neural network

CNNs are **multi-layer neural network with two constraints**:

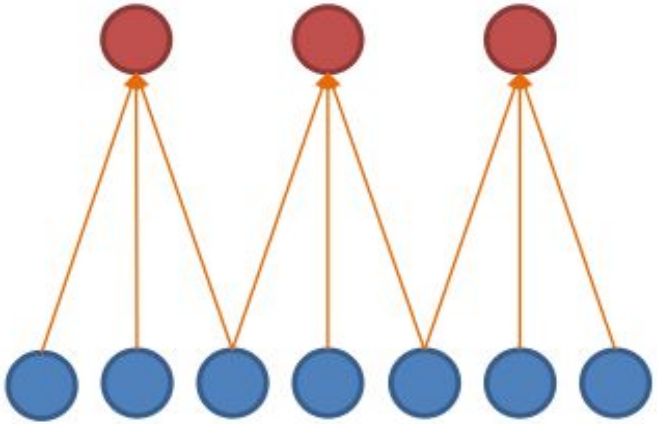
1. Local connectivity
2. Parameter sharing

CNN: Local connectivity (LC)



Hidden layer (3 nodes)

Input layer (7 nodes)



MLNN (7 X 3 = 21 parameters)

MLNN-LC (3 X 3 = 9 parameters)

2.3X runtime and storage efficient.

In general for a level with *m* input and *n* output nodes and CNN-local connectivity of *k* nodes *(k<m)*:

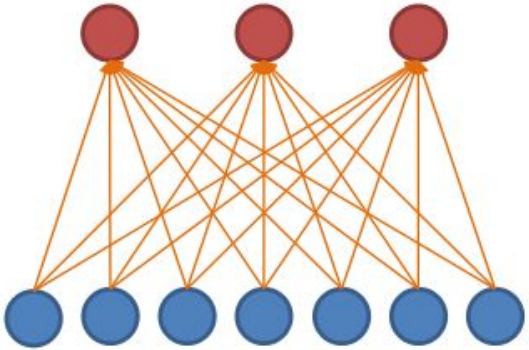
MLNN have

1. *m x n* parameters to store.
2. O(*m x n*) runtime

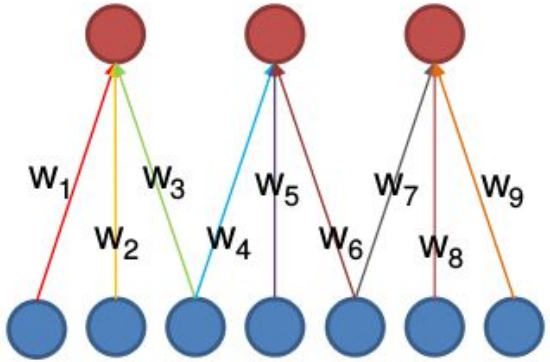
MLNN-LC have:

1. *k x n* parameters to store.
2. O(*k x n*) runtime

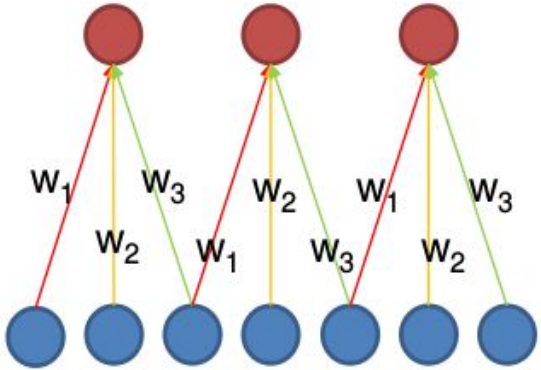
CNN: Parameter sharing (PS)



MLNN (21 parameters)



MLNN-LC (3 X 3 = 9 parameters)
2.3X runtime and storage efficient.



MLNN-LC-PS (3 parameters)
2.3X faster,
& 7X storage efficient.

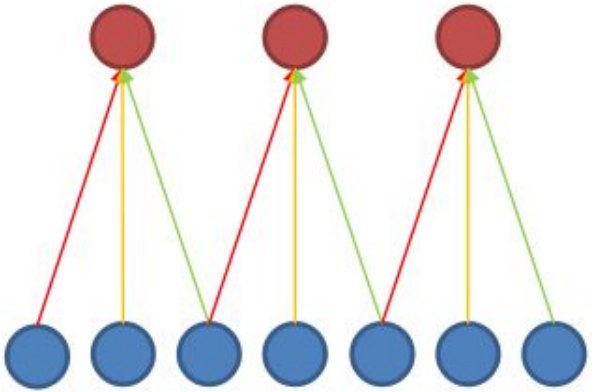
In general for a level with *m* input and *n* output nodes and CNN-local connectivity of *k* nodes *(k<m)*:

MLNN have
1. *m x n* parameters to store.
2. O(*m x n*) runtime

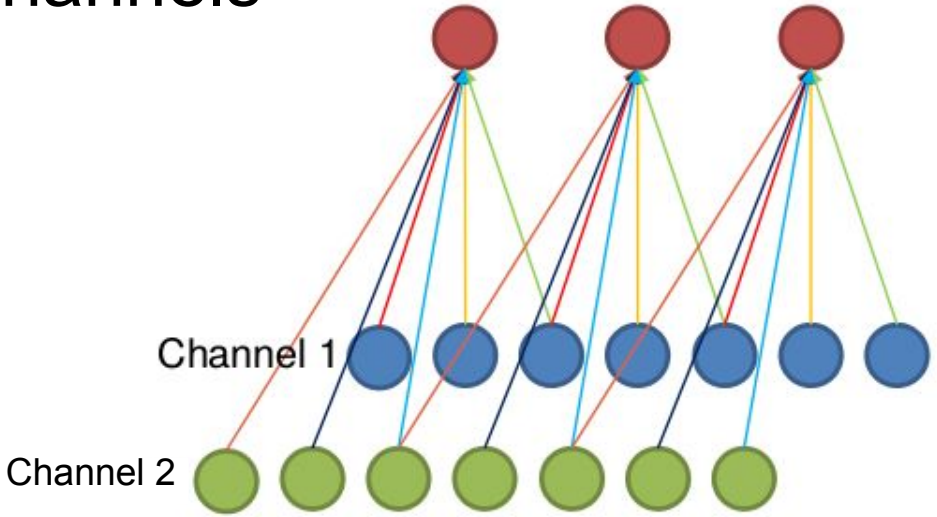
MLNN-LC have:
1. *k x n* parameters to store.
2. O(*k x n*) runtime

MLNN-LC-PS have:
1. *k* parameters to store.
2. O(*k x n*) runtime

CNN with multiple input channels



Single input channel

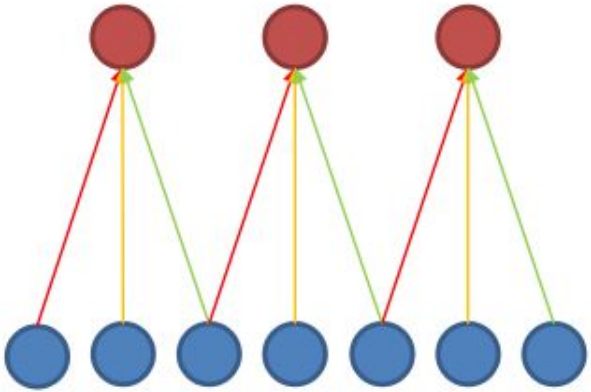


Two input channels

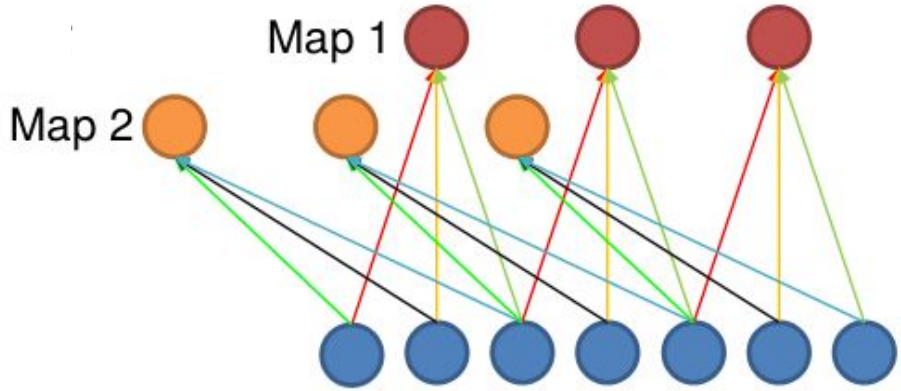
Filter weights

Filter weights

CNN with multiple output maps

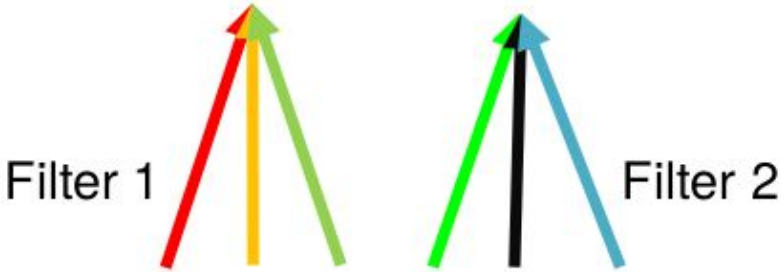


Single input map



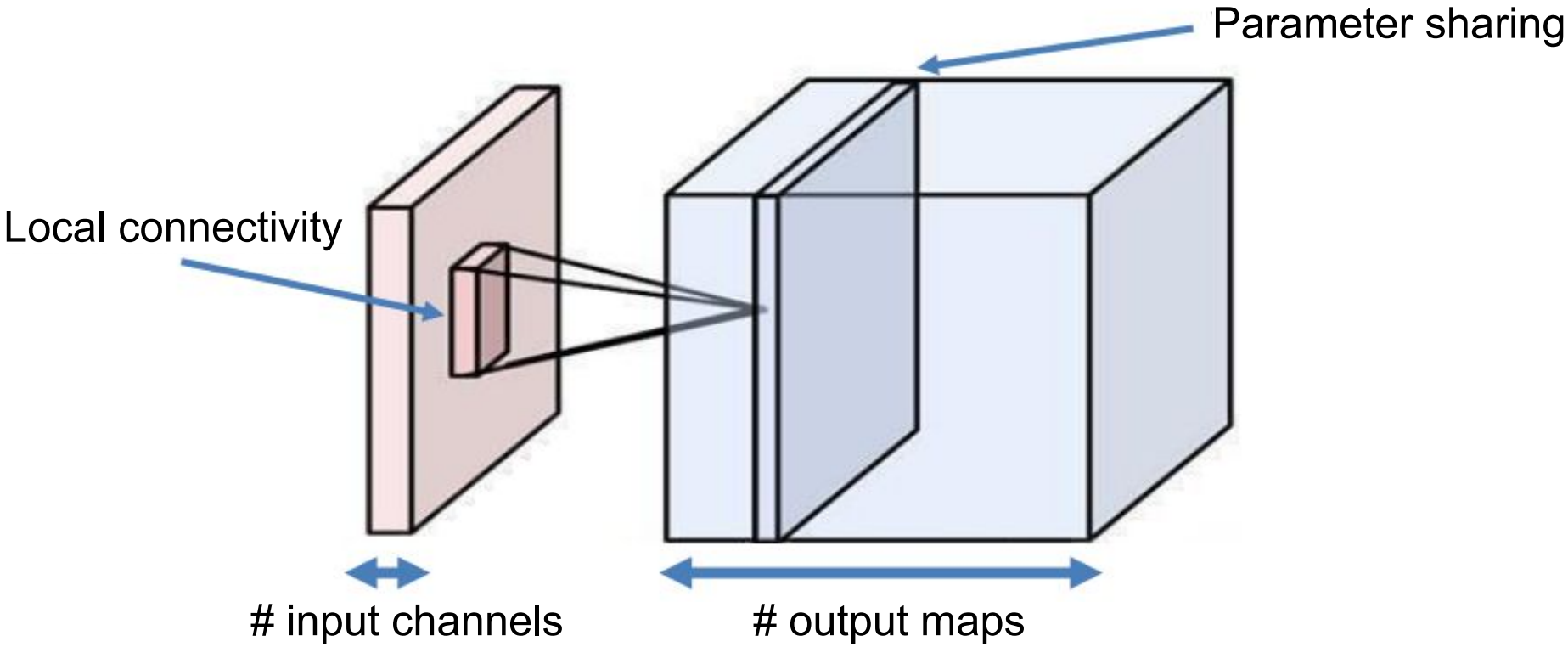
Two output maps

Filter weights



Filter weights

A generic level of CNN



Topics

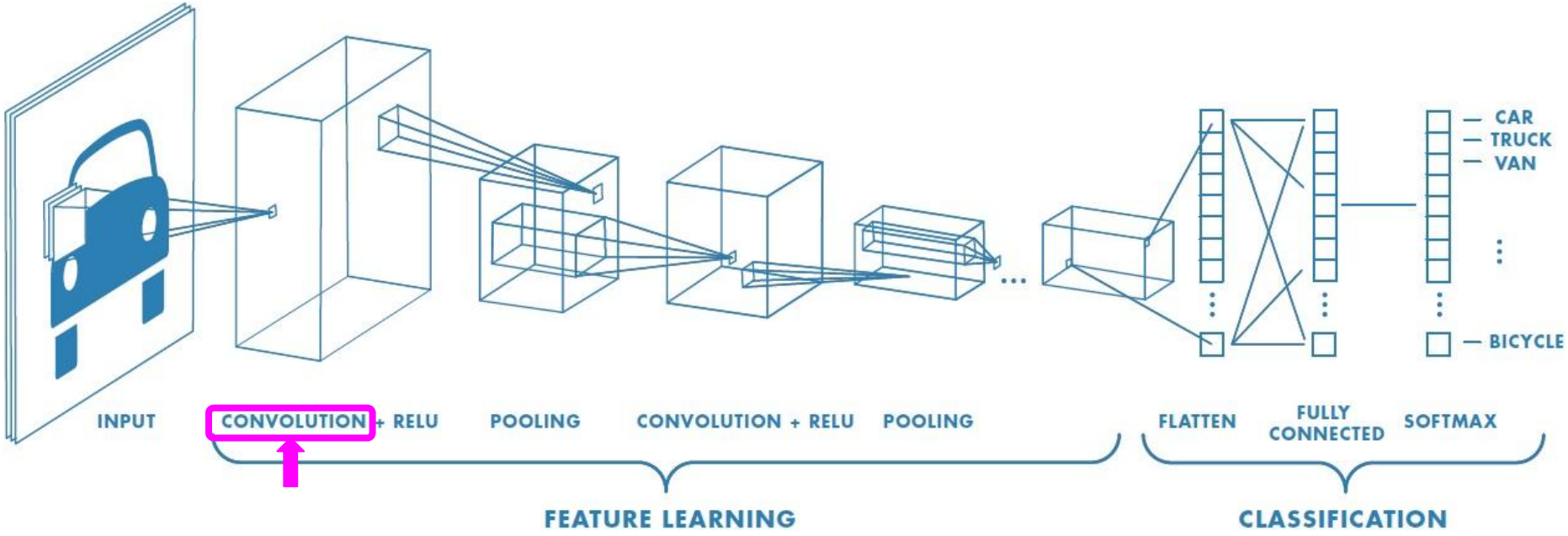
General and biological motivation.

Hand-coded to learnt features.

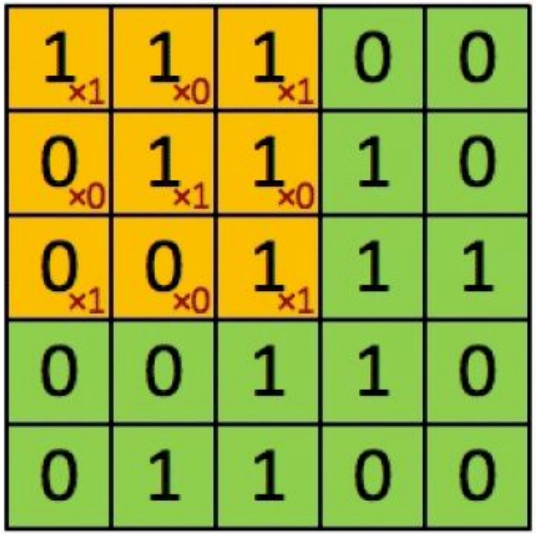
CNNs over multi-layer neural networks.

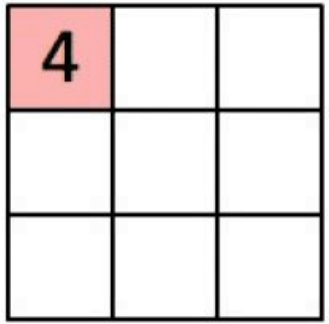
Different layers in CNN architecture (pooling, relu, etc.)

Different layers of CNN architecture



CNN: Convolutional layer

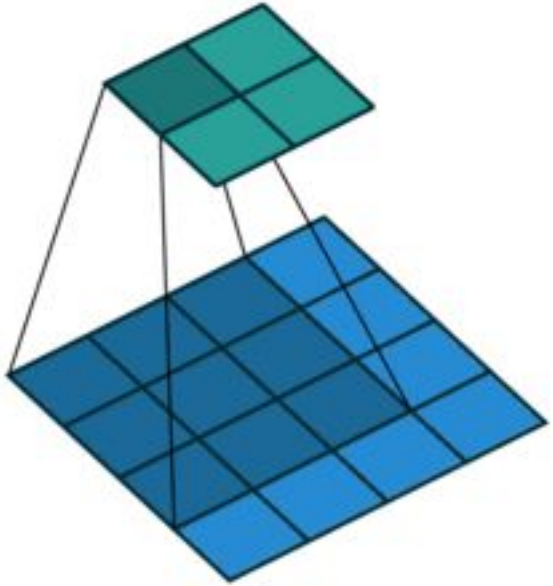




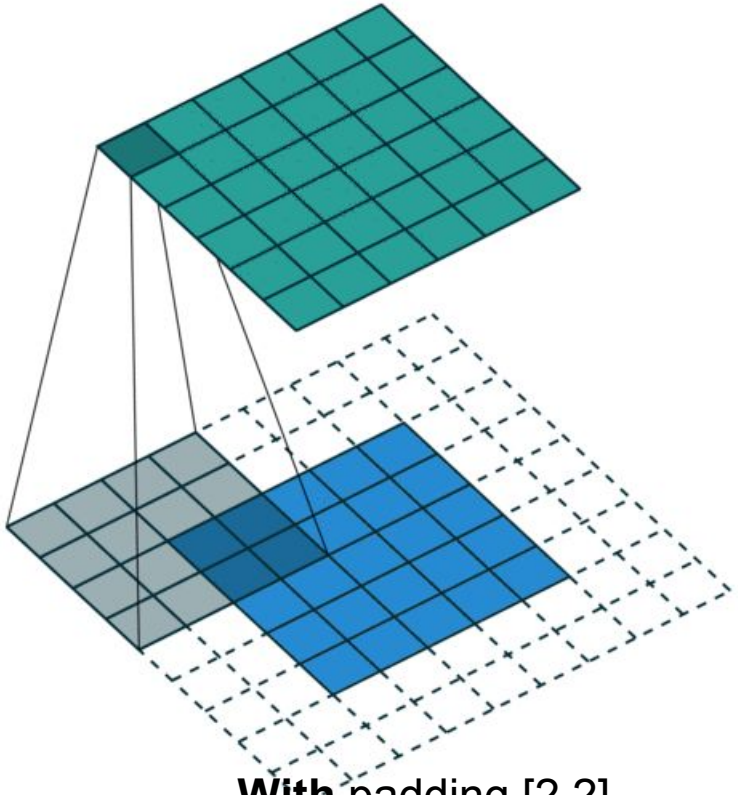
1. To reduce the number of weights (through local connectivity).
2. To provide spatial invariance (through parameter sharing).

Hyper parameters for convolutional layer.

1. Zero padding (to control input size spatially.)



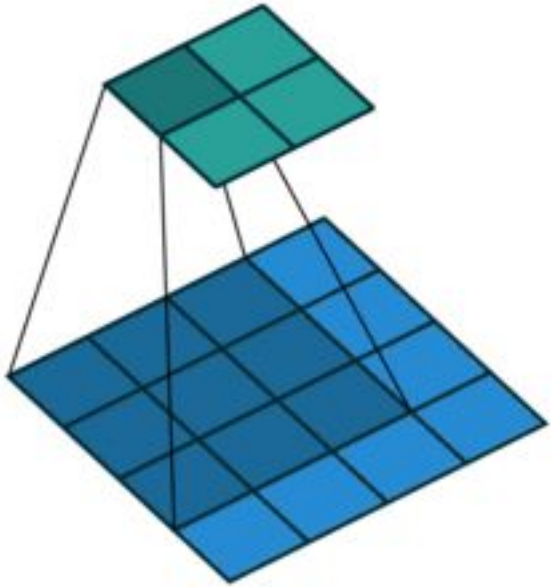
Without padding (i.e., [0,0])



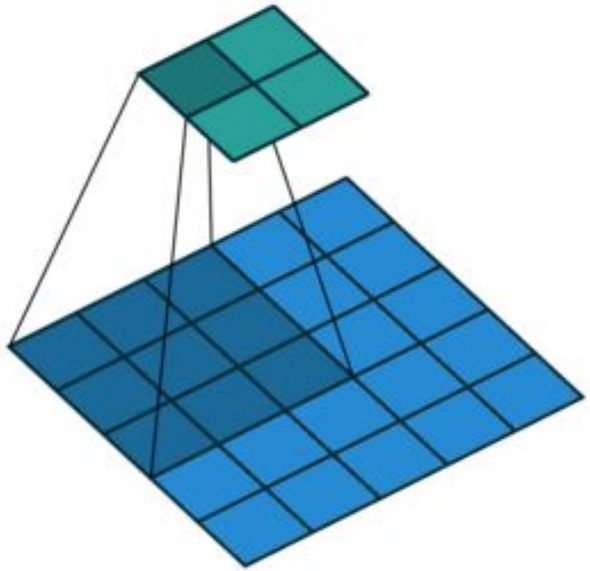
With padding [2,2]

Hyper parameters for convolutional layer.

2. Stride (to produce smaller output volumes spatially.)



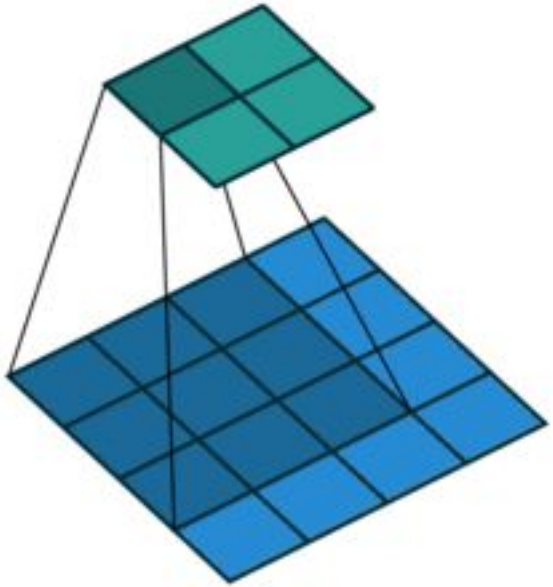
Without stride (i.e., [1,1])



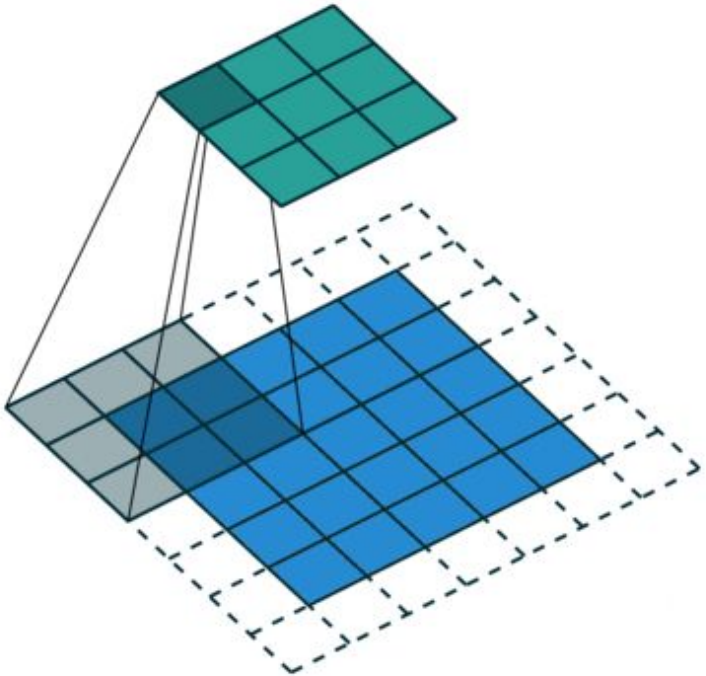
With stride [2,2]

Hyper parameters for convolutional layer.

Both padding and stride



Without padding and stride

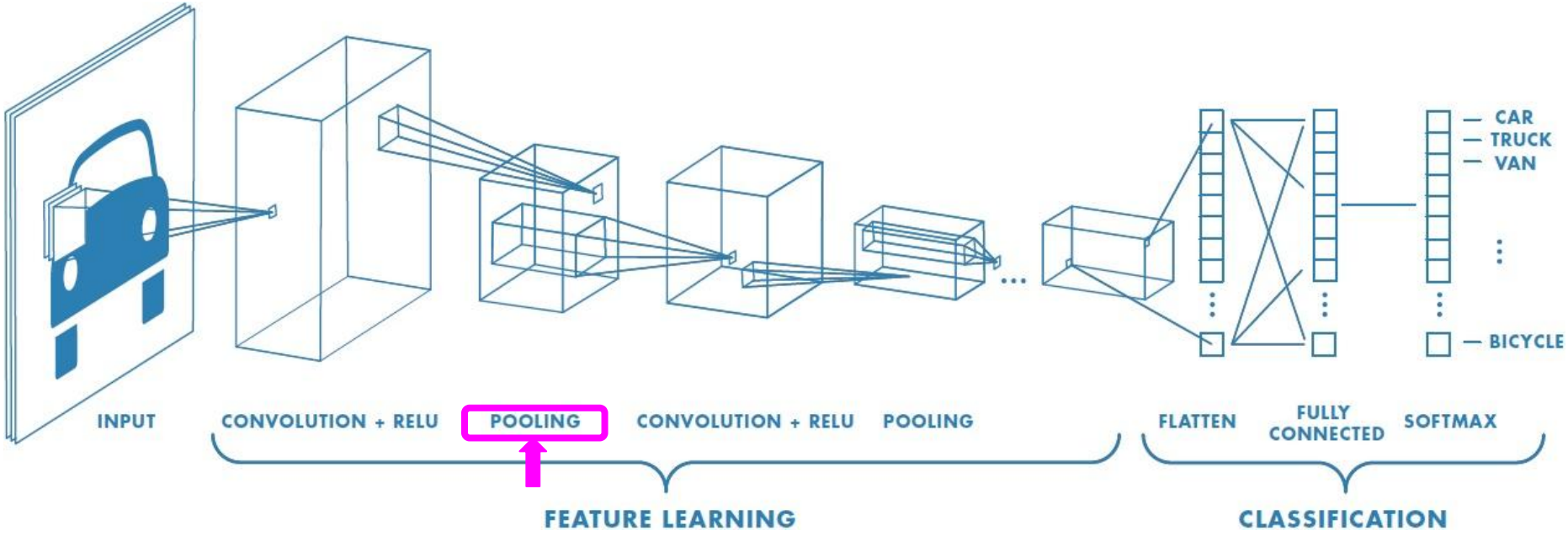


With padding [1,1] & stride [2,2]

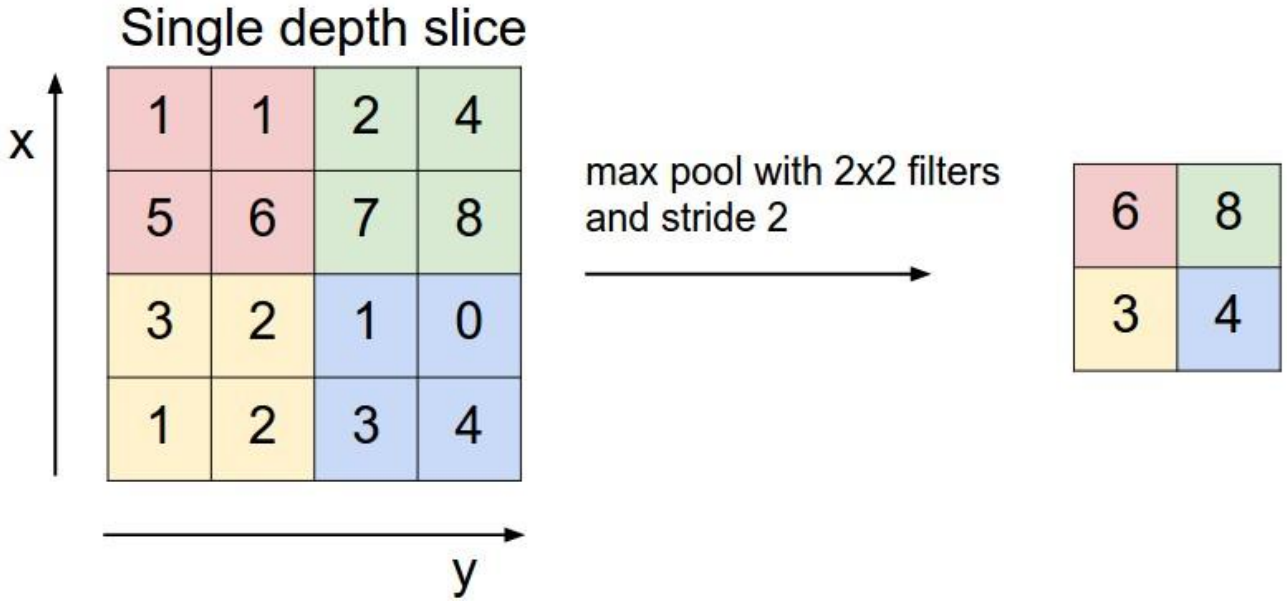
CONVOLUTIONAL LAYER

1. Accepts a volume of size ***W1*** **X** ***H1*** **X** ***D1***.
2. Requires four hyperparameters:
   a. Number of filters ***K***
   b. the spatial extent of filter ***F***
   c. their stride ***S***
   d. the amount of zero padding ***P***
3. Produces an output volume of size ***W2*** **X** ***H2*** **X** ***D2*** where:
   $W2=(W1-F+2P)/S+1,\ H2=(H1-F+2P)/S+1,\ D2=K$
4. With parameter sharing, it introduces $F \cdot F \cdot D1$ weights per filter, for a total of $(F \cdot F \cdot D1) \cdot K$ weights and ***K*** biases.
5. In the output volume, the ***d***-th depth slice (of size ***W2*** **X** ***H2***) is the result of performing a valid convolution of the ***d***-th filter over the input volume with a stride of ***S***, and then offset by ***d***-th bias.

Different layers of CNN architecture



CNN: Pooling layer

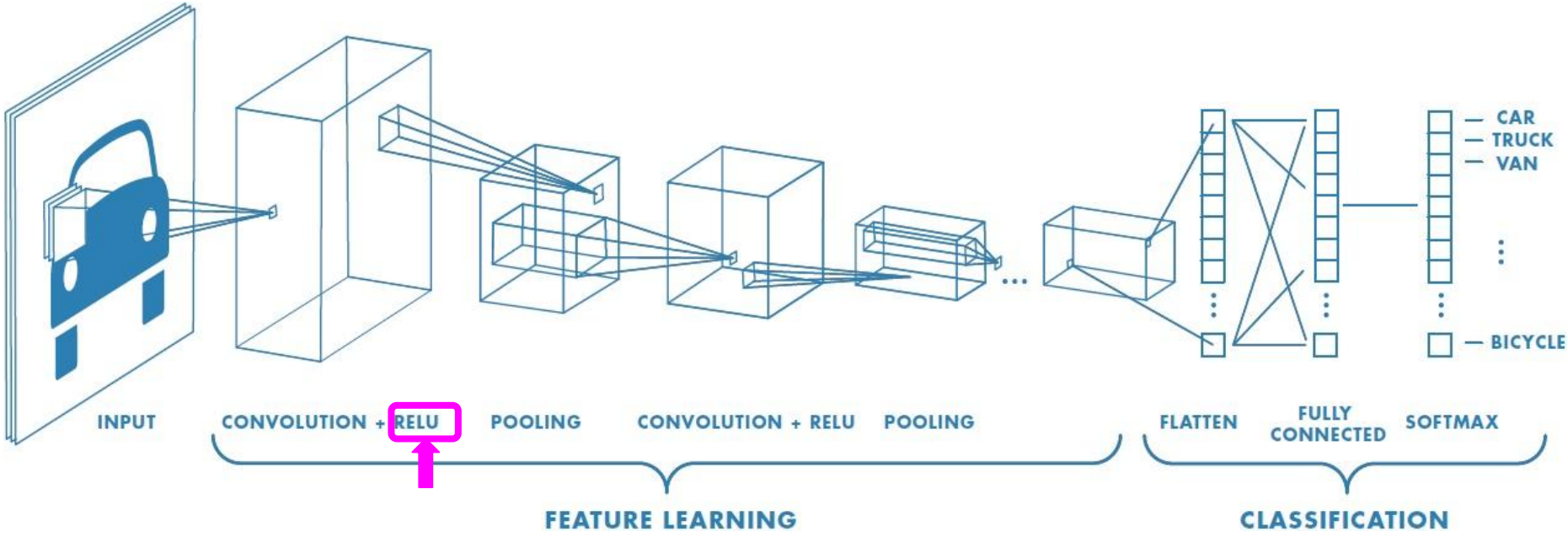


1. To reduce the spatial size of the representation to reduce the amount of parameters and computation in the network.
2. Average pooling or L2 pooling can also be used, but not popular like max pooling.

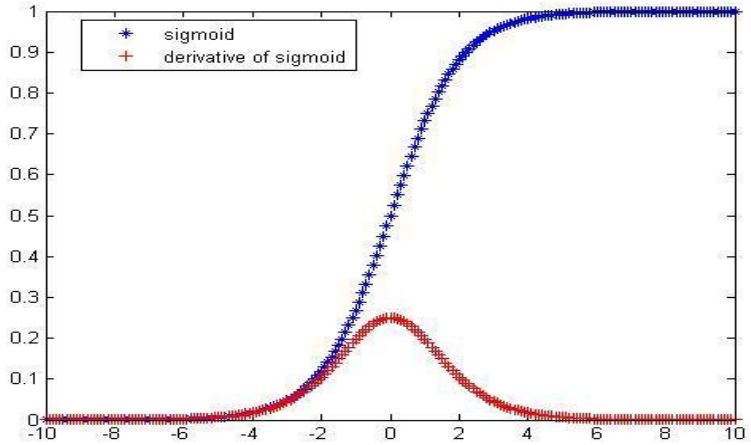
POOLING LAYER

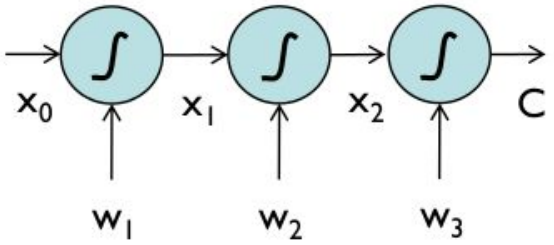
1. Accepts a volume of size ***W1*** **X** ***H1*** **X** ***D1***.
2. Requires two hyperparameters:
   a. the spatial extent of filter ***F***
   b. their stride ***S***
   c. the amount of zero padding ***P*** (commonly ***P*** = 0).
3. Produces an output volume of size ***W2*** **X** ***H2*** **X** ***D2*** where:
   W2=(W1−F+2P)/S+1, H2=(H1−F+2P)/S+1, D2=K
4. Introduces **zero** parameters since it computes a fixed function of the input.

Different layers of CNN architecture



Activation functions: Sigmoidal function





$$x_i = \sigma'(w_i{}^T x_{i-1})$$

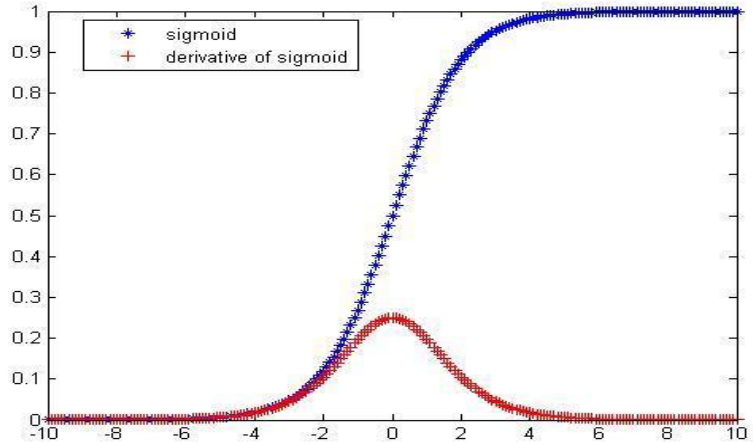
$$\frac{\partial C}{\partial w_1(j)} = \sigma'(w_3{}^T x_2) \cdot x_2(j) \cdot \sigma'(w_2{}^T x_1) \cdot x_1(j) \cdot \sigma'(w_1{}^T x_0) \cdot x_0(j)$$

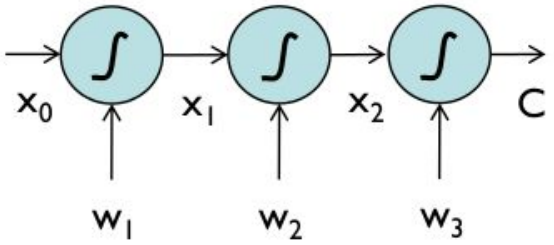
Drawback 1: *Sigmoids saturate and kill gradients* (when the neuron's activation saturates at either tail of 0 or 1).

0 ⇒ fails to update weights while back-prop.

$$\frac{\partial C}{\partial w_1(i)} = \sigma'(w_3{}^T x_2) \cdot x_2(i) \cdot \sigma'(w_2{}^T x_1) \cdot x_1(i) \cdot \sigma'(w_1{}^T x_0) \cdot x_0(i)$$

Activation functions: Sigmoidal function





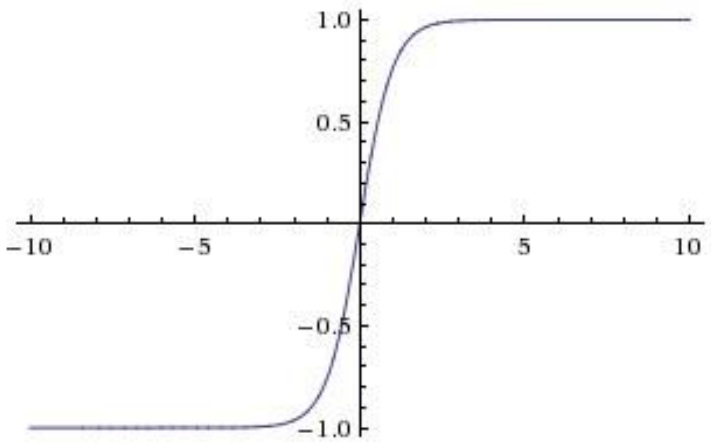
$$x_i = \sigma\left(w^T x_{i-1}\right)$$

$$\frac{\partial C}{\partial w_1(j)} = \sigma'({w_3}^T x_2) \cdot x_2(j) \cdot \sigma'({w_2}^T x_1) \cdot x_1(j) \cdot \sigma'({w_1}^T x_0) \cdot x_0(j)$$

Drawback 2: *Undesirable zig-zagging dynamics in the gradient updates for the weights* (because data coming into a neuron is always positive, then the gradient on the weights *w* will during backpropagation become either all be positive, or all negative.

$$\frac{\partial C}{\partial w_1(j)} = \sigma'({w_3}^T x_2) \cdot x_2(j) \cdot \sigma'({w_2}^T x_1) \cdot x_1(j) \cdot \sigma'({w_1}^T x_0) \cdot x_0(j)$$

Activation functions: tanh function



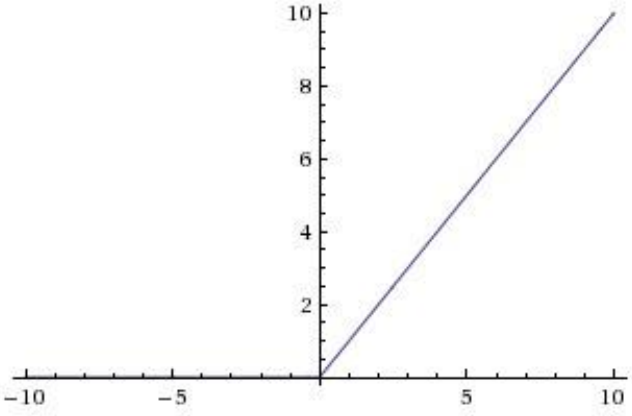
$$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

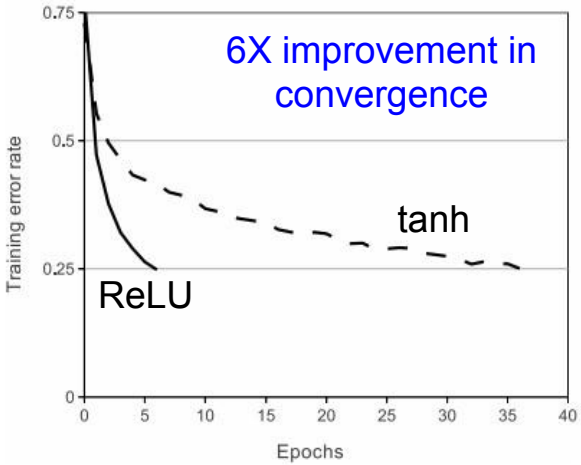
Advantage: **Eliminate** *Undesirable zig-zagging dynamics in the gradient updates for the weights* (because data coming into a neuron can be positive and negative.

$$\frac{\partial C}{\partial w_1(j)} = \sigma'({w_3}^T x_2) \cdot x_2(j) \cdot \sigma'({w_2}^T x_1) \cdot x_1(j) \cdot \sigma'({w_1}^T x_0) \cdot x_0(j)$$

Drawback 1: *But still saturate and kill gradients.*

Activation functions: Rectified Linear Unit (very popular).



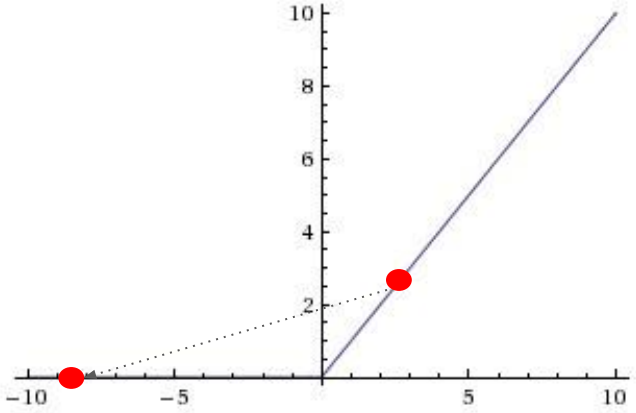


Advantage 1: **Eliminate** *saturation and killing of gradients for positive inputs*.

Advantage 2: Greatly **accelerate** convergence of SGD. (*Krizhevsky et al.* argued that this is due to its linear, non-saturating form.)

Advantage 3: tanh/sigmoid neurons involve expensive operations (exponentials, etc.), whereas ReLU can be implemented by **simply thresholding** activations at zero.

Activation functions: Rectified Linear Unit (very popular).



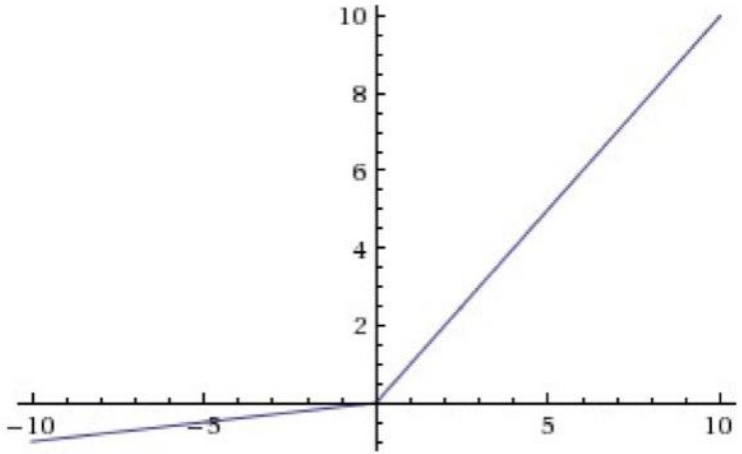
Drawback: ReLU units can **irreversibly die** during training. A large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again.

0 ⇒ fails to update weights while back-prop.

$$\frac{\partial C}{\partial w_1(i)} = \sigma'({w_3}^T x_2) \cdot x_2(i) \cdot \sigma'({w_2}^T x_1) \cdot x_1(i) \cdot \sigma'({w_1}^T x_0) \cdot x_0(i)$$

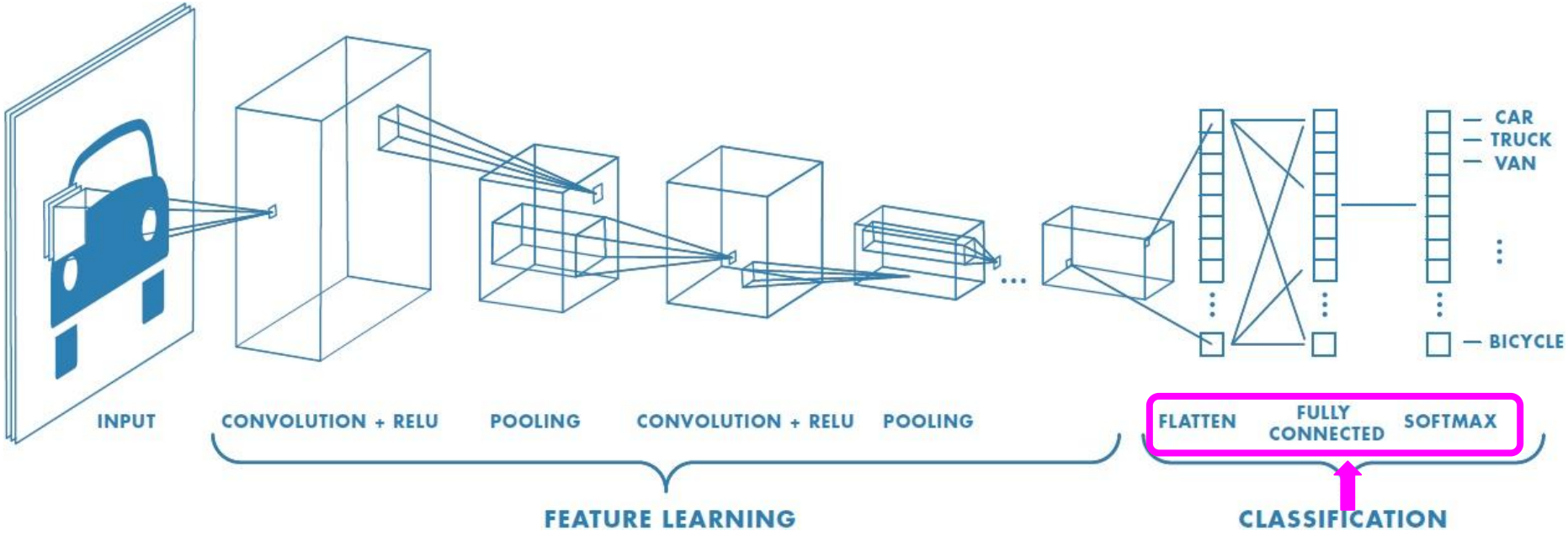
Activation functions: Leaky ReLU.

Instead of the function being zero when x < 0, a leaky ReLU will instead have a small negative slope (a hyper-parameter).

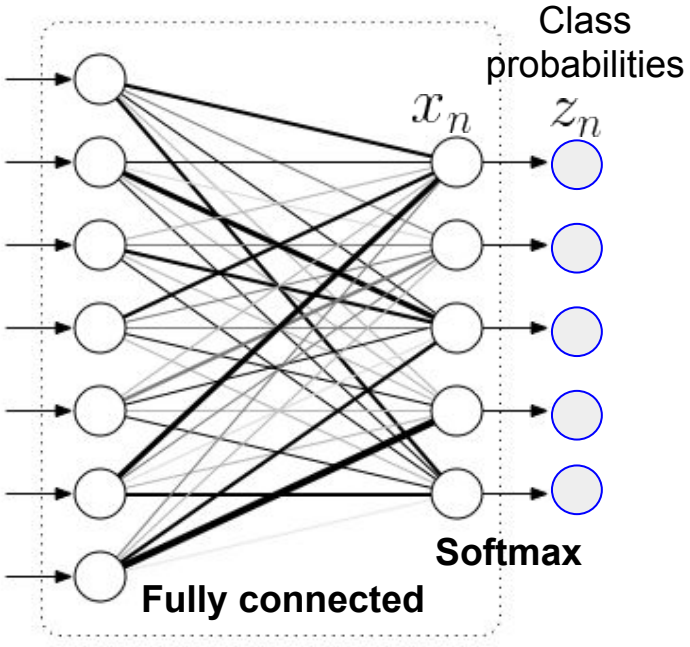


Advantages: **Eliminate** irreversible dying of neurons, as in ReLU.
Have all the advantages of ReLU.

Different layers of CNN architecture



Flattening, fully connected (FC) layer and softmax



Flattening

1. Vectorization (converting ***M X N X D*** tensor to a ***MND X 1*** vector).

FC layer

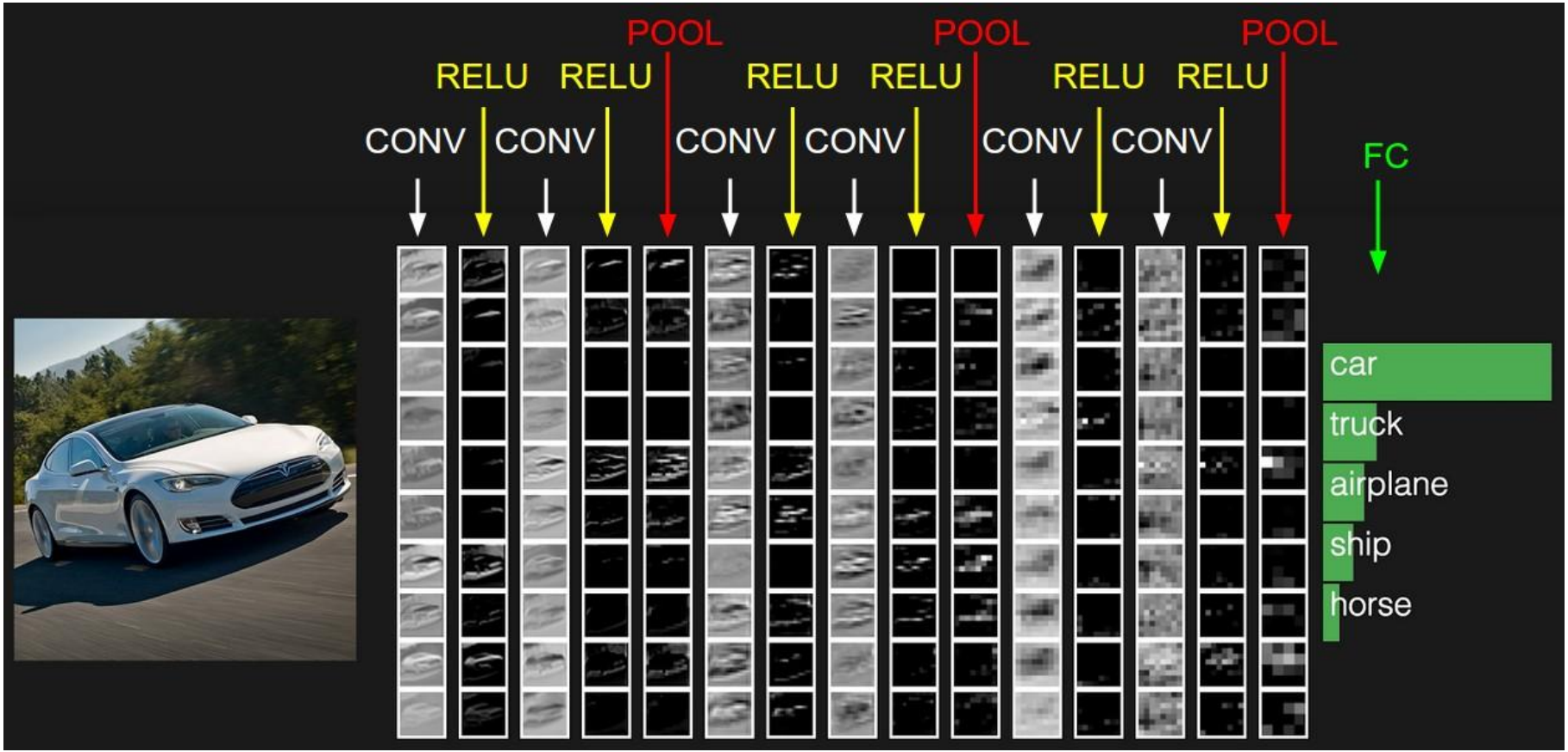
1. Multilayer perceptron.
2. Generally used in final layers to classify the object.
3. Role of a classifier.

Softmax layer

1. Normalize output as discrete class probabilities.

$$z_n = \frac{e^{x_n}}{\sum_{i=1}^{K} e^{x_i}}$$

A CNN Example



Topics

Backpropagation in Convolutional Neural Networks

Batch Normalization

Dropout

Finetuning/ Transfer Learning

Backpropagation

Convolutional Neural Networks

Slides based on Jefkine's article at http://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/

Review of Conv Nets

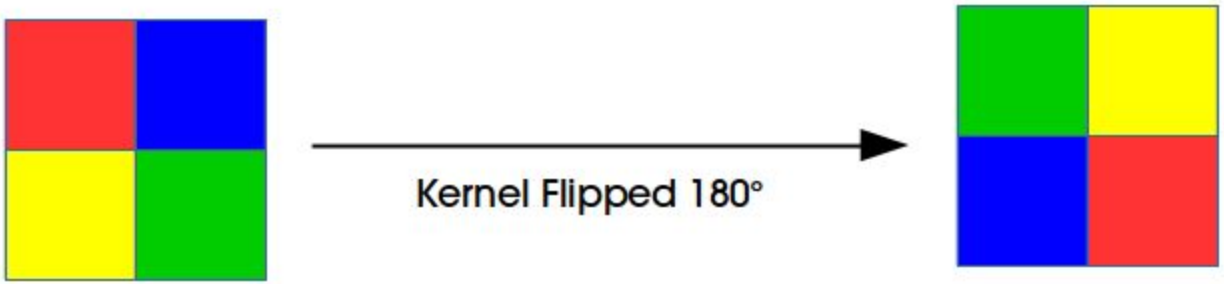
- Biologically inspired variation of MLPs
- Steps involved
  - Perform convolutions
  - Apply non-linearity
  - Pooling (ReLU - most commonly used (why?))
- Acts as a feature extractor

Notations

- l is the l^{th} layer where l = 1,2,...,L
- $w_{i,j}^{l}$ is the weights connecting layer to layer l+1
- b_{l} is the bias at layer l
- $x_{i,j}^{l}$ is defined as $x_{i,j}^{l} = \sum_{i'} \sum_{j'} w_{i',j'}^{l} o_{i-i',j-j'}^{l-1} + b^{l}$
- where $o_{i,j}^{l}$ is the output vector at layer l after the non-linearity $o_{i,j}^{l} = f(x_{i,j}^{l})$
- f(.) is the non-linearity

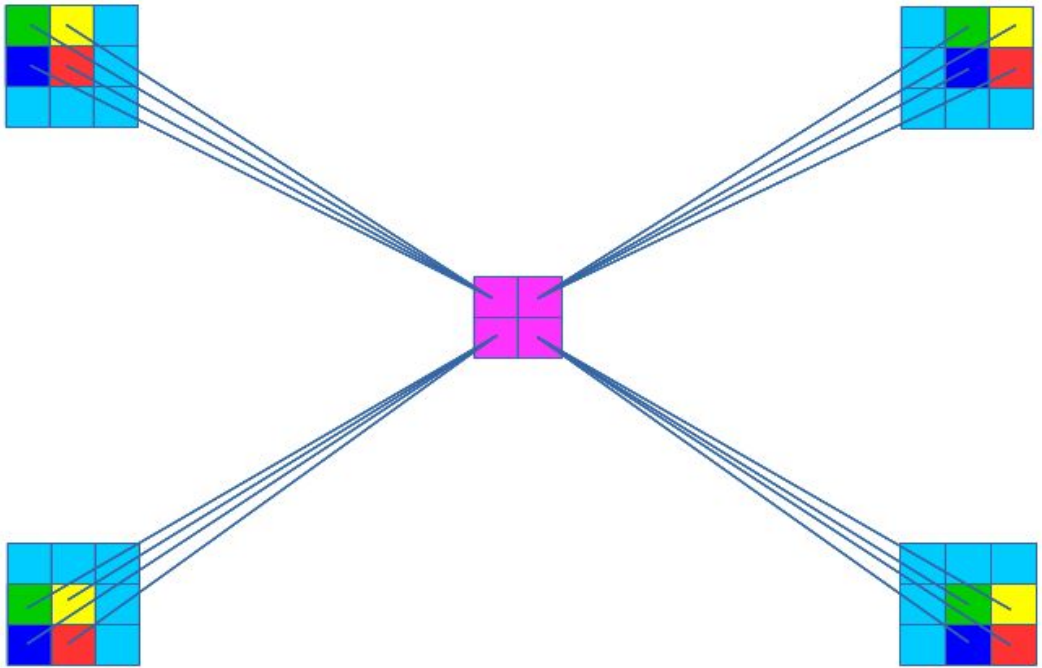
Forward Propagation

- Flip the kernel



Forward Propagation

- Convolution



Forward Propagation

- Mathematical representation

$$x_{i,j}^{l} = \mathrm{rot}_{180^\circ}\left\{\sum_{i'}\sum_{j'} w_{i',j'}^{l} o_{i+i',j+j'}^{l-1} + b_{i,j}^{l}\right\}$$

$$x_{i,j}^{l} = \mathrm{rot}_{180^\circ}\left\{w_{i,j}^{l}\right\} * o_{i,j}^{l-1} + b_{i,j}^{l}$$

$$o_{i,j}^{l} = f(x_{i,j}^{l})$$

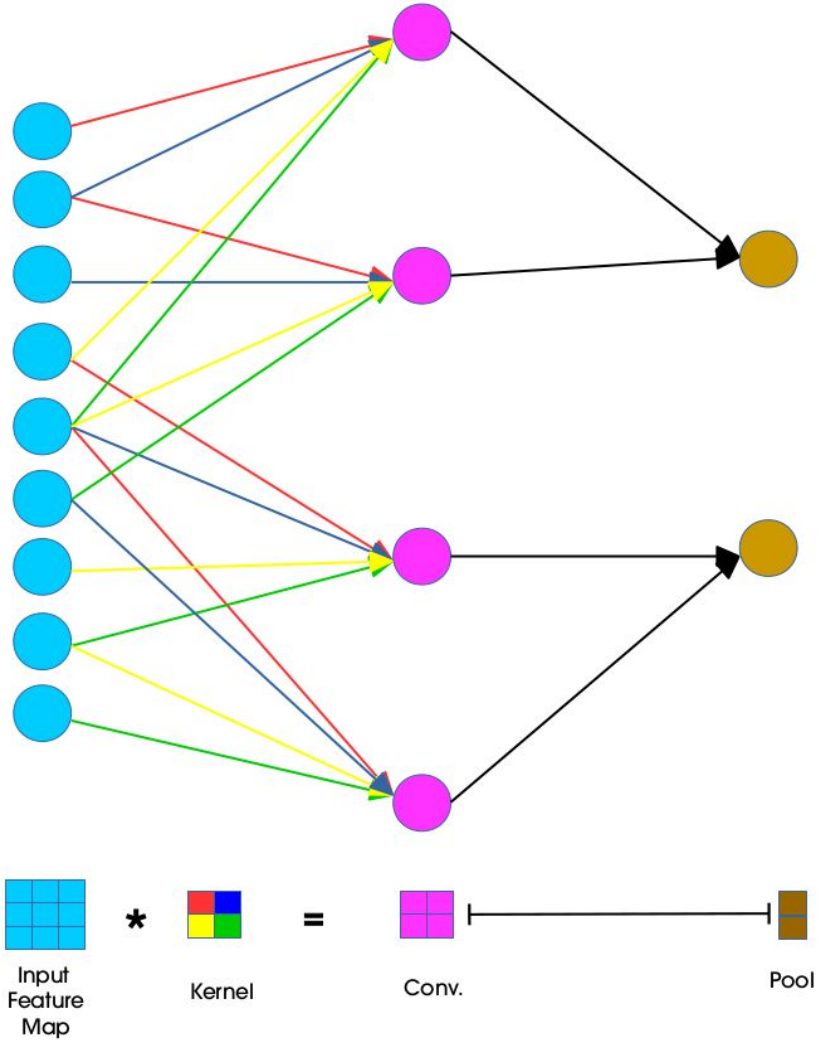
Error / Loss

- t_p - target labels
- a_p^L - predicted labels

$$E = \frac{1}{P}\sum_{p=1}^{P}(t_p - a_p^L)^2$$

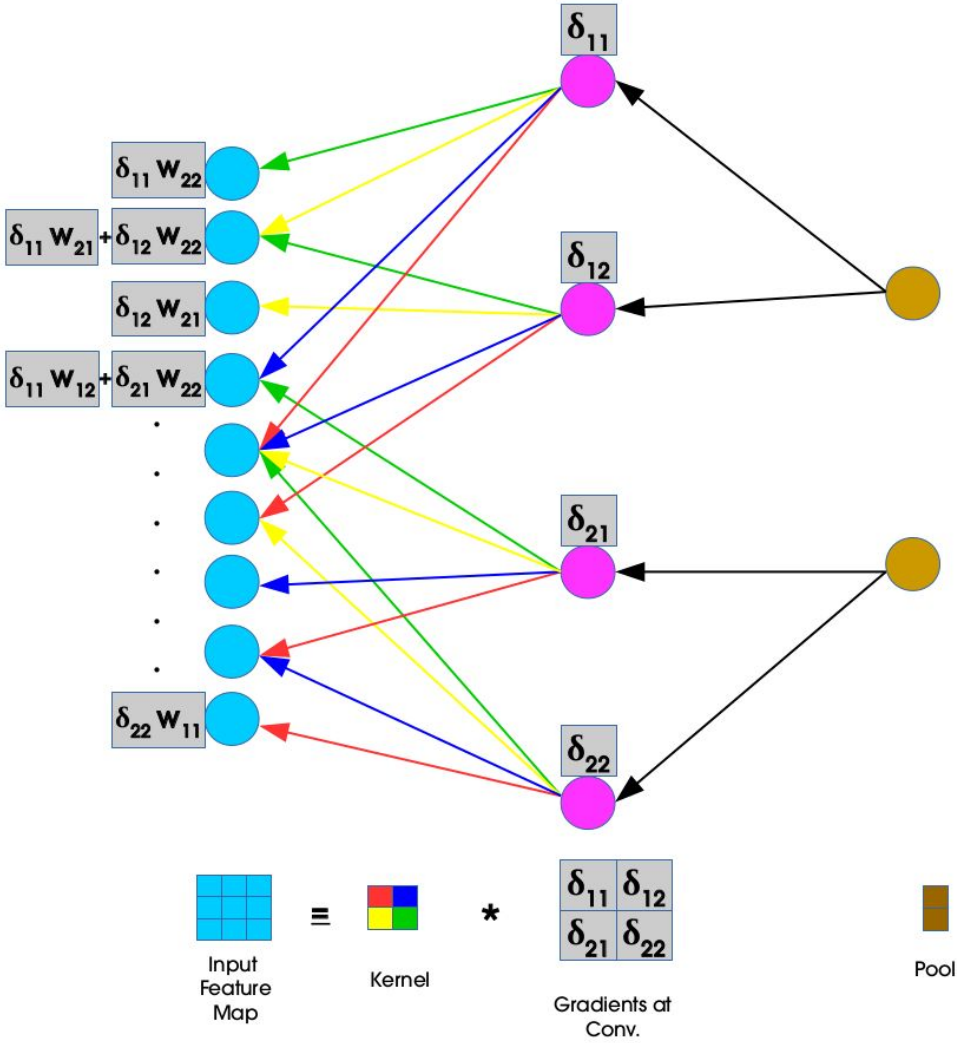
Backpropagation

Convolution layer forward prop



Backpropagation

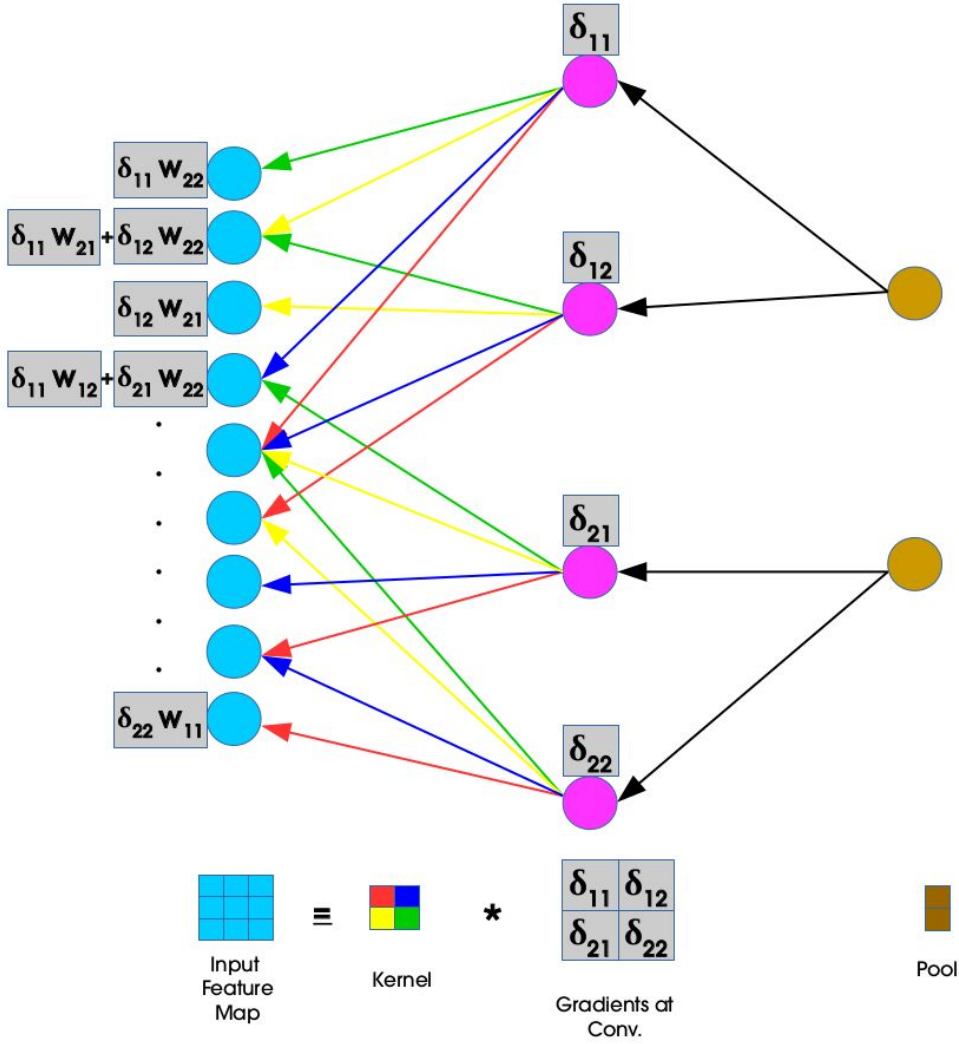
- Two updates are performed
  - For weights
  - For deltas



Backpropagation

$$\frac{\partial E}{\partial w_{i,j}^l} = \sum_{i'} \sum_{j'} \frac{\partial E}{\partial x_{i',j'}^l} \frac{\partial x_{i',j'}^l}{\partial w_{i,j}^l}$$

$$= \sum_{i'} \sum_{j'} \delta_{i',j'}^l \frac{\partial x_{i',j'}^l}{\partial w_{i,j}^l}$$



Backpropagation

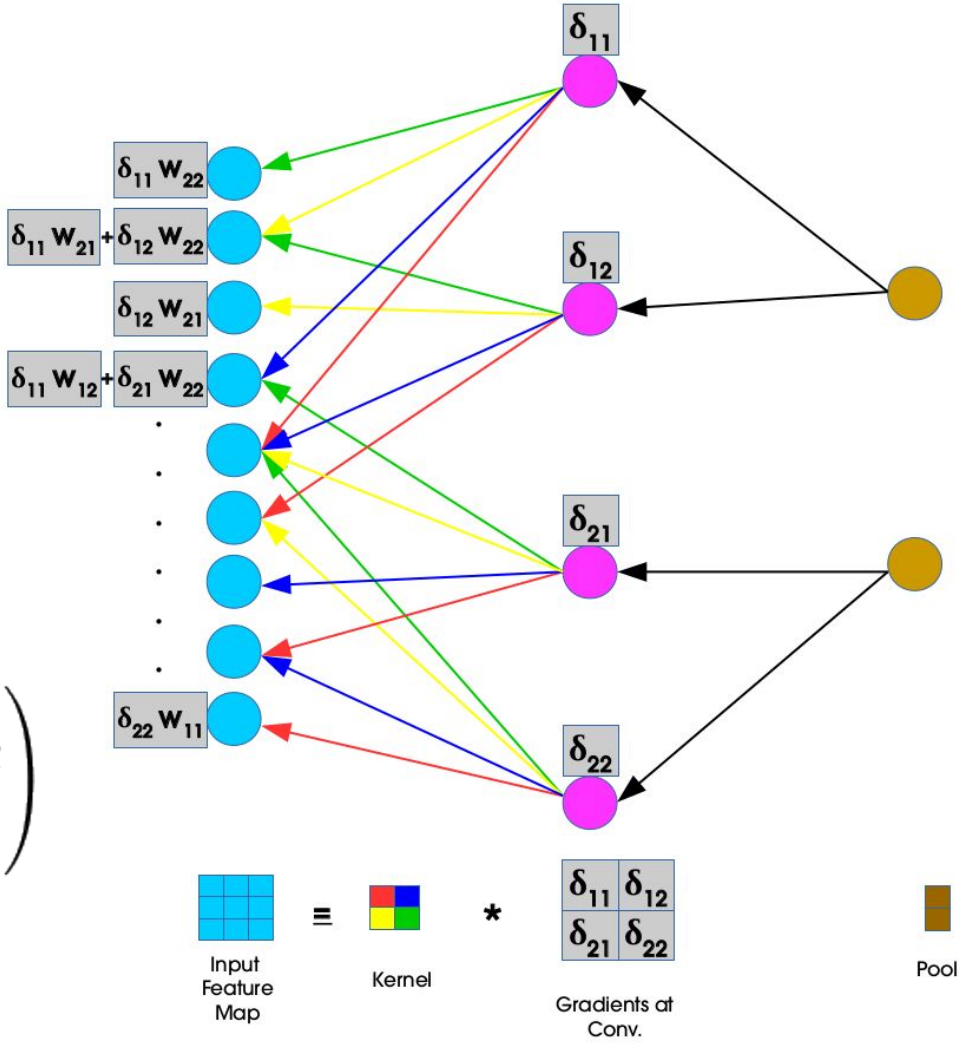
$$\frac{\partial E}{\partial w^l_{i,j}} = \sum_{i'}\sum_{j'} \frac{\partial E}{\partial x^l_{i',j'}} \frac{\partial x^l_{i',j'}}{\partial w^l_{i,j}}$$

$$= \sum_{i'}\sum_{j'} \delta^l_{i',j'} \frac{\partial x^l_{i',j'}}{\partial w^l_{i,j}}$$

Where,

$$\frac{\partial x^l_{i',j'}}{\partial w^l_{i,j}} = \frac{\partial}{\partial w^l_{i,j}} \left(\sum_{i''}\sum_{j''} w^l_{i'',j''} o^{l-1}_{i'-i'',j'-j''} + b^l \right)$$

$$= o^{l-1}_{i'-i,j'-j}$$



Backpropagation- Weight update

$$
\begin{aligned}
\frac{\partial E}{\partial w_{i,j}^{l}} &= \sum_{i'} \sum_{j'} \delta_{i',j'}^{l} o_{i'-i,j'-j}^{l-1} \\
&= \mathrm{rot}_{180^\circ} \left\{ \sum_{i'} \sum_{j'} \delta_{i',j'}^{l} o_{i'+i,j'+j}^{l-1} \right\} \\
&= \mathrm{rot}_{180^\circ} \left\{ \delta_{i,j}^{l} \right\} * o_{i,j}^{l-1}
\end{aligned}
$$

Delta update

$$\delta_{i,j}^{l} = \frac{\partial E}{\partial x_{i,j}^{l}}$$

$$\frac{\partial E}{\partial x_{i,j}^{l}} = \sum_{i'} \sum_{j'} \frac{\partial E}{\partial x_{i',j'}^{l+1}} \frac{\partial x_{i',j'}^{l+1}}{\partial x_{i,j}^{l}}$$

$$= \sum_{i'} \sum_{j'} \delta_{i',j'}^{l+1} \frac{\partial x_{i',j'}^{l+1}}{\partial x_{i,j}^{l}}$$

$$\frac{\partial x_{i',j'}^{l+1}}{\partial x_{i,j}^{l}} = \frac{\partial}{\partial x_{i,j}^{l}} \left(\sum_{i''} \sum_{j''} w_{i'',j''}^{l+1} o_{i'-i'',j'-j''}^{l} + b^{l+1} \right)$$

$$= \frac{\partial}{\partial x_{i,j}^{l}} \left(\sum_{i''} \sum_{j''} w_{i'',j''}^{l+1} f\left(x_{i'-i'',j'-j''}^{l} \right) + b^{l+1} \right)$$

Delta Update

$$
\begin{aligned}
\frac{\partial x_{i',j'}^{l+1}}{\partial x_{i,j}^{l}} &= \frac{\partial}{\partial x_{i,j}^{l}} \left(w_{0,0}^{l+1} f\left(x_{i'-0,j'-0}^{l}\right) + \cdots + w_{i'-i,j'-j}^{l+1} f\left(x_{i,j}^{l}\right) + \cdots + b^{l+1} \right) \\
&= \frac{\partial}{\partial x_{i,j}^{l}} \left(w_{i'-i,j'-j}^{l+1} f\left(x_{i,j}^{l}\right) \right) \\
&= w_{i'-i,j'-j}^{l+1} \frac{\partial}{\partial x_{i,j}^{l}} \left(f\left(x_{i,j}^{l}\right) \right) \\
&= w_{i'-i,j'-j}^{l+1} f'\left(x_{i,j}^{l}\right)
\end{aligned}
$$

Where $f'(x_{ij}^{l})$ is the derivative of the activation function $f(.)$

Delta update

$$\frac{\partial E}{\partial x_{i,j}^{l}} = \sum_{i'} \sum_{j'} \delta_{i',j'}^{l+1} w_{i'-i,j'-j}^{l+1} f'\left(x_{i,j}^{l}\right)$$

$$= \mathrm{rot}_{180^\circ} \left\{ \sum_{i'} \sum_{j'} \delta_{i',j'}^{l+1} w_{i'+i,j'+j}^{l+1} \right\} f'\left(x_{i,j}^{l}\right)$$

$$= \mathrm{rot}_{180^\circ} \left\{ \sum_{i'} \sum_{j'} \delta_{i'+i,j'+j}^{l+1} w_{i',j'}^{l+1} \right\} f'\left(x_{i,j}^{l}\right)$$

$$= \delta_{i,j}^{l+1} * \mathrm{rot}_{180^\circ} \left\{ w_{i,j}^{l+1} \right\} f'\left(x_{i,j}^{l}\right)$$

Where $f'(x_{ij}^{l})$ is the derivative of the activation function $f(.)$

Derivatives of some common activation functions

- Relu: $f'(x_{ij}^l) = \begin{cases} 1 & x_{ij}^l > 0 \\ 0 & x_{ij}^l \leq 0 \end{cases}$
- Sigmoid: $f'(x_{ij}^l) = (f(x_{ij}^l))(1 - f(x_{ij}^l))$
- Tanh: $f'(x_{ij}^l) = 1 - f(x_{ij}^l)^2$

Backprop in Pooling Layer

- Max Pooling
  - the error is just assigned to where it comes from
- Average Pooling
  - The error is multiplied by 1/(NxN) and assigned to the whole pooling block

Topics

Backpropagation in Convolutional Neural Networks

Batch Normalization

Dropout

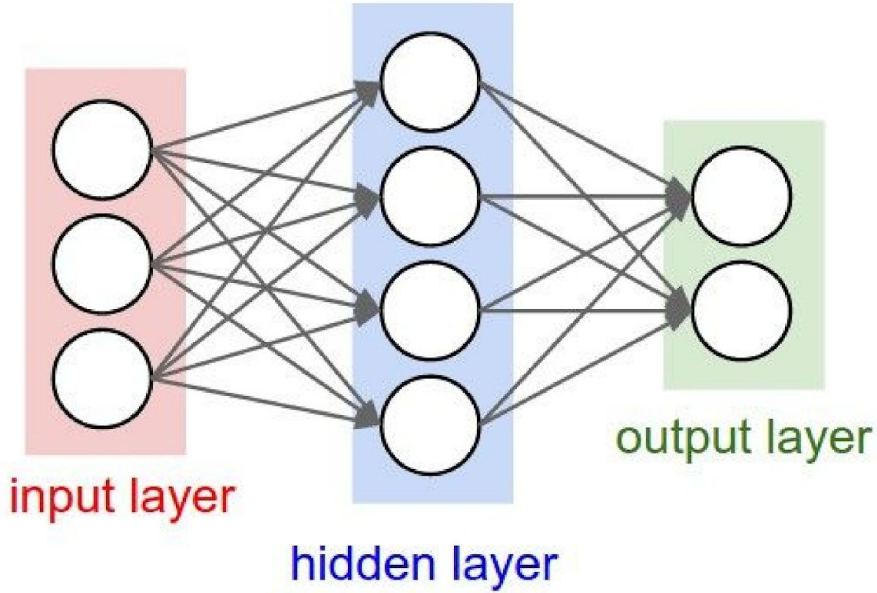
Finetuning/ Transfer Learning

Batch Normalization

Slides derived from Stanford CS231n and Berkeley CS294 lecture slides

Weight Initialization in Neural Nets

- Q: what happens when W=0 init is used?



- First idea: **Small random numbers**
(gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

Works ~okay for small networks, but problems with deeper networks.

Lets look at some activation statistics

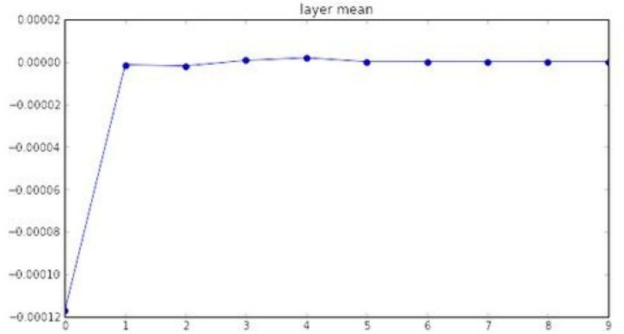
E.g. 10-layer net with 500 neurons on each layer, using tanh non-linearities, and initializing as described in last slide.

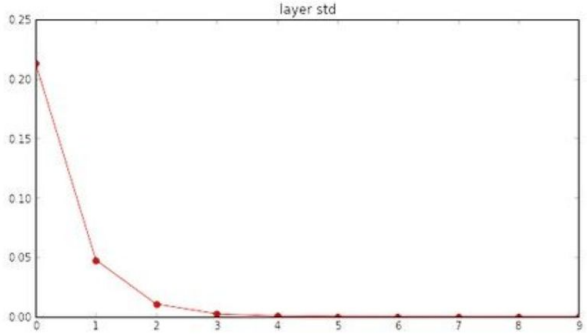
```
# assume some unit gaussian 10-D input data
D = np.random.randn(1000, 500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)
```

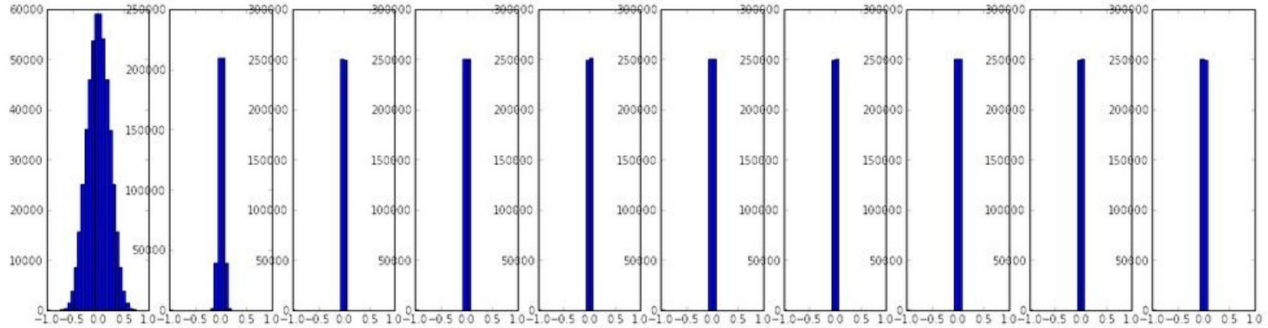
```
act = {'relu':lambda x:np.maximum(0,x), 'tanh':lambda x:np.tanh(x)}
Hs = {}
for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1] # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01 # layer initialization

    H = np.dot(X, W) # matrix multiply
    H = act[nonlinearities[i]](H) # nonlinearity
    Hs[i] = H # cache result on this layer
```

```
# look at distributions at each layer
print 'input layer had mean %f and std %f' % (np.mean(D), np.std(D))
layer_means = [np.mean(H) for i,H in Hs.iteritems()]
layer_stds = [np.std(H) for i,H in Hs.iteritems()]
for i,H in Hs.iteritems():
    print 'hidden layer %d had mean %f and std %f' % (i+1, layer_means[i], layer_stds[i])

# plot the means and standard deviations
plt.figure()
plt.subplot(121)
plt.plot(Hs.keys(), layer_means, 'ob-')
plt.title('layer mean')
plt.subplot(122)
plt.plot(Hs.keys(), layer_stds, 'or-')
plt.title('layer std')

# plot the raw distributions
plt.figure()
for i,H in Hs.iteritems():
    plt.subplot(1,len(Hs),i+1)
    plt.hist(H.ravel(), 30, range=(-1,1))
```

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```







All activations become zero!

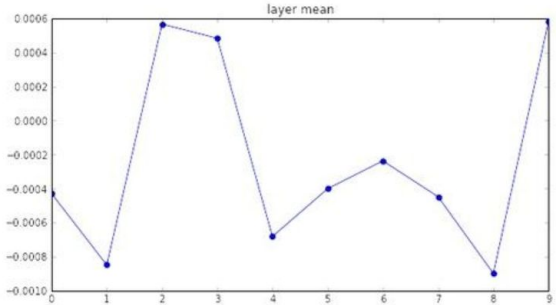
Q: think about the backward pass. What do the gradients look like?

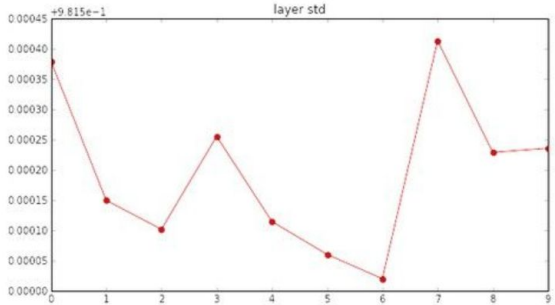
Hint: think about backward pass for a W*X gate.

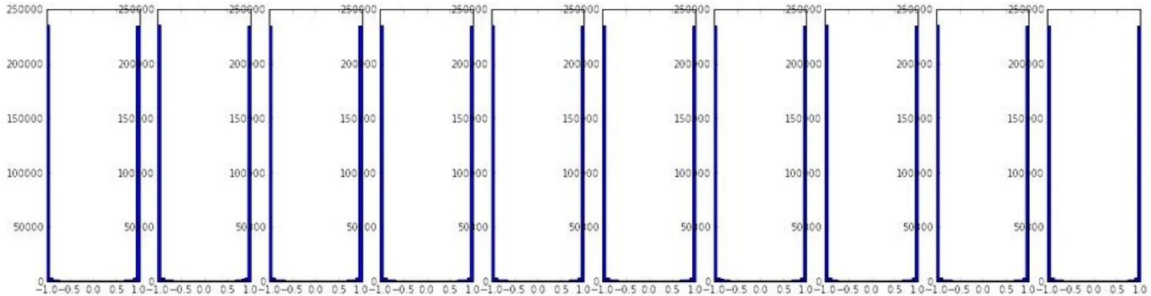
```
W = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736
```

*1.0 instead of *0.01







Almost all neurons completely saturated, either -1 and 1. Gradients will be all zero.

Motivation - Internal covariate shift

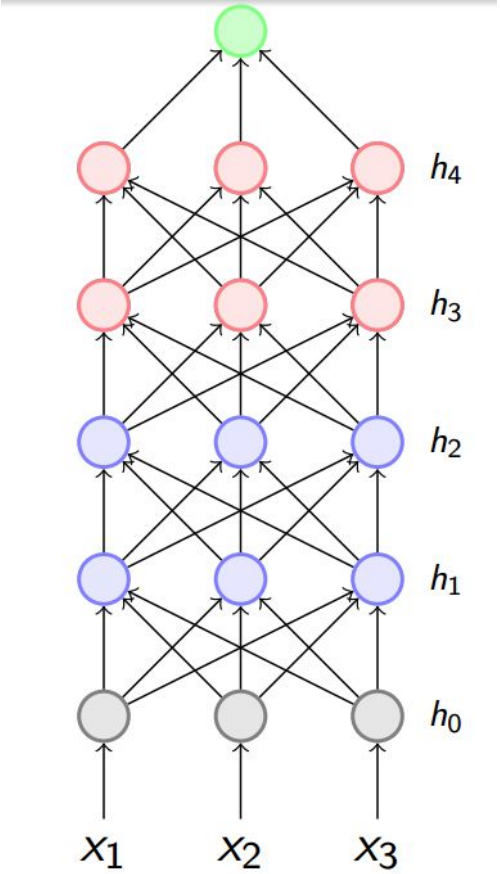
- Covariate shift : change in the input distribution to a learning system
  - Violating iid assumption

Motivation - Internal covariate shift

- Covariate shift : change in the input distribution
  - Violating iid assumption
- Internal covariate shift : change in input distribution at the nodes of of a neural network

Motivation

- Consider a deep neural network
- Consider the learning of the weights at layer h_3,h_4
- Output distribution of the layer h_3 is changing during each iteration since the weights are getting updated during training
- This makes the learning process slow for h_4 .

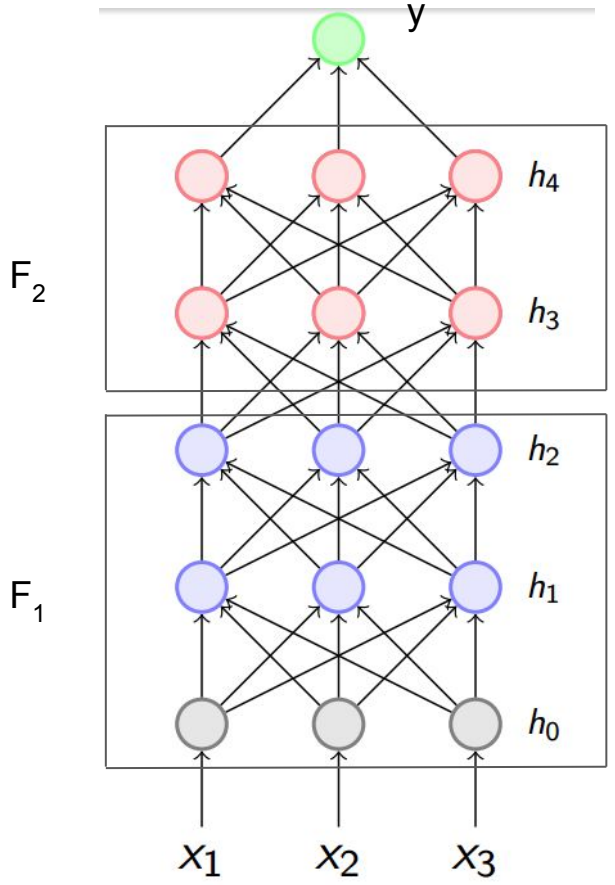


Motivation

- Consider a deep neural network

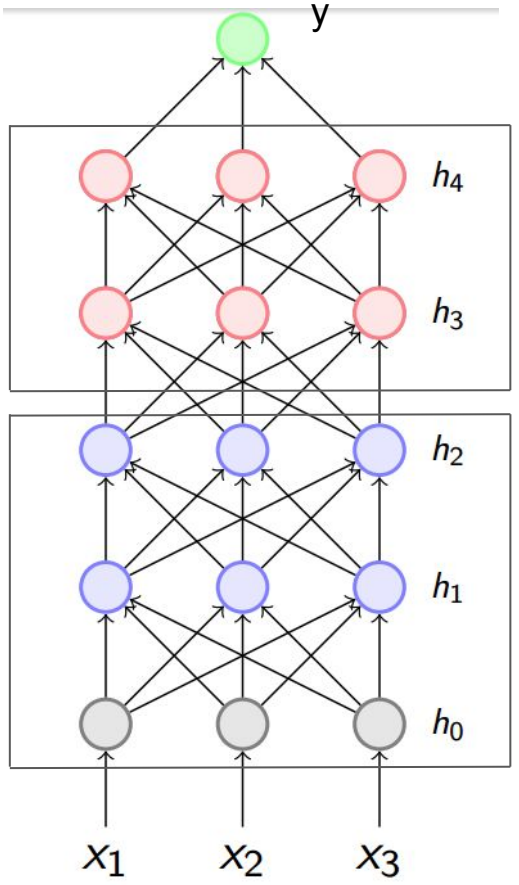
$$y = F_2(F_1(x; \theta_1); \theta_2)$$

$$y' = F_1(x; \theta_1)$$



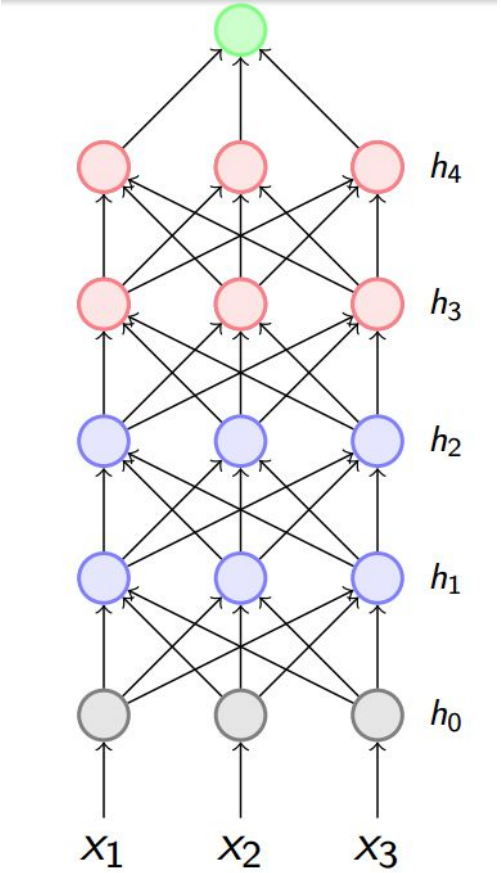
Motivation

- Consider a deep neural network
- P(x) remains constant
- P(y') keeps changing since parameters θ_1 are changing with each iteration
- Machine learning model assumes that data is sampled from the same distribution each time
- For F_2 this assumption is violated



Motivation

- It’s desirable to have the output distribution of each layer as zero mean and unit variance Gaussian
- Why not explicitly transform the output distribution of each layer to a zero mean and unit variance Gaussian?



Solution

- Whiten the input (LeCun et. al. 1998)

Solution

- Whiten the input (LeCun et. al. 1998)
  - Problem?

Solution

- Whiten the input (LeCun et. al. 1998)
  - Need to compute covariance matrix, which is very very costly

Tractable solution - Batch normalization

- Consider a batch of activations x^k at any layer and apply

$$\hat{x}^k = \frac{x^k - E[x^k]}{\sqrt{Var[x^k]}}$$

Tractable solution - Batch normalization

- Consider a batch of activations x^k at any layer and apply

$$\hat{x}^k = \frac{x^k - E[x^k]}{\sqrt{Var[x^k]}}$$

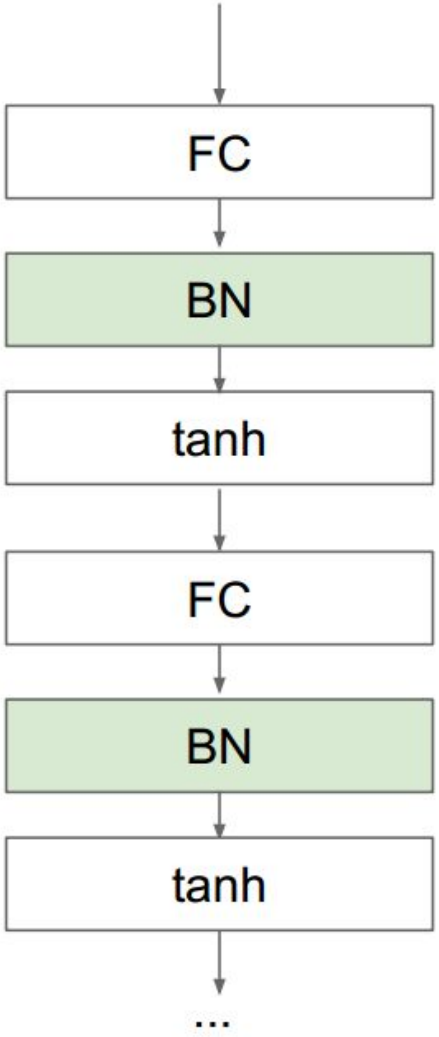
- Is this differentiable?

Batch normalization (BN) layer

- Where to insert the Batch Normalization (BN) layer

Batch normalization (BN) layer

- Where to insert the Batch Normalization (BN) layer



Batch normalization (BN) layer

- Normalize

$$\hat{x}^k = \frac{x^k - E[x^k]}{\sqrt{Var[x^k]}}$$

- And allow the network to squash the input range

$$y^k = \gamma^k \hat{x}^k + \beta^k$$

- γ^k and β^k are trainable parameters

Batch normalization (BN) layer

- Normalize

$$\hat{x}^k = \frac{x^k - E[x^k]}{\sqrt{Var[x^k]}}$$

- And allow the network to squash the input range

$$y^k = \gamma^k \hat{x}^k + \beta^k$$

- γ^k and β^k are trainable parameters
- How to make $\hat{x}^k = x^k$? Think in terms of the parameters γ^k and β^k

Advantages of BN

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization
- Accelerates training

What happens at test time?

- Do you still compute the mean and variance of the test batch for the BN layer?

What happens at test time?

- A single empirical mean and variance of activations during training is used
  - Can be estimated using running averages

Summary

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

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

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

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

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

Topics

Backpropagation in Convolutional Neural Networks

Batch Normalization

Dropout

Finetuning/ Transfer Learning

Dropout

Slides and figures derived from Sargur Srihari and Stanford CS231n lecture slides

Dropout

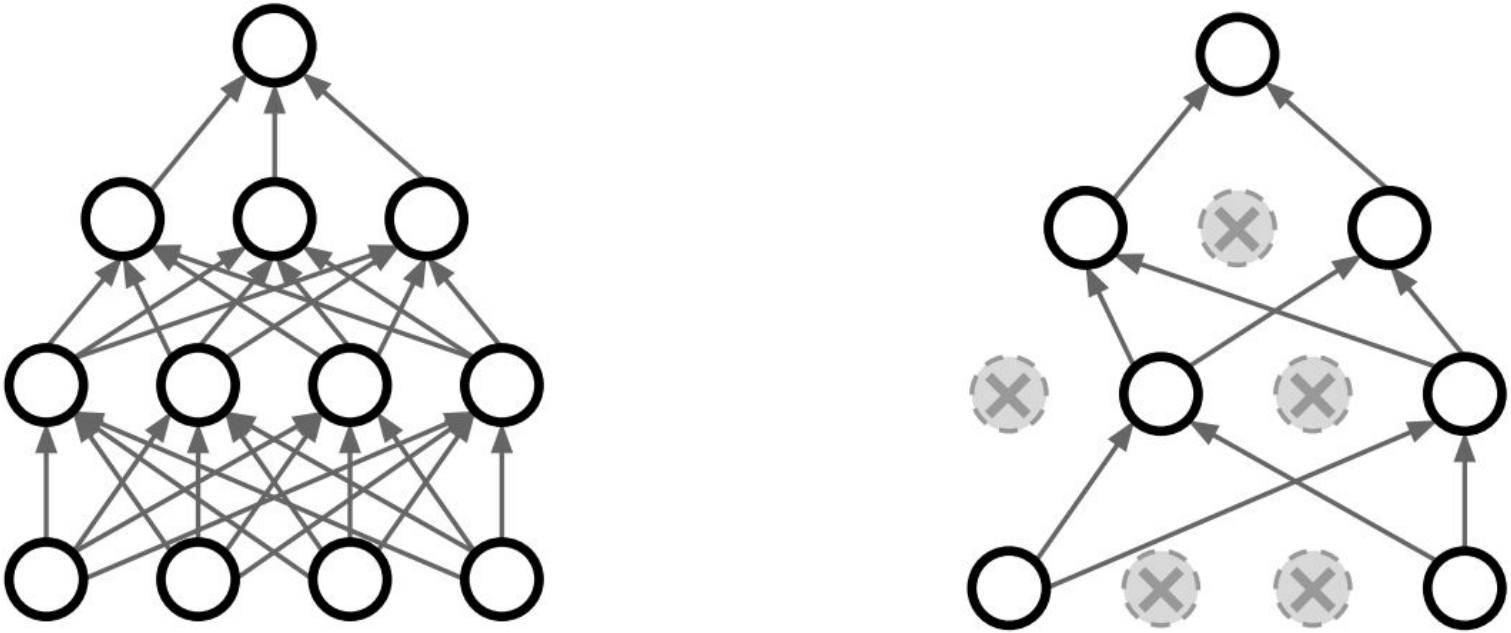
- A type of regularization
  - A method of ensemble learning applied to neural networks

Dropout

- A type of regularization
  - A method of ensemble learning applied to neural networks
- Impractical to train many neural networks
  - Dropout makes it practical

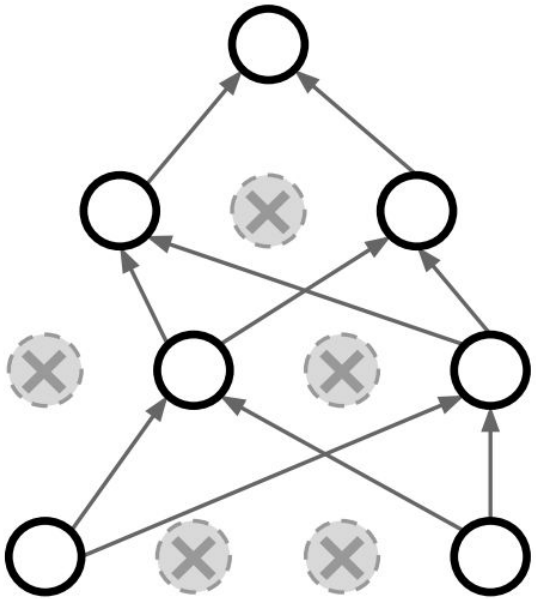
Dropout

- In each forward pass randomly set some units to zero.



Why Dropout is a good idea?

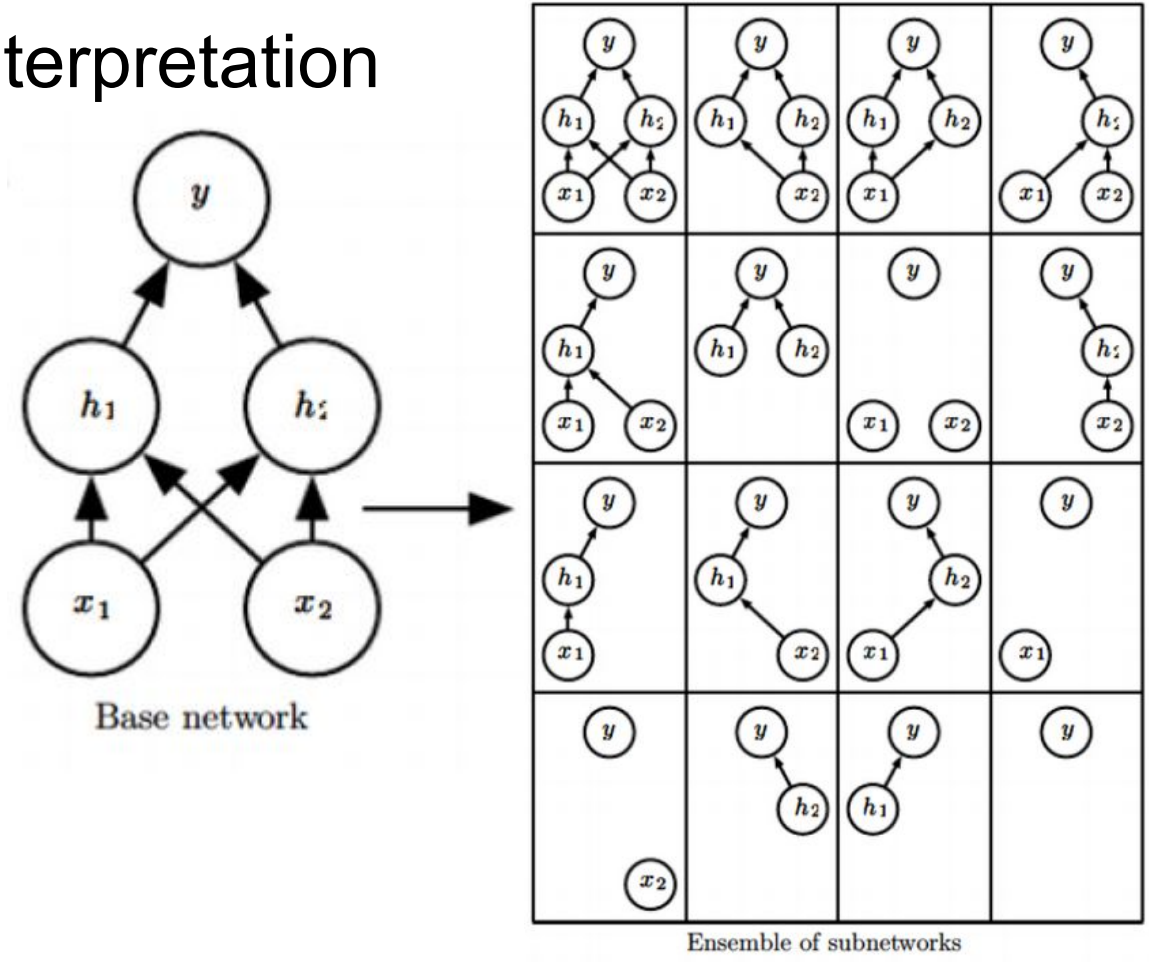
Why Dropout is a good idea?



Forces the network to have a redundant representation;
Prevents co-adaptation of features

Dropout - another interpretation

Ensemble of networks



Dropout training

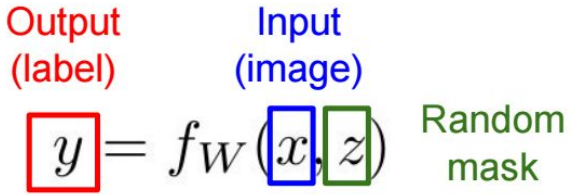
- At each step randomly sample a binary mask
  - Probability of including a unit is a hyperparameter - typically 0.5

Dropout training

- At each step randomly sample a binary mask
  - Probability of including a unit is a hyperparameter - typically 0.5
- Multiply the units by the binary mask
  - Forward prop proceeds as usual

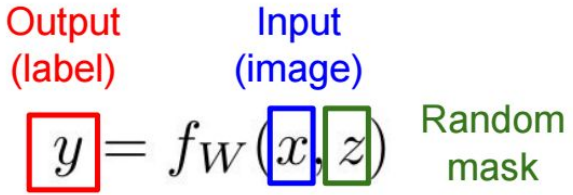
Dropout - test time

- Dropout makes the output random



Dropout - test time

- Dropout makes the output random
- Average out randomness



$$y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz$$

Dropout - test time

- Dropout makes the output random
- Average out randomness

Output (label) Input (image)

$$y = f_W(x, z)$$

Random mask

$$y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz$$

Integral is hard to compute

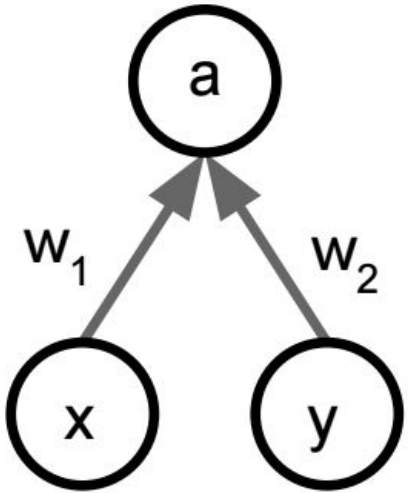
Dropout - test time

- Approximate the integral $y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz$

Dropout - test time

- Approximate the integral $y = f(x) = E_z[f(x,z)] = \int p(z) f(x,z) dz$

Consider a single neuron

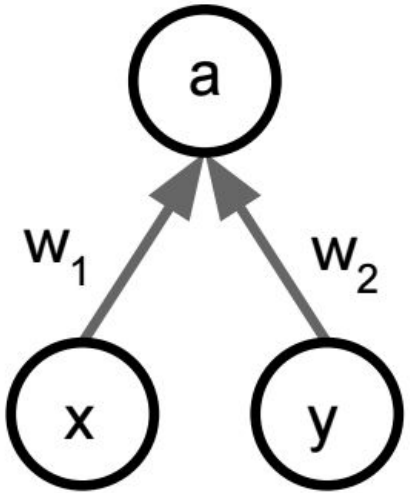


At test time, we have $E[a] = w_1 x + w_2 y$

Dropout - test time

- Approximate the integral $y = f(x) = E_z[f(x,z)] = \int p(z) f(x,z) dz$

Consider a single neuron



At test time, we have $E[a] = w_1 x + w_2 y$

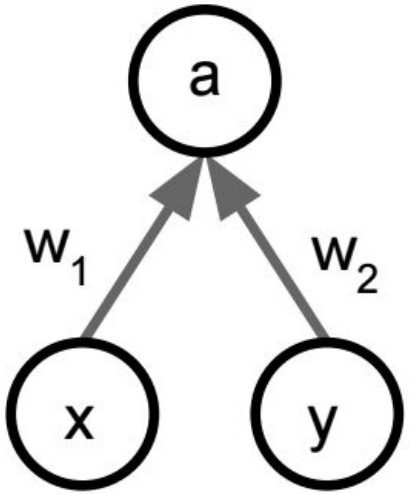
During training, we have

$$\begin{aligned} E[a] =& \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\ &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ =& \frac{1}{2}(w_1 x + w_2 y) \end{aligned}$$

Dropout - test time

- Approximate the integral $y = f(x) = E_z[f(x,z)] = \int p(z) f(x,z) dz$

Consider a single neuron



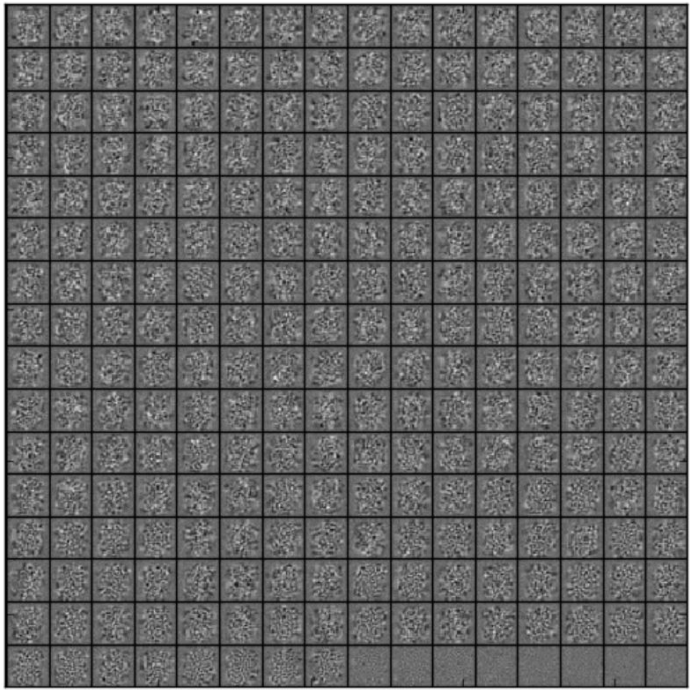
At test time, we have $E[a] = w_1 x + w_2 y$

During training, we have

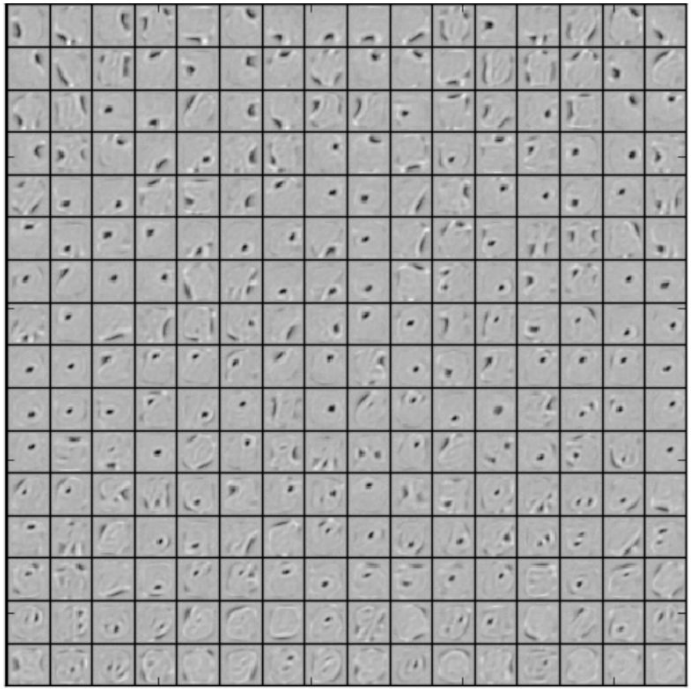
$$E[a] = \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) = \frac{1}{2}(w_1 x + w_2 y)$$

At test time, multiply by dropout probability

Effects of using Dropout



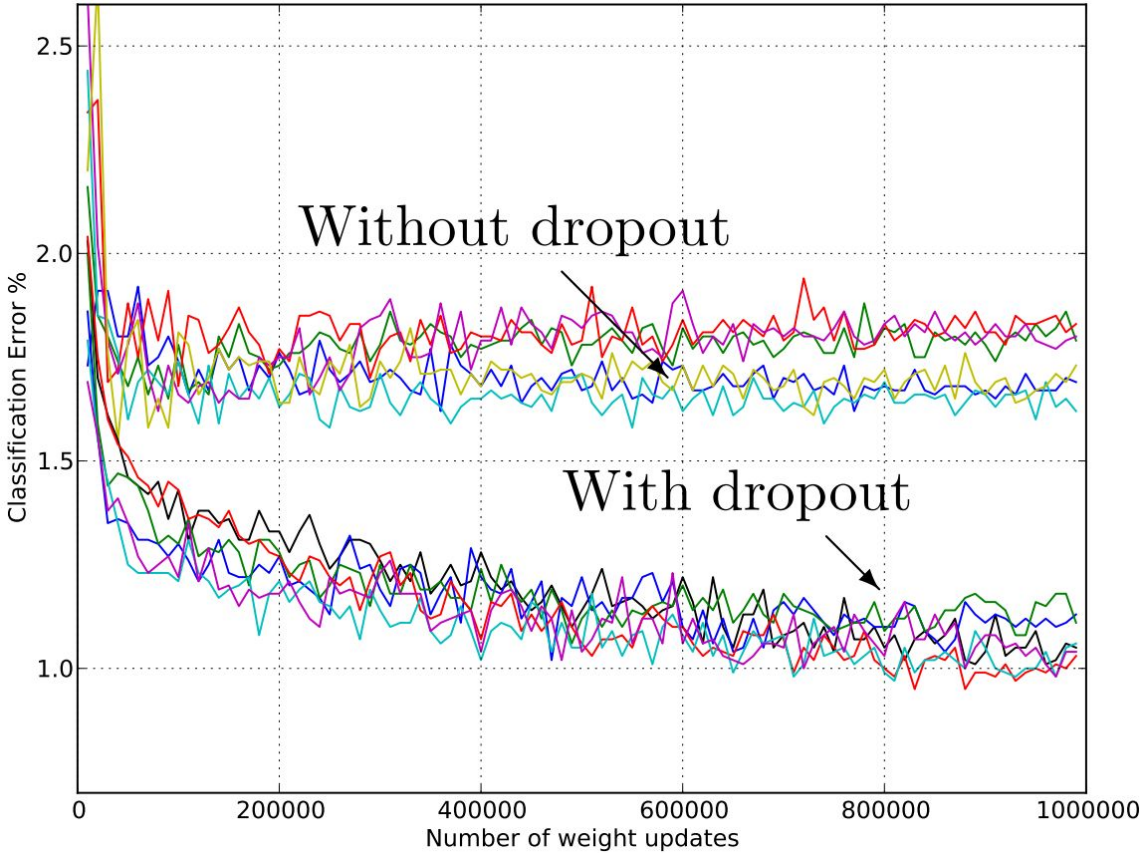
(a) Without dropout



(b) Dropout with $p = 0.5$.

Effects on a set of feature detectors taken from a convolutional neural network. While the features in (a) are mostly indistinguishable for humans, and seem to contain a big portion of white noise, the features in (b) are the product of a training with dropout. As visible the detectors are able to filter meaningful features such as spots, corners and strokes in an image. Image source: https://wiki.tum.de/display/lfdv/Dropout

Effects of using Dropout



Comparison of test error with and without dropout. Each color represents a different network architecture, ranging from 2 to 4 hidden layers (each fully connected) and 1024 to 2048 units. Image source: https://wiki.tum.de/display/lfdv/Dropout

Topics

Backpropagation in Convolutional Neural Networks

Batch Normalization

Dropout

Finetuning/ Transfer Learning

Transfer Learning

"You need a lot of a data if you want to train/use CNNs"

Slides taken from CS231n lecture slides

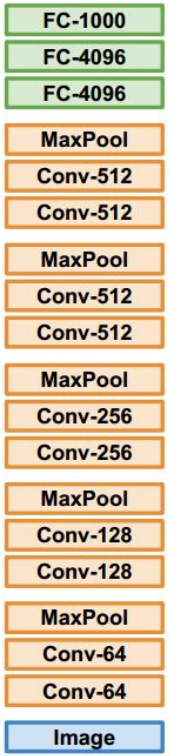
Transfer Learning

“You need a lot of a data if you want to train/use CNNs”

Transfer Learning with CNNs

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

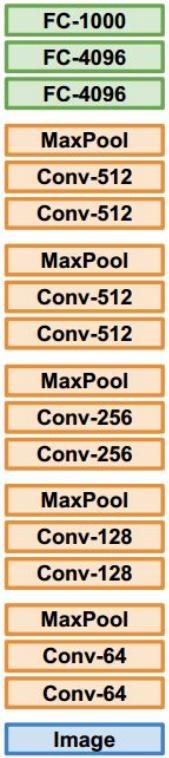
1. Train on Imagenet



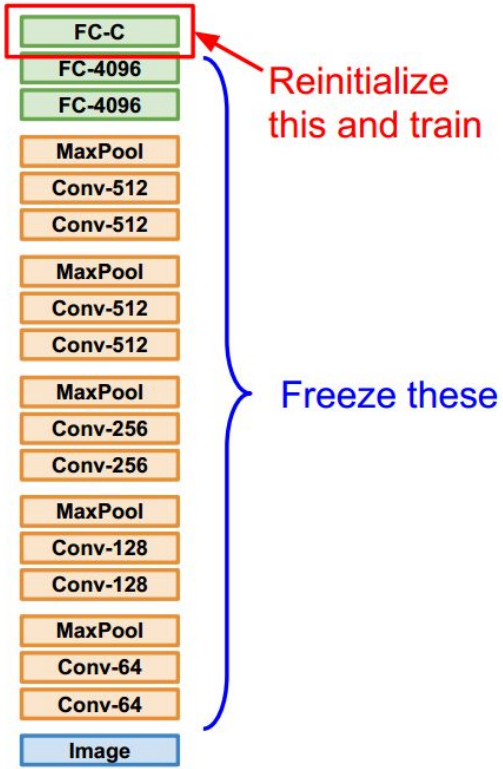
Transfer Learning with CNNs

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

1. Train on Imagenet



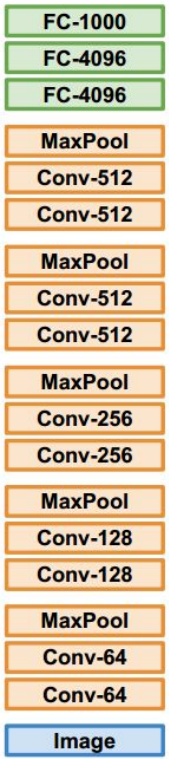
2. Small Dataset (C classes)



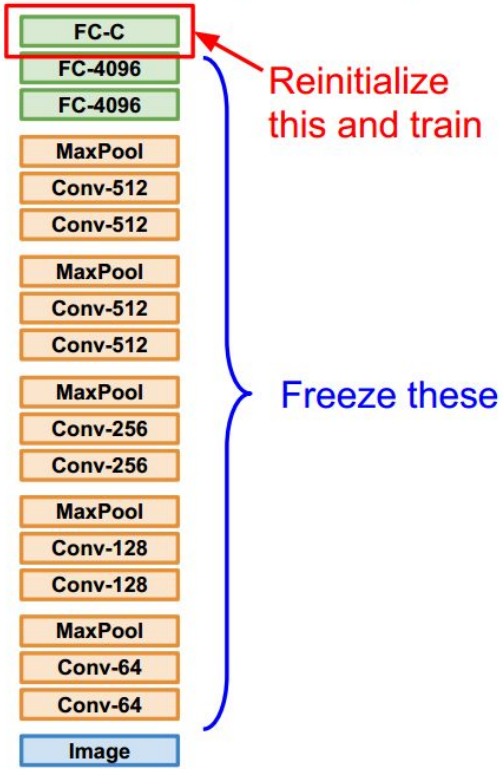
Transfer Learning with CNNs

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

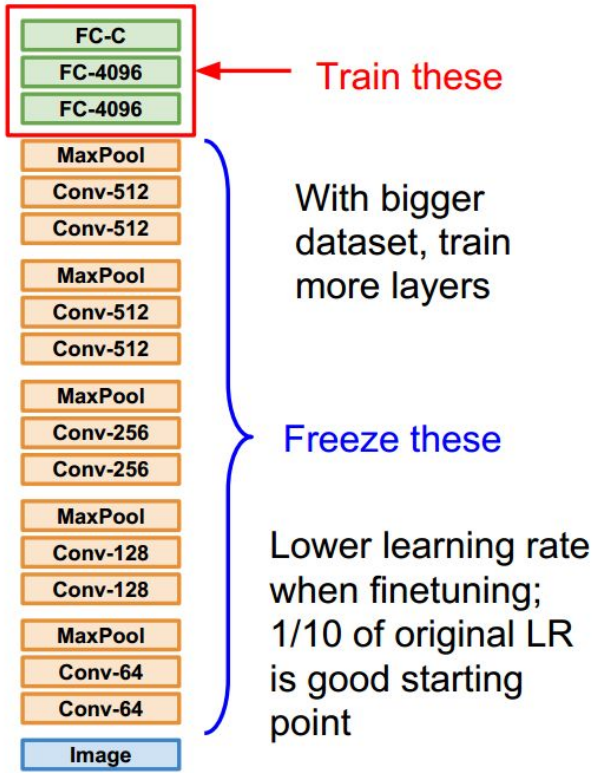
1. Train on Imagenet

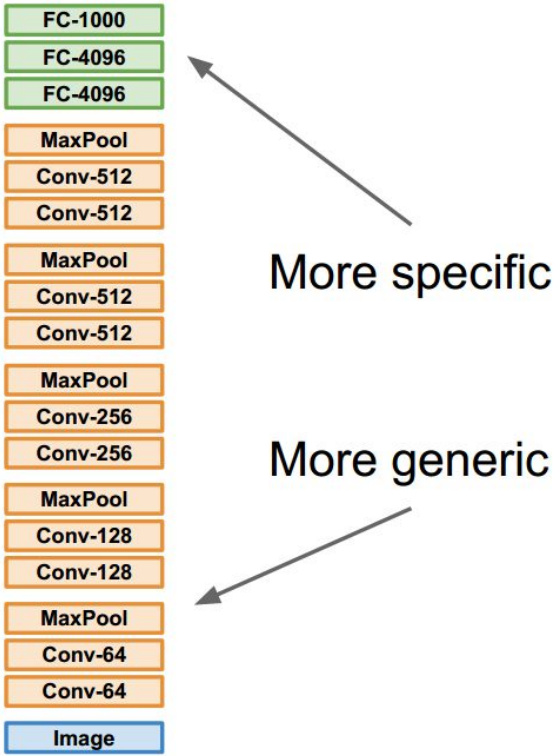


2. Small Dataset (C classes)

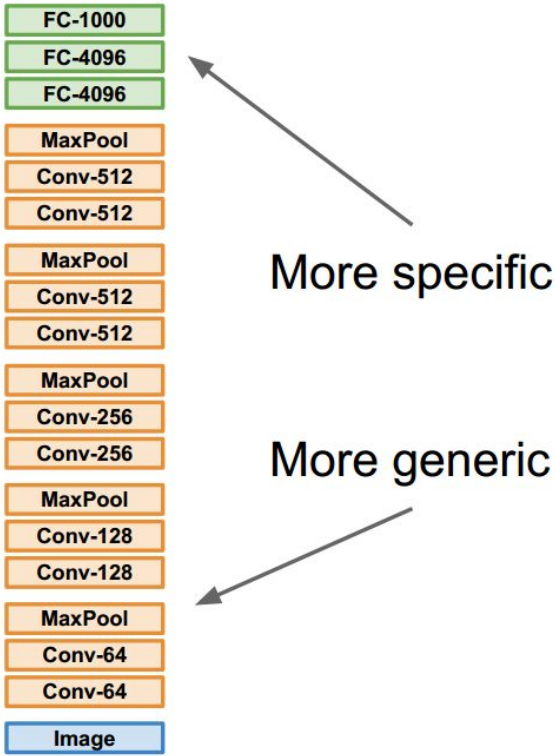


3. Bigger dataset

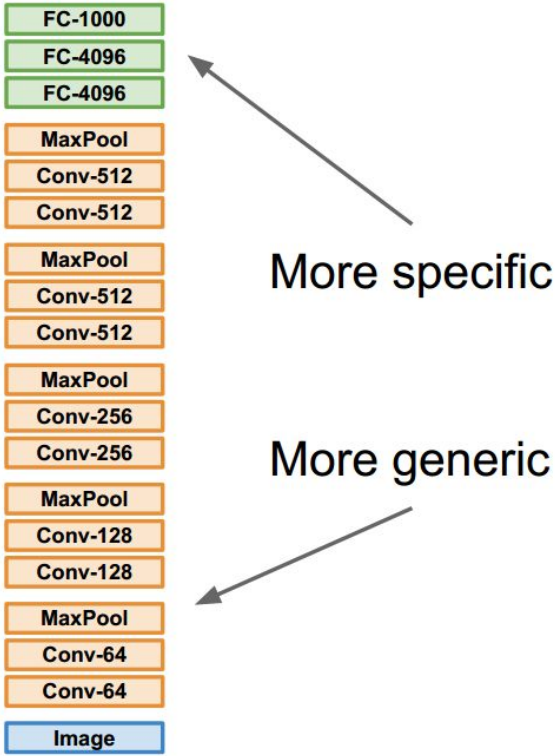




	very similar dataset	very different dataset
very little data	?	?
quite a lot of data	?	?



	very similar dataset	**very different dataset**
very little data	Use Linear Classifier on top layer	?
quite a lot of data	Finetune a few layers	?



	very similar dataset	**very different dataset**
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

Transfer learning with CNNs is pervasive…
(it's the norm, not an exception)

Object Detection
(Fast R-CNN)

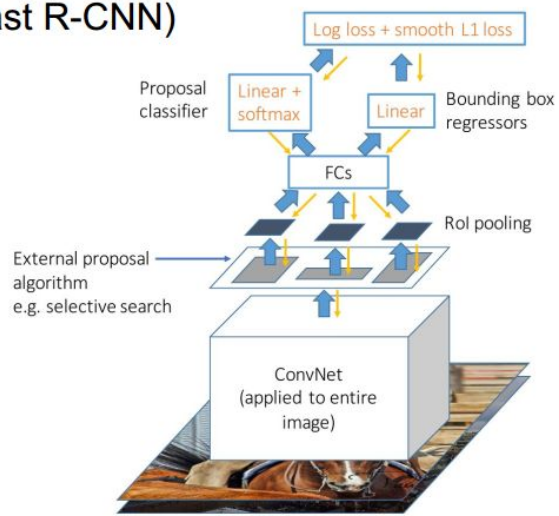
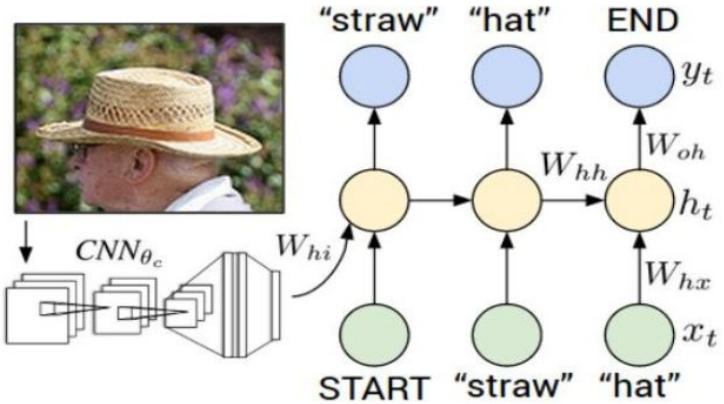


Image Captioning: CNN + RNN



Girshick, "Fast R-CNN", ICCV 2015
Figure copyright Ross Girshick, 2015. Reproduced with permission.

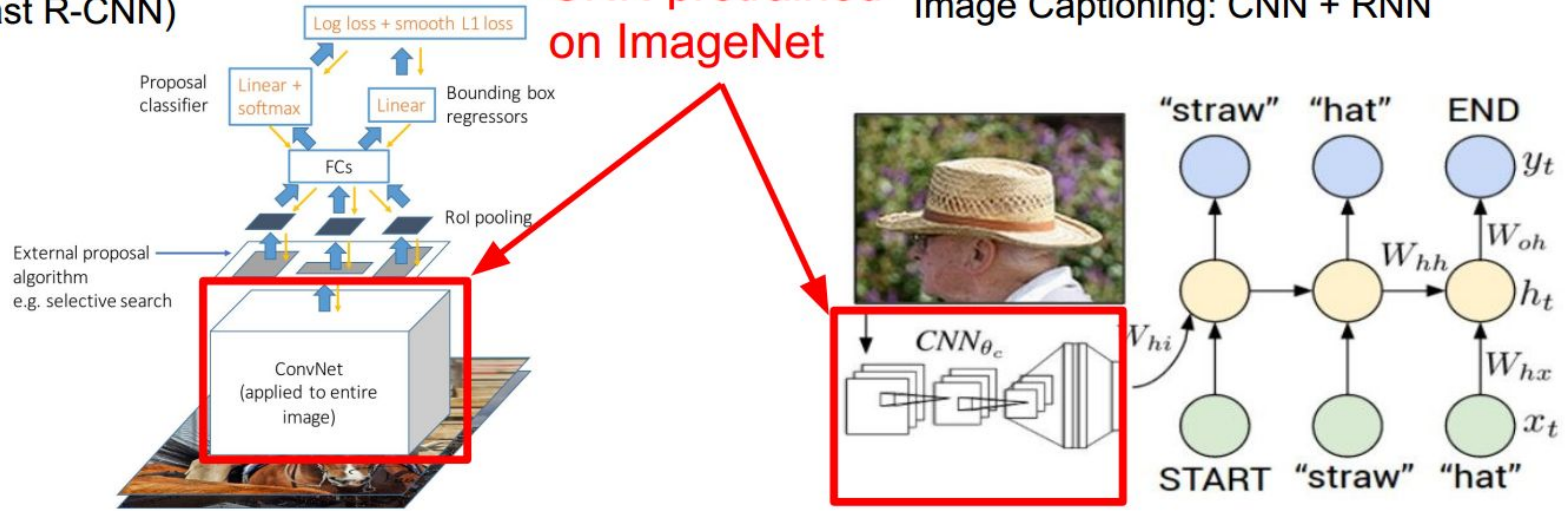
Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

Transfer learning with CNNs is pervasive…
(it's the norm, not an exception)

Object Detection
(Fast R-CNN)

CNN pretrained on ImageNet

Image Captioning: CNN + RNN

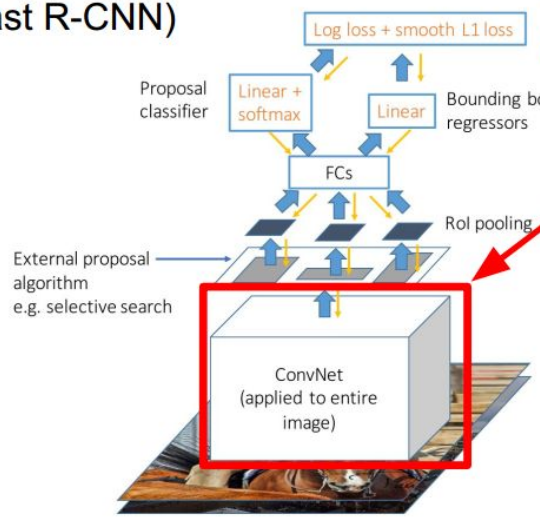


Girshick, "Fast R-CNN", ICCV 2015
Figure copyright Ross Girshick, 2015. Reproduced with permission.

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

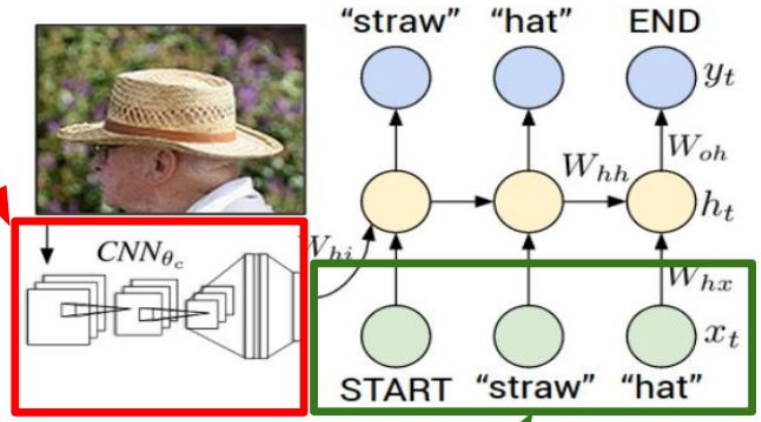
Transfer learning with CNNs is pervasive…
(it's the norm, not an exception)

Object Detection
(Fast R-CNN)



CNN pretrained on ImageNet

Image Captioning: CNN + RNN



Word vectors pretrained with word2vec

Girshick, "Fast R-CNN", ICCV 2015
Figure copyright Ross Girshick, 2015. Reproduced with permission.

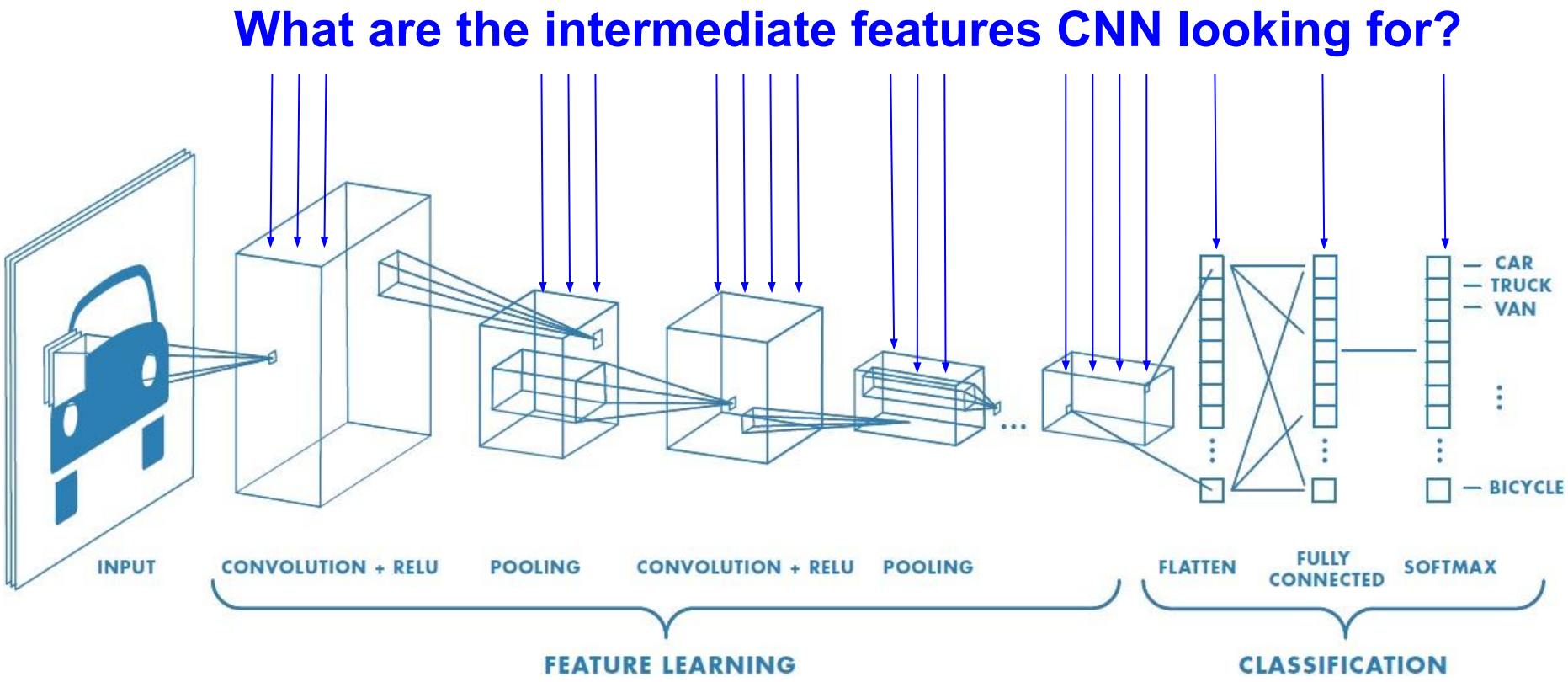
Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

Topics

Understanding and Visualizing CNN.

1. Visualizing the layer activations or Conv/FC filters
2. Image embedding
3. Saliency/occlusion
4. Activation maximization
5. Feature inversion
6. Back-propagating activations
7. Fooling CNN

Understanding and Visualizing CNN.



Topics

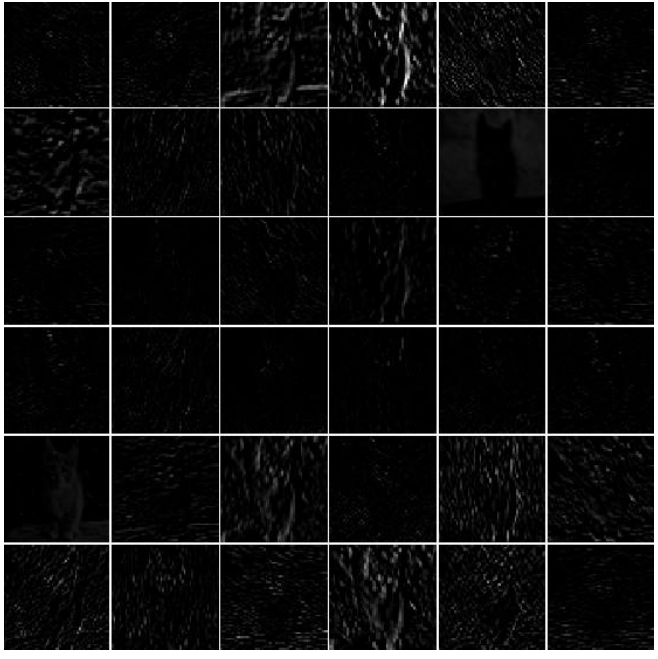
Understanding and Visualizing CNN.

1. Visualizing the layer activations or Conv/FC filters
2. Image embedding
3. Saliency/occlusion
4. Activation maximization
5. Feature inversion
6. Back-propagating activations
7. Fooling CNN

Visualizing the layer activations

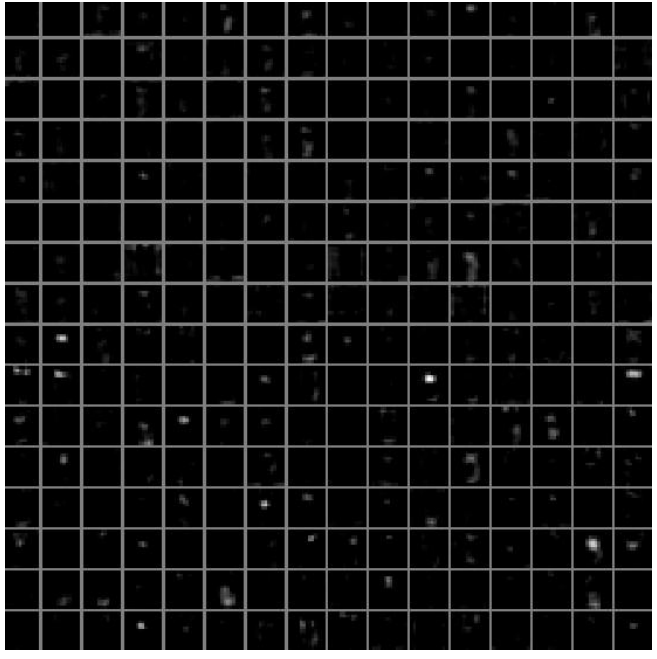
For ReLU networks

1. The activations usually start out looking relatively **blobby and dense**.
2. As the training progresses the activations usually become **more sparse and localized**.
3. Activation maps which are zero for *many different* inputs **indicate *dead* filters** (a symptom of high learning rates).



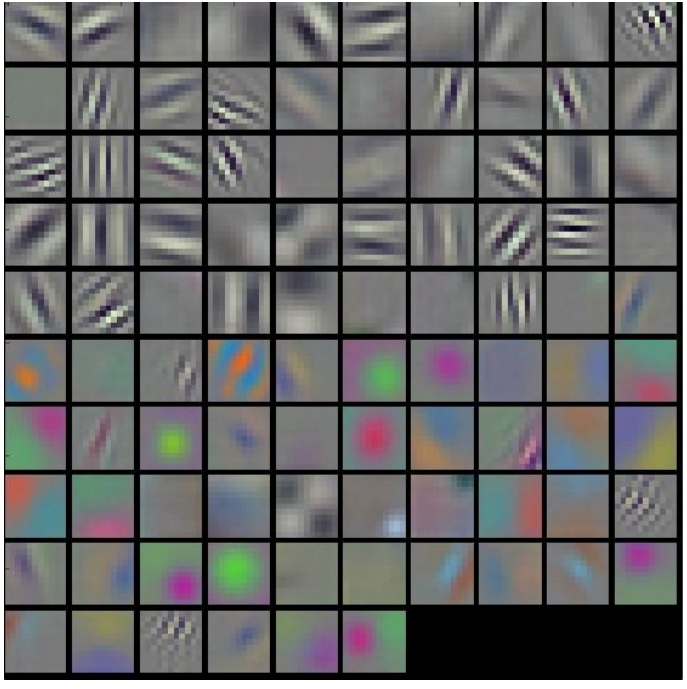
AlexNet
1st CONV
layer

AlexNet
5th CONV
layer
(at a picture of cat)



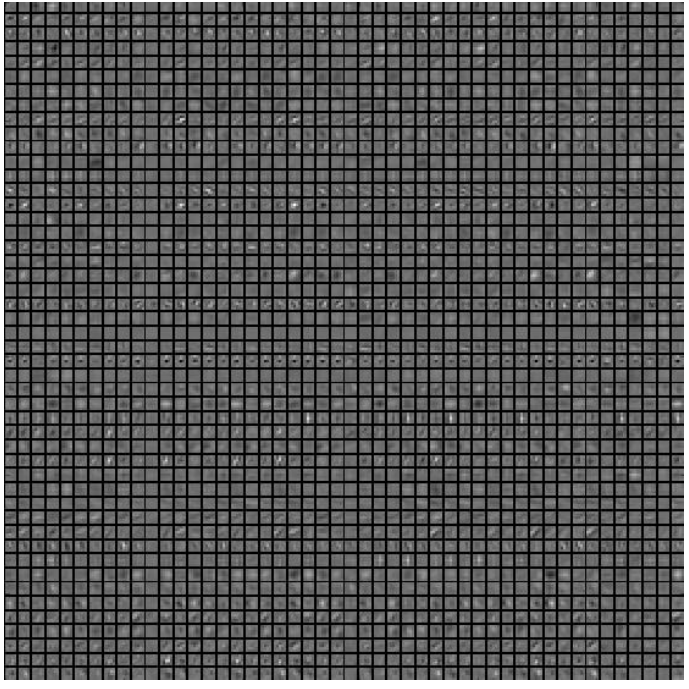
Visualizing the Conv/FC filters

1. **First Conv layer** weights are the most interpretable (since they operate on image pixels).
2. Well-trained networks usually display **nice and smooth filters**.
3. **Noisy patterns** can be an indicator of a network
   a. that hasn't been trained for long enough, or
   b. possibly a very low regularization strength that may have led to overfitting.



AlexNet
1st Conv
layer filters

AlexNet
2nd Conv
layer filters

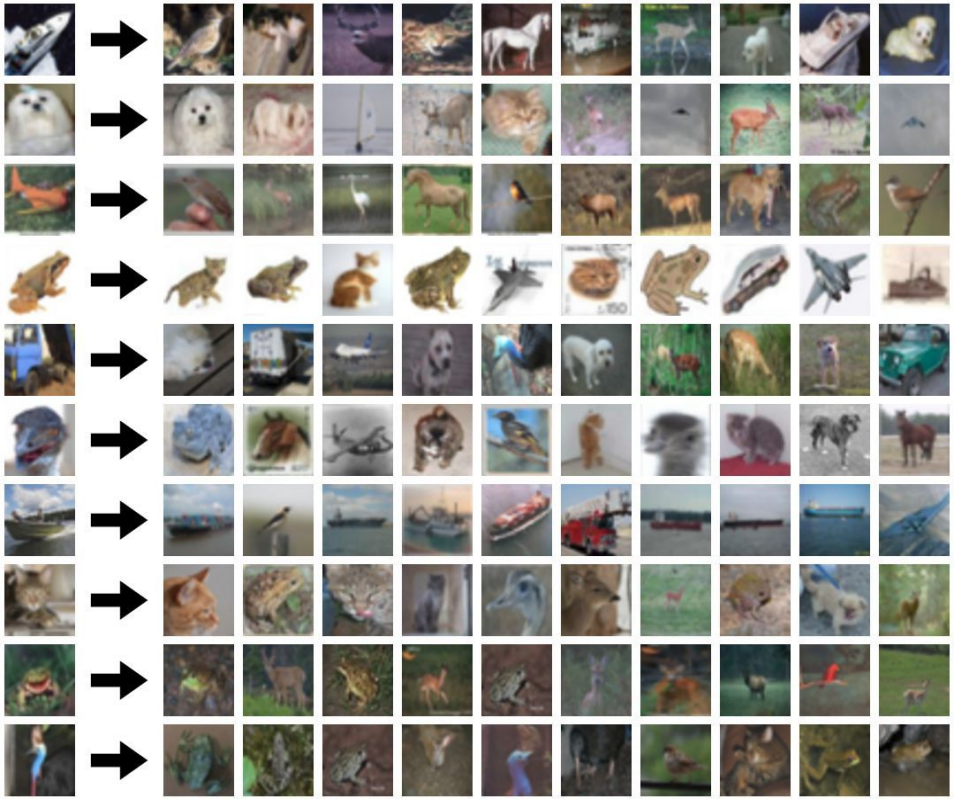


Topics

Understanding and Visualizing CNN.

1. Visualizing the layer activations or Conv/FC filters
2. Image embedding
3. Saliency/occlusion
4. Activation maximization
5. Feature inversion
6. Fine-grained recognition
7. Back-propagating activations

Last layer features: Nearest Neighbours



Input **Nearest neighbors in pixel space**

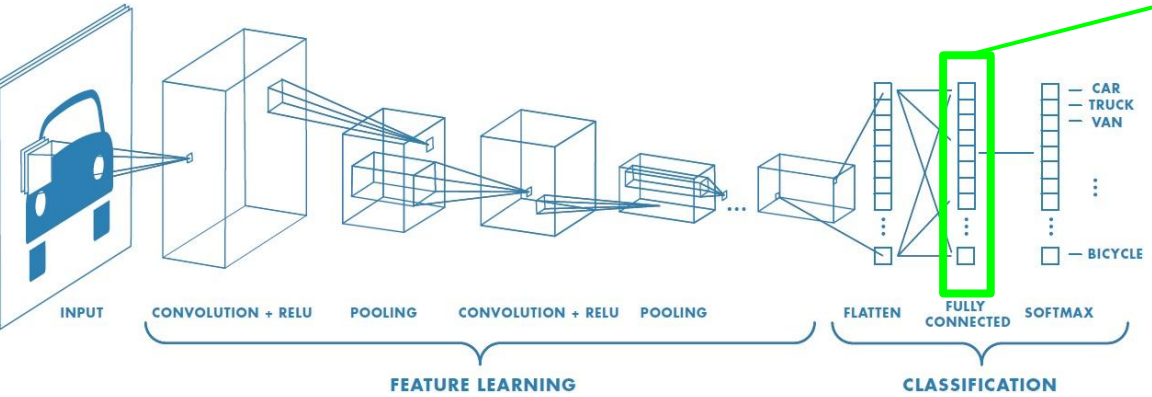
Useful to visualize **invariance** captured by the network.

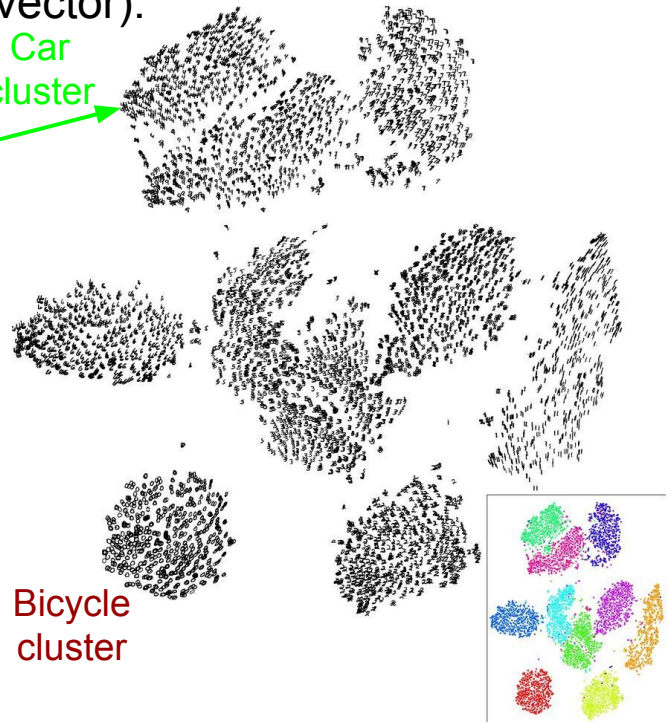
Input **Nearest neighbors in feature space (last layer)**

Krizhevsky et al, “ImageNet Classification with Deep Convolutional Neural Networks”, NIPS 2012.

Last Layer: Dimensionality Reduction

Visualize the space of last layer feature vectors by reducing to **dimensionality two.**
(e.g., 4096 X 1 vector in AlexNet to 2 X 1 vector).





Principle Component Analysis (PCA)

Last Layer: Embedding the codes with t-SNE

Embedding images in 2D so that their low-dimensional representation has approximately equal distances than their high-dimensional representation (of last layer feature, e.g., 4096).

Notice that the similarities are **more often class-based and semantic** rather than pixel and color-based.

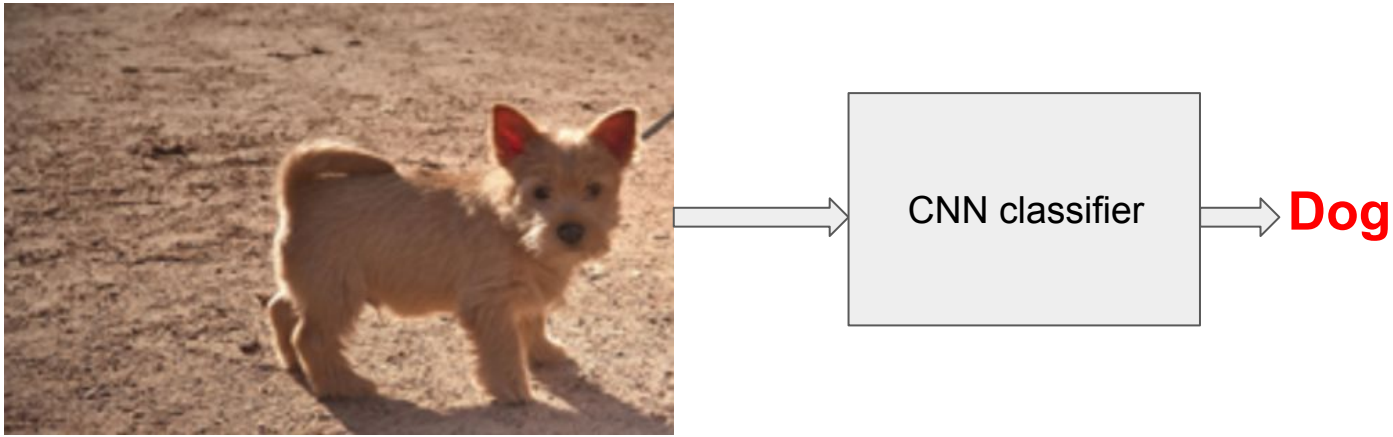
Van der Maaten and Hinton, “Visualizing Data using t-SNE”, JMLR 2008

Topics

Understanding and Visualizing CNN.

1. Visualizing the layer activations or Conv/FC filters
2. Image embedding
3. Saliency or occlusion
4. Activation maximization
5. Feature inversion
6. Back-propagating activations
7. Fooling CNN

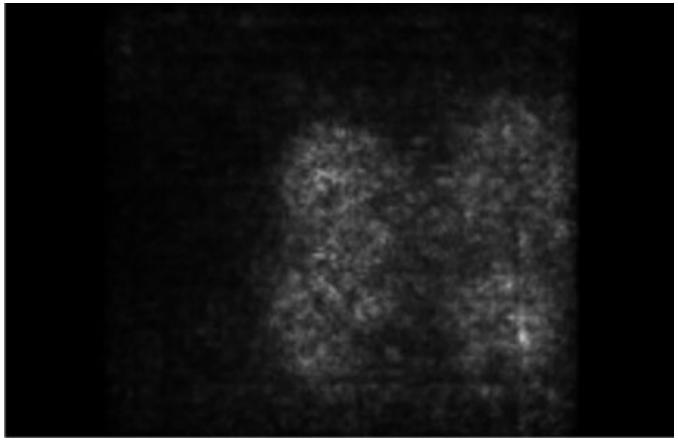
Saliency Maps



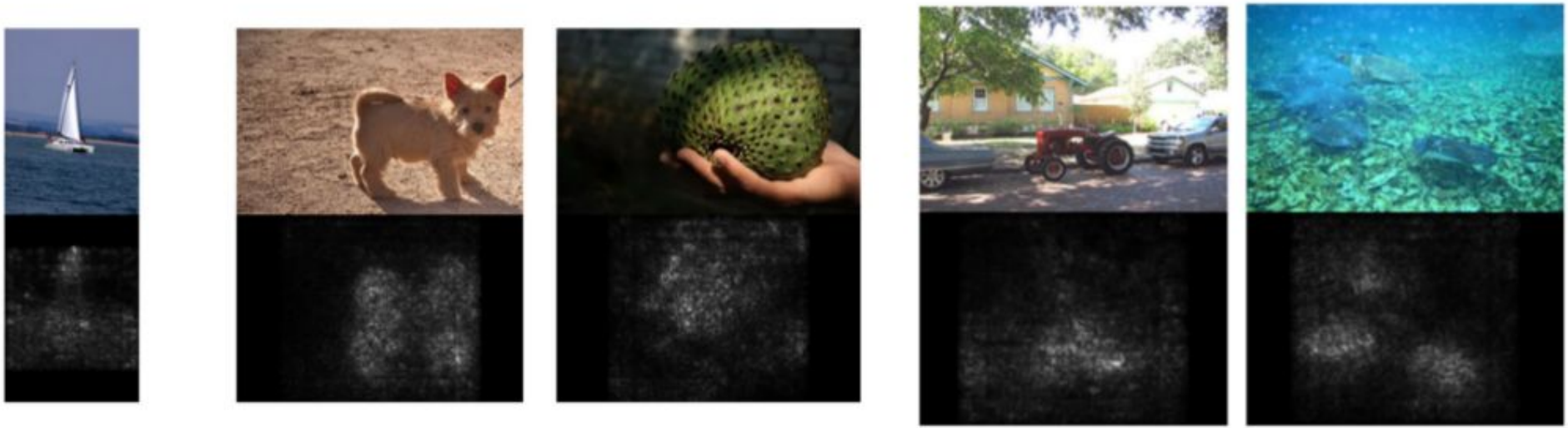
How can we be certain that it's actually picking up on the dog in the image **as opposed to** some contextual cues from the background or some other miscellaneous object?

Saliency Maps

Compute gradient of (unnormalized) class score
with **respect to image pixels**,
take absolute value and
max over RGB channels



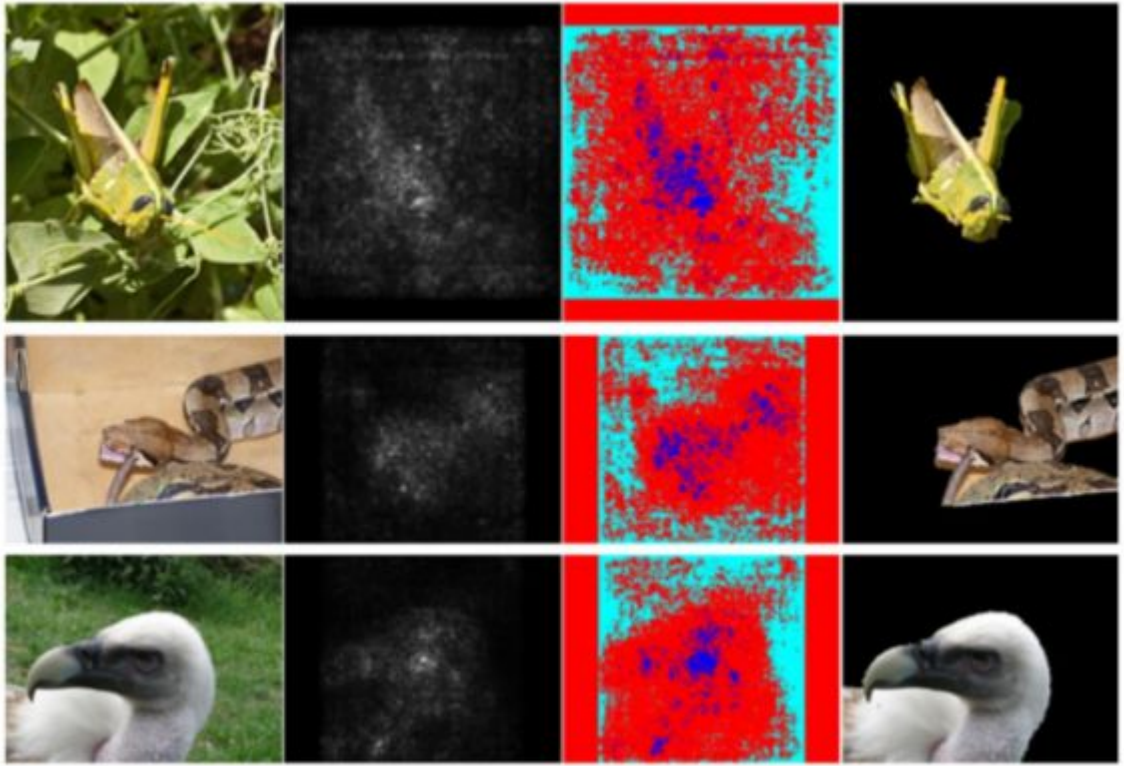
Saliency Maps - Results



Simonyan, Vedaldi, and Zisserman, “Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps”, ICLR Workshop 2014.

Saliency Maps - Segmentation without supervision

Use grabcut on saliency map, using highest intensity as sure foreground (blue) and lowest intensity as sure background (cyan).

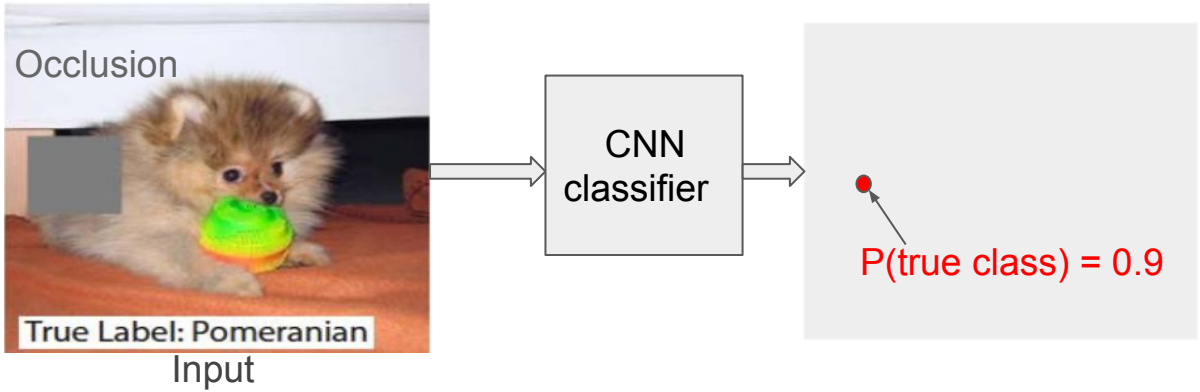


Occlusion experiments

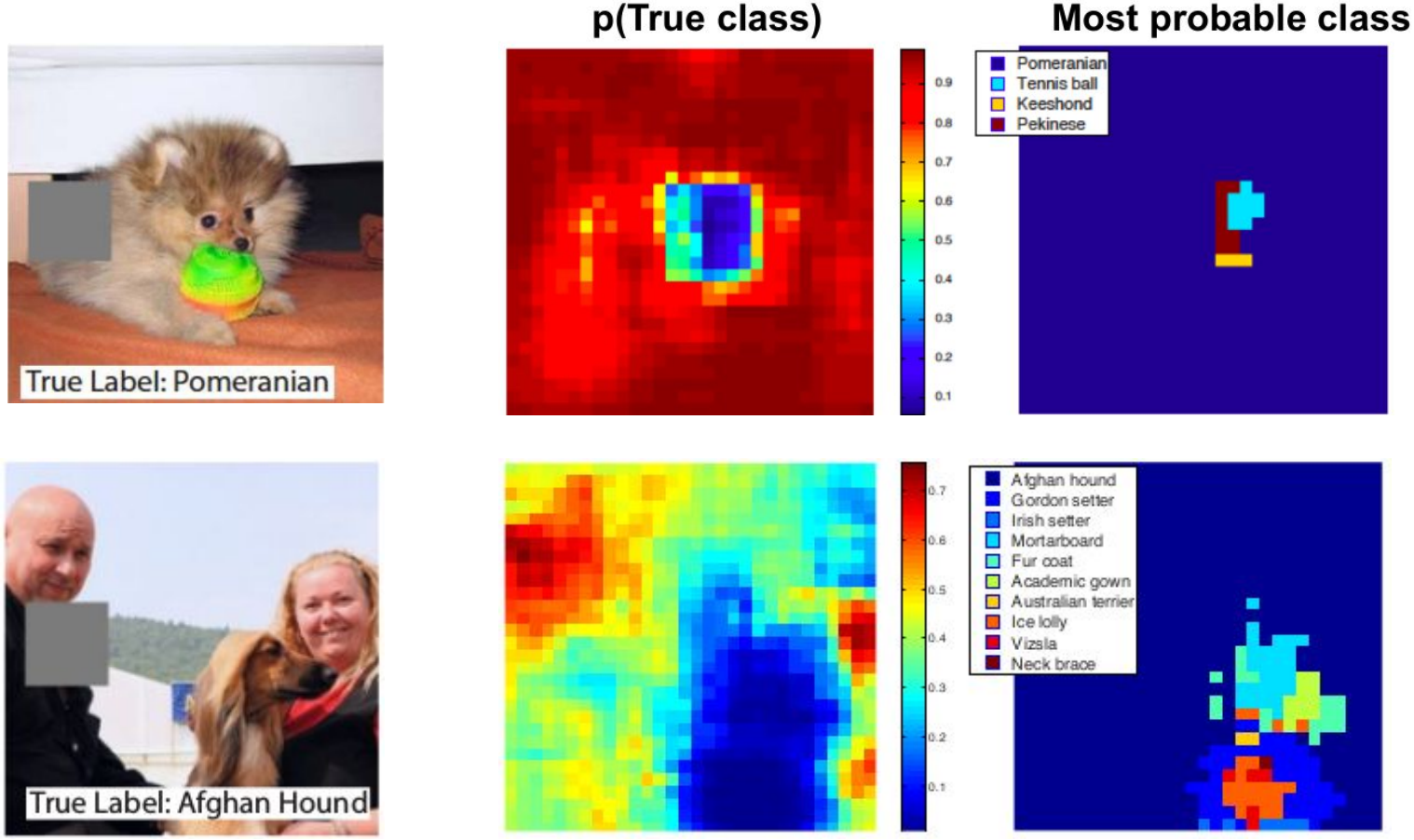
Objective is to study the contribution of different scene features in classification accuracy.

Experiment

1. Mask parts of input with occluding squares.
2. Observe the classification accuracy.



Occlusion experiments (Results)



Topics

Understanding and Visualizing CNN.

1. Visualizing the layer activations or Conv/FC filters
2. Image embedding
3. Saliency or occlusion
4. Activation maximization
5. Feature inversion
6. Back-propagating activations
7. Fooling CNN

Activation maximization

Objective: Generate **a** synthetic image that maximally activates **a neuron**.

$$I^* = \arg\max_I \mathbf{f}(I) + \mathbf{R}(I)$$

Neuron value (scalar)

Natural image prior

Drawback:

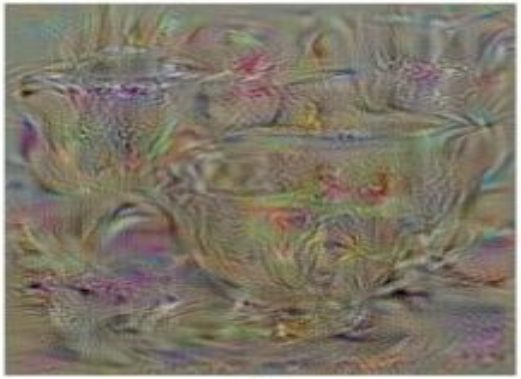
1. Computationally expensive (since require high dimensional optimization).
2. Invariance learnt by neuron cannot be obtained (since only one image per node).

Simonyan, Vedaldi, and Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014.

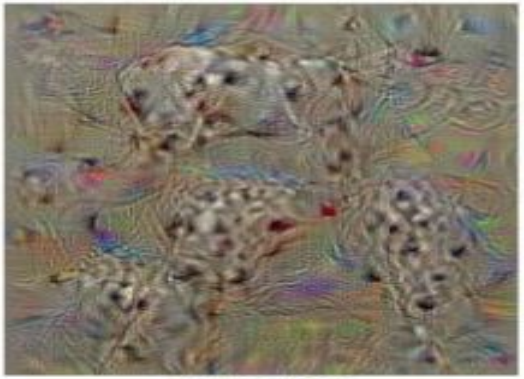
Activation maximization - Results for neuron before Softmax

(with minimum l2 norm as image prior) $$I^* = \arg\max_I \mathbf{S_c}(I) + \|I\|_2^2$$

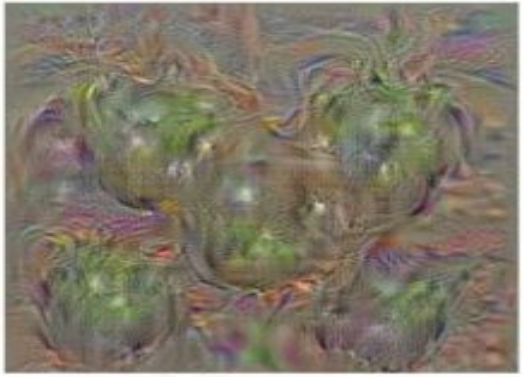
dumbbell



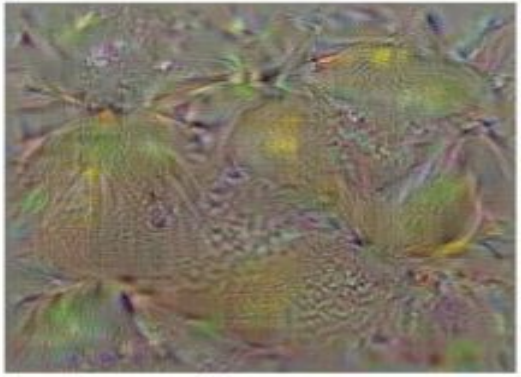
cup



dalmatian



bell pepper

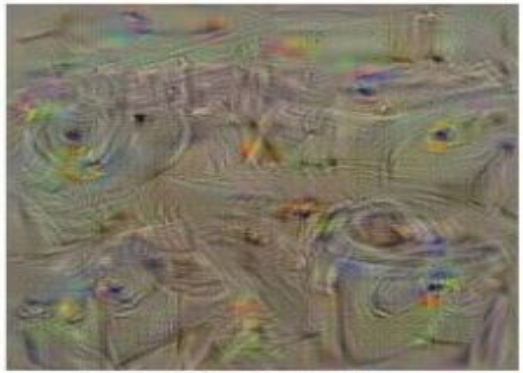


lemon

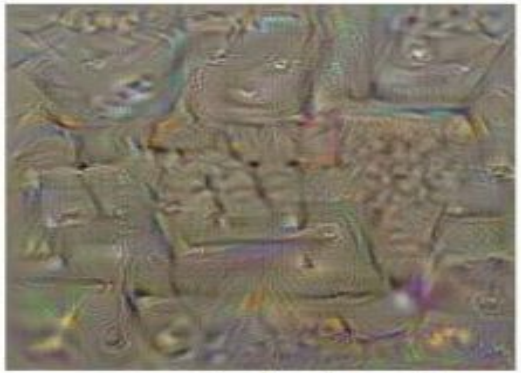
husky

Activation maximization - Results for neuron before Softmax

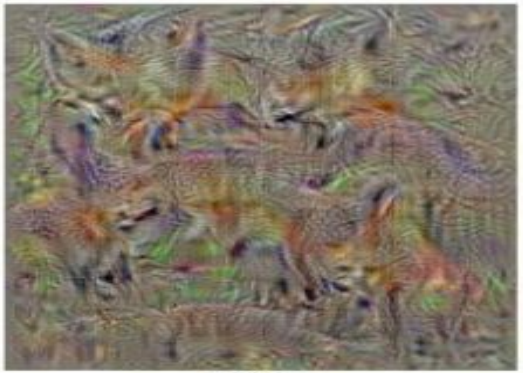
(with minimum l2 norm as image prior) $$I^* = \arg\max_I \mathbf{S_c}(I) + \|I\|_2^2$$



washing machine



computer keyboard



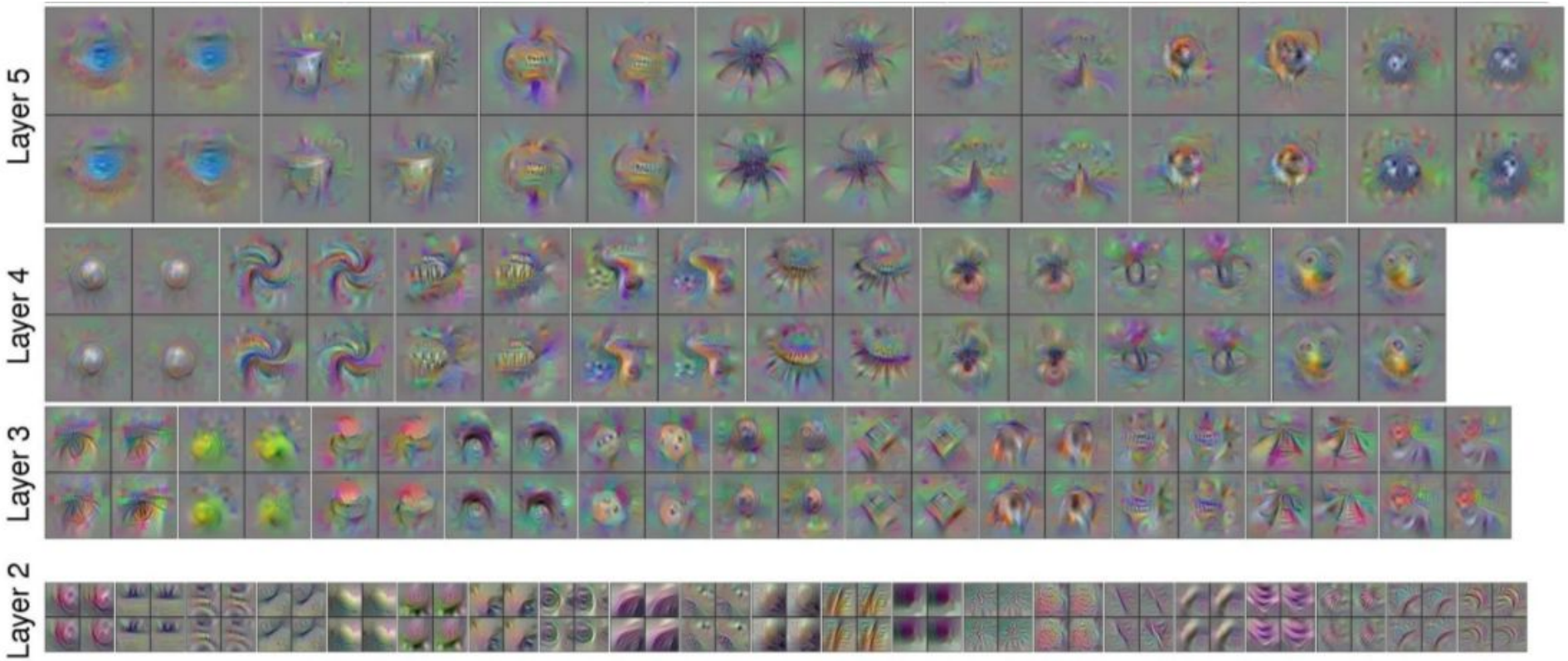
kit fox

goose

ostrich

limousine

Activation maximization - Results for intermediate neurons.



Yosinski et al, "Understanding Neural Networks Through Deep Visualization", ICML DL Workshop 2014.

Topics

Understanding and Visualizing CNN.

1. Visualizing the layer activations or Conv/FC filters
2. Image embedding
3. Saliency or occlusion
4. Activation maximization
5. Feature inversion
6. Back-propagating activations
7. Fooling CNN

Feature inversion

Given a **CNN feature vector** for an image, find a new image that:

1. Matches the given feature vector
2. “looks natural” (image prior regularization)

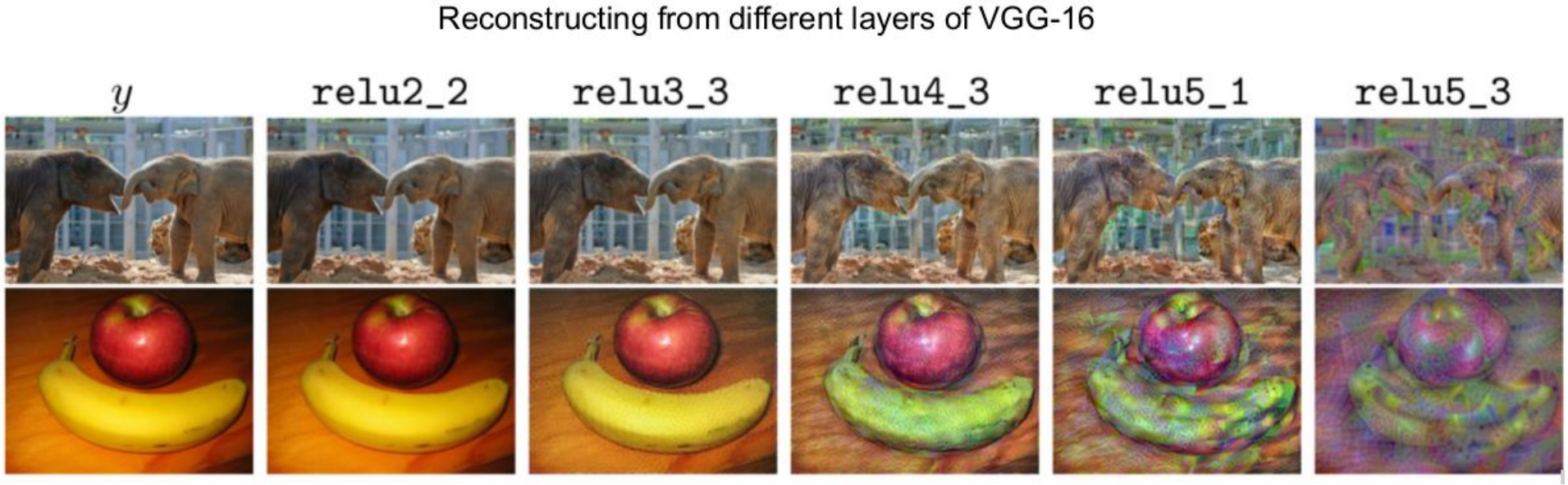
$$I^* = \arg\min_I \|\Phi(I) - \Phi_0\|^2 + \lambda \cdot \mathrm{TV}(I)$$

Features of image I

Given feature vector

Total variation prior

Feature inversion - Results



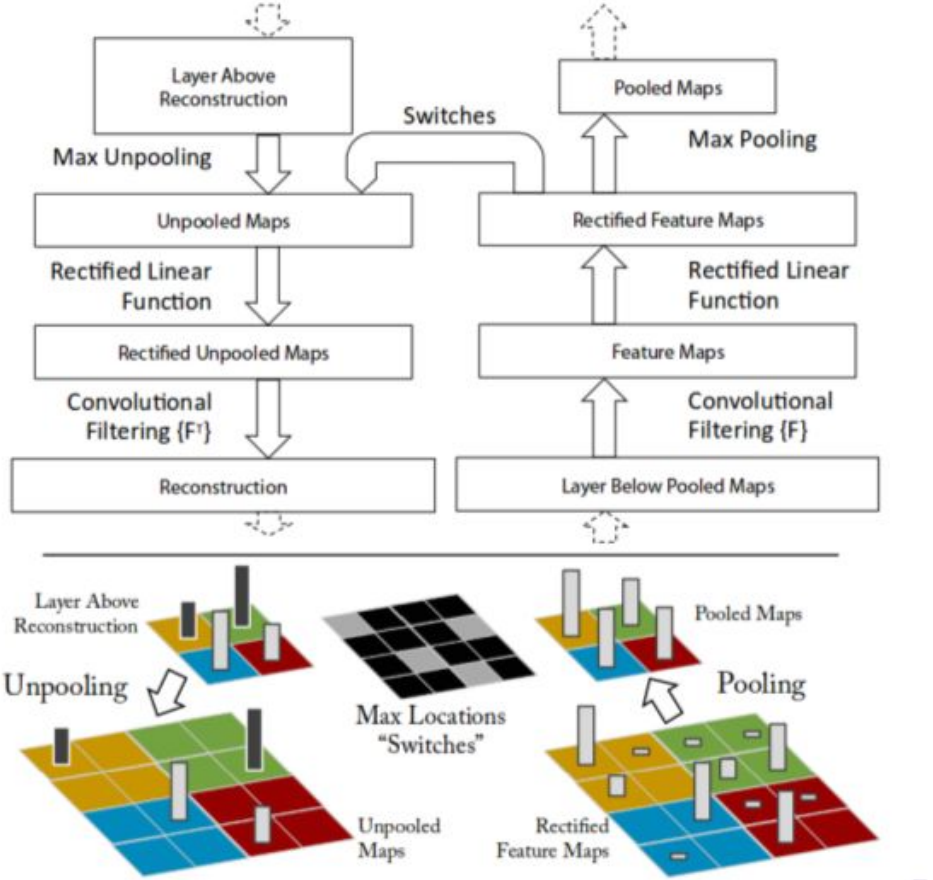
Mahendran and Vedaldi, "Understanding Deep Image Representations by Inverting Them", CVPR 2015

Topics

Understanding and Visualizing CNN.

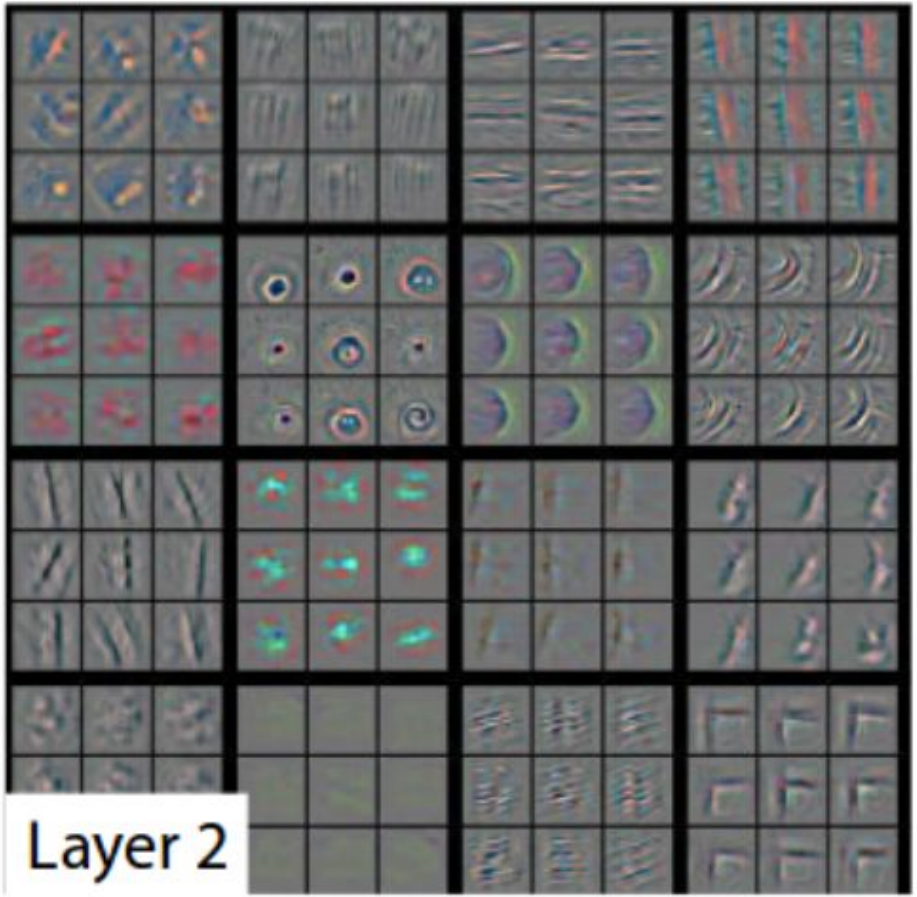
1. Visualizing the layer activations or Conv/FC filters
2. Image embedding
3. Saliency or occlusion
4. Activation maximization
5. Feature inversion
6. Back-propagating activations
7. Fooling CNN

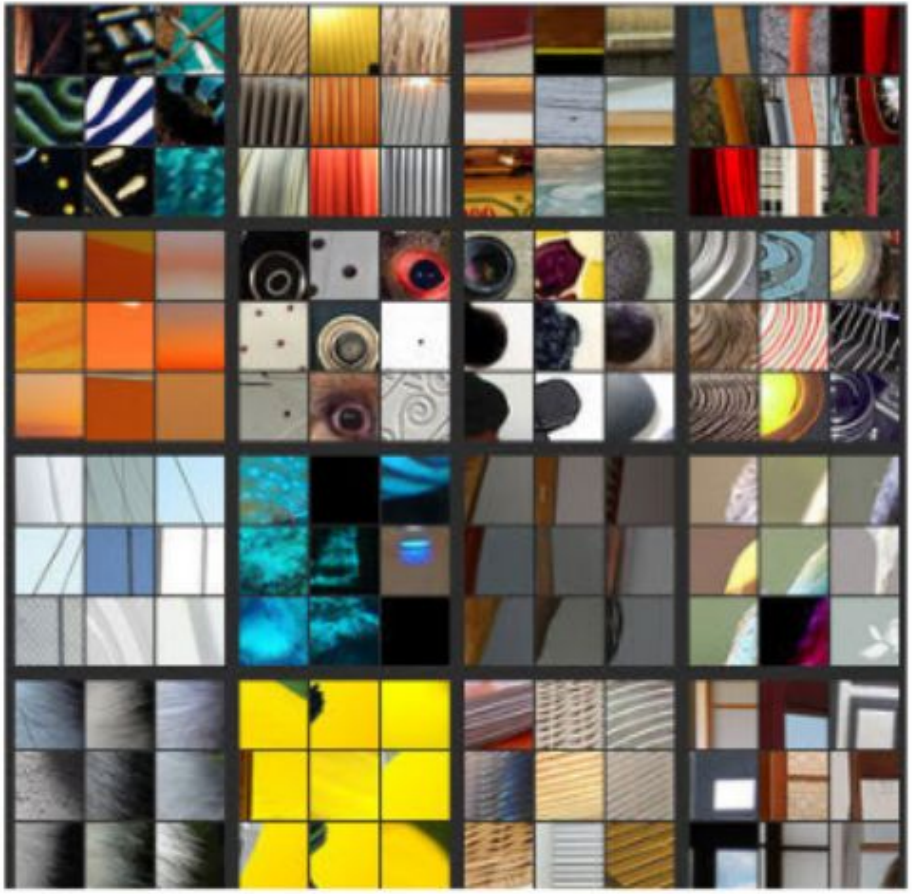
How can we invert max pooling?



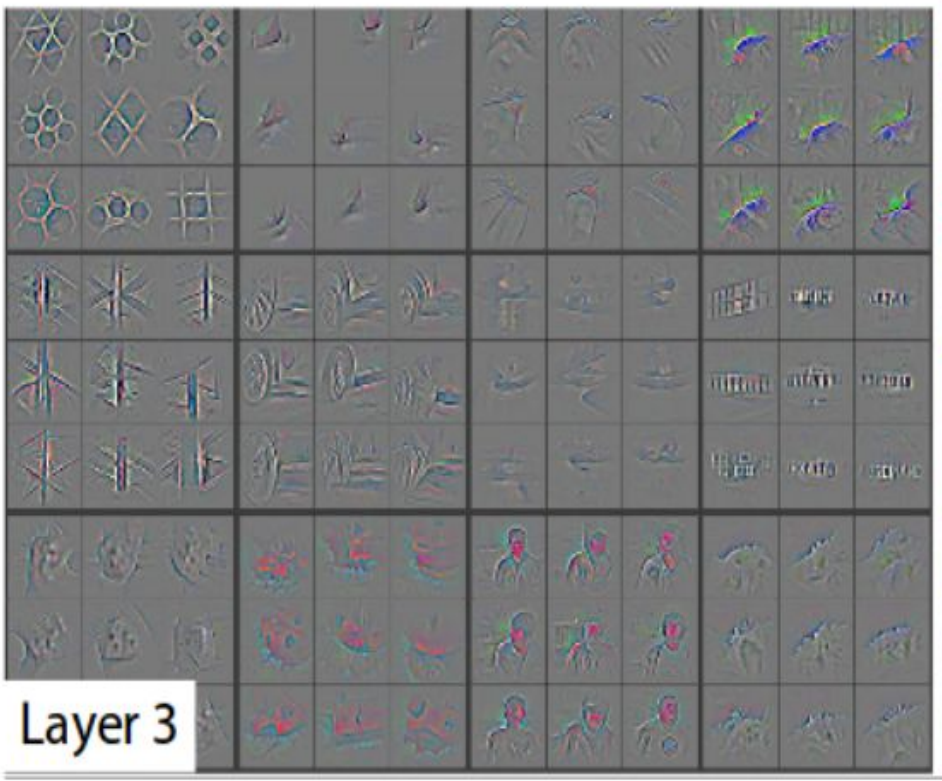
Zeiler and Fergus, "Visualizing and Understanding Convolutional Networks", ECCV 2014

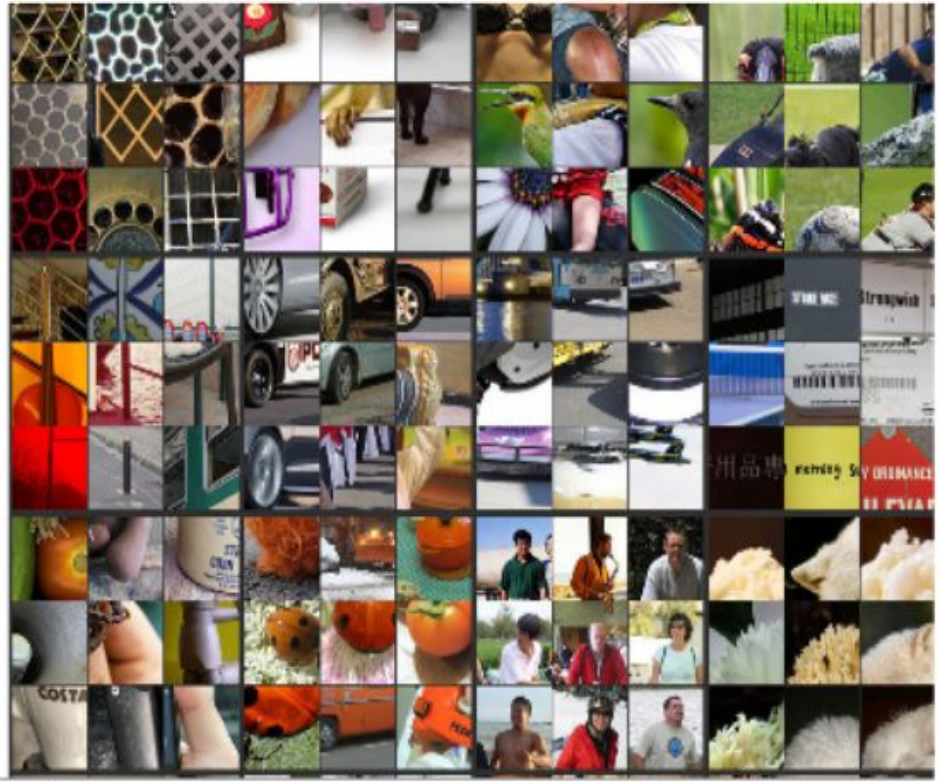
Layer 2 features.



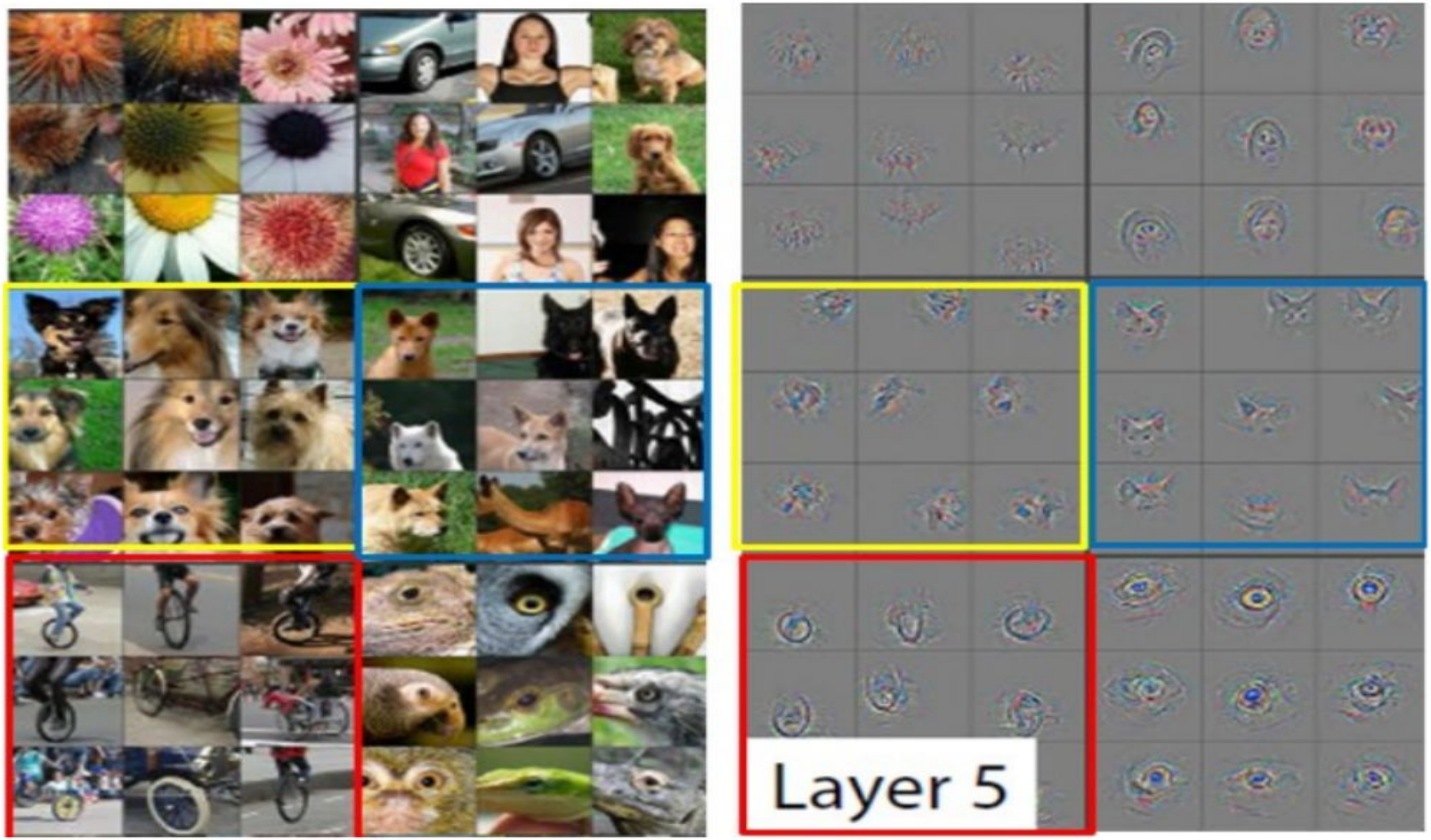


Layer 3 features.





Layer 5 features.



Topics

Understanding and Visualizing CNN.

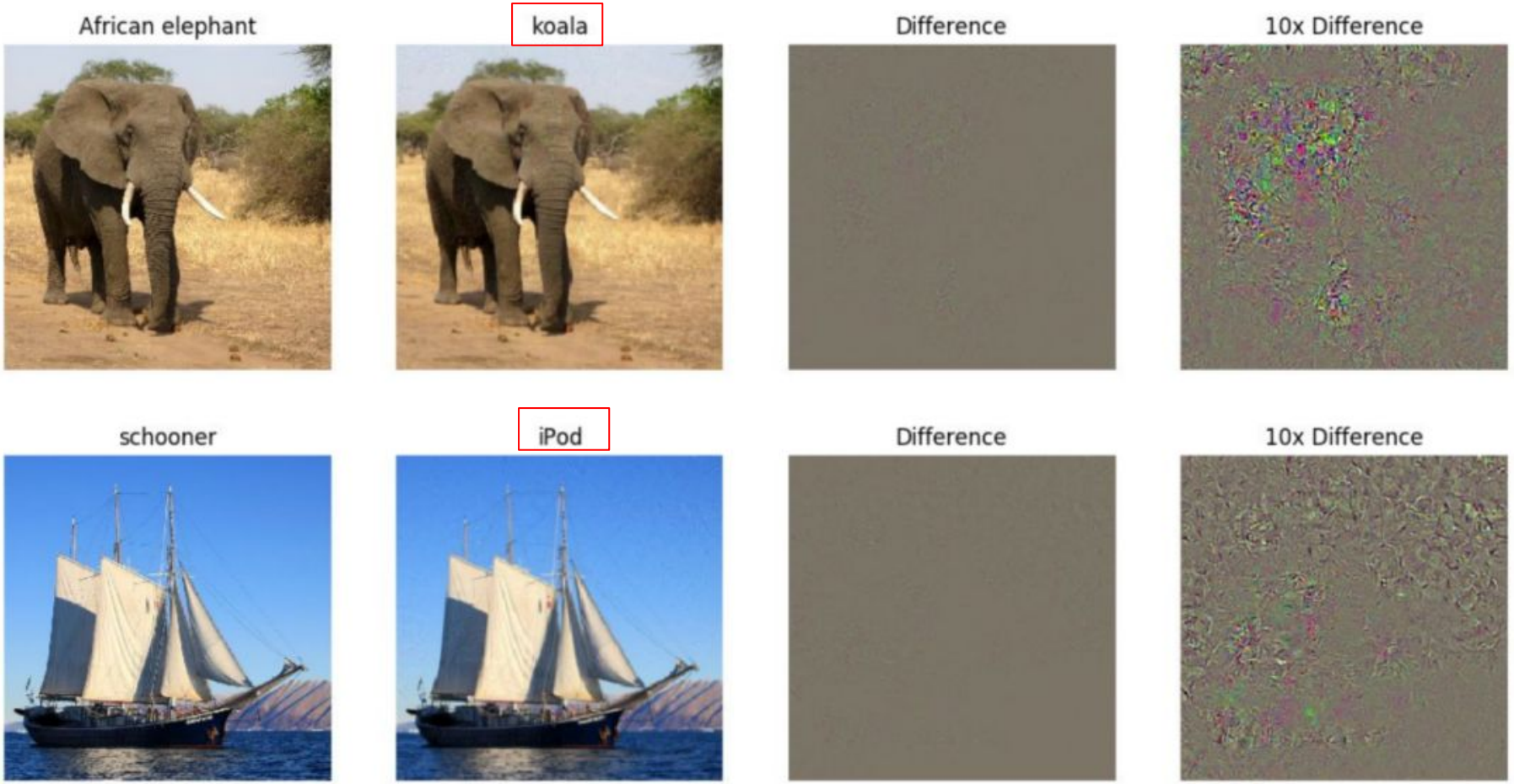
1. Visualizing the layer activations or Conv/FC filters
2. Image embedding
3. Saliency or occlusion
4. Activation maximization
5. Feature inversion
6. Back-propagating activations
7. Fooling CNN

Fooling CNN

Fooling Images / Adversarial examples

1. Start from an arbitrary image.
2. Pick an arbitrary class.
3. Modify the **image** to maximize the class (Remember for training we modify the weights).
4. Repeat until network is fooled.

Fooling CNN - Results



CNN Architectures

Slides taken from CS231n lecture slides

Today: CNN Architectures

Case Studies

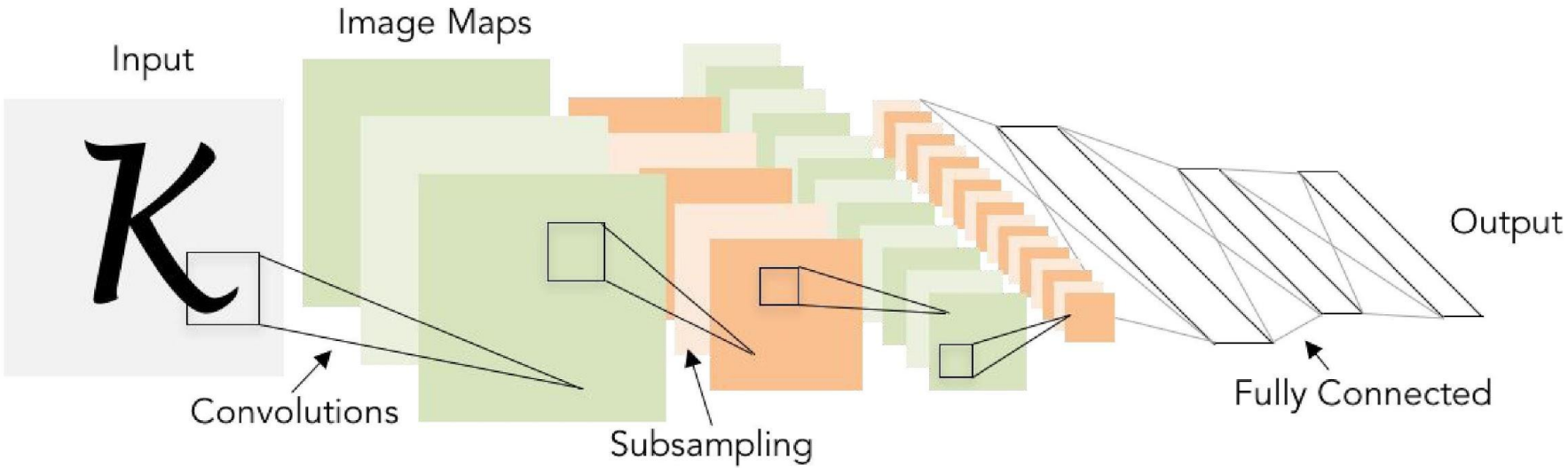
- AlexNet
- VGG
- GoogLeNet
- ResNet

Also....

- NiN (Network in Network)
- Wide ResNet
- ResNeXT
- Stochastic Depth
- DenseNet
- FractalNet
- SqueezeNet

Review: LeNet-5

[LeCun et al., 1998]

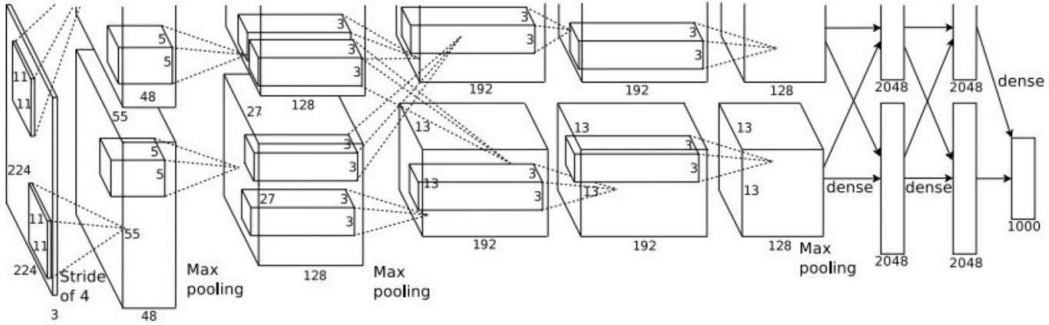


Conv filters were 5x5, applied at stride 1
Subsampling (Pooling) layers were 2x2 applied at stride 2
i.e. architecture is [CONV-POOL-CONV-POOL-FC-FC]

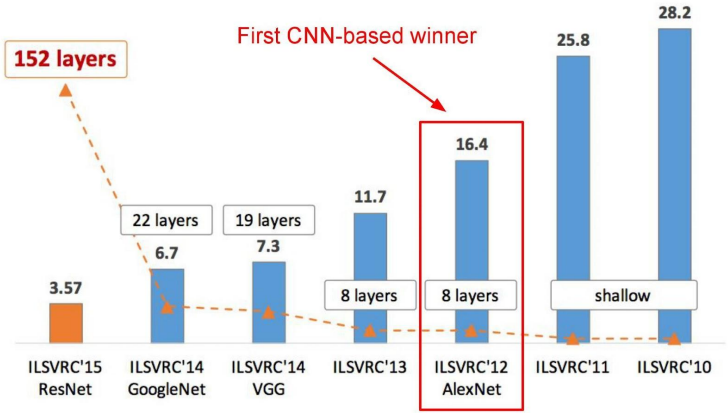
Case Study: AlexNet

[Krizhevsky et al. 2012]

Architecture:
CONV1
MAX POOL1
NORM1
CONV2
MAX POOL2
NORM2
CONV3
CONV4
CONV5
Max POOL3
FC6
FC7
FC8

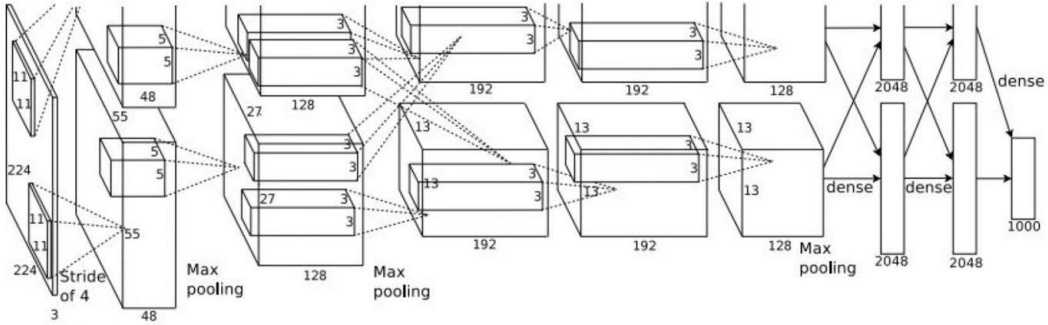


ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Case Study: AlexNet

[Krizhevsky et al. 2012]

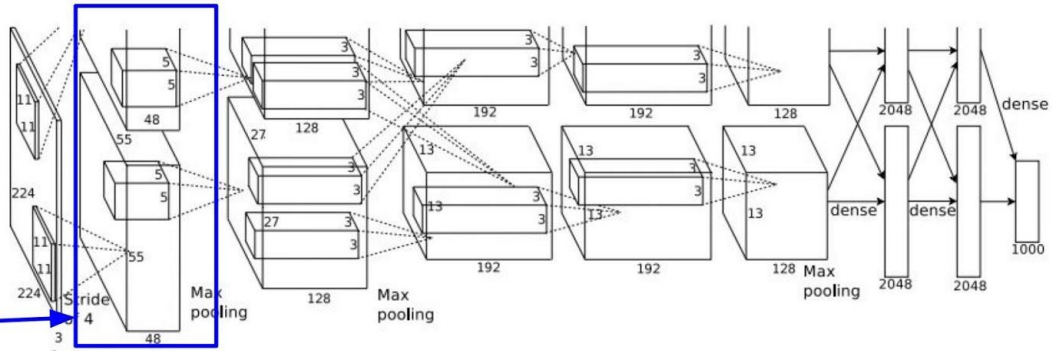


Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

Details/Retrospectives:
- first use of ReLU
- used Norm layers (not common anymore)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%

Case Study: AlexNet

[Krizhevsky et al. 2012]



[55x55x48] x 2

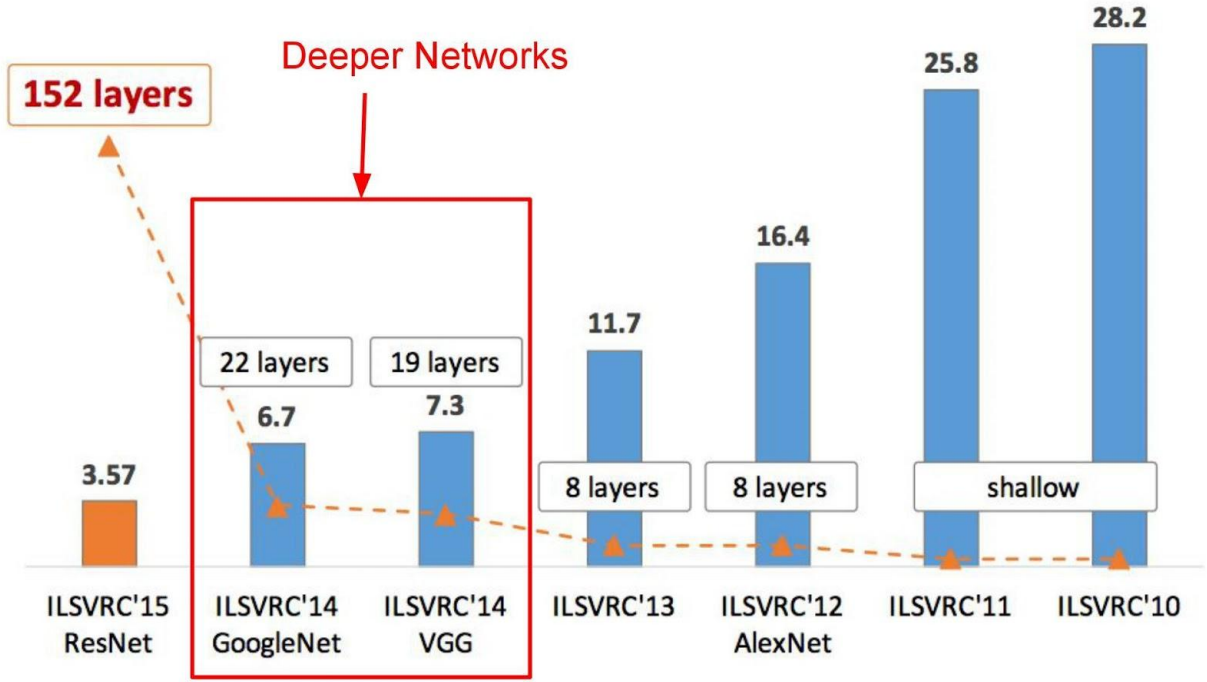
Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

Historical note: Trained on GTX 580 GPU with only 3 GB of memory. Network spread across 2 GPUs, half the neurons (feature maps) on each GPU.

Note on Alexnet

- Notice the use of filters of size 11x11 in the initial layer.
- It increases the number of parameters that need to be trained

ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners



Case Study: VGGNet

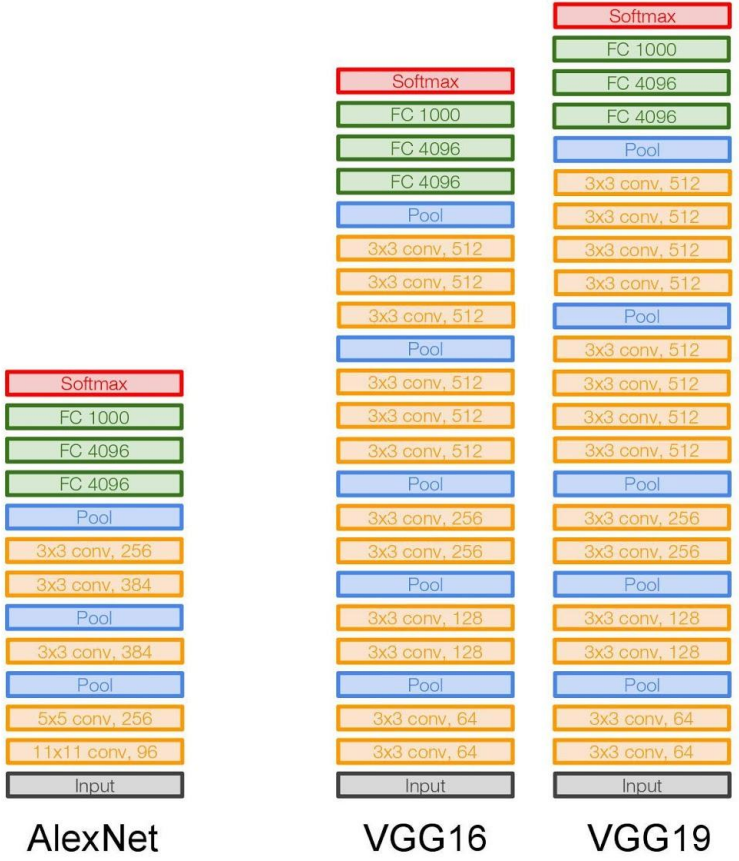
[Simonyan and Zisserman, 2014]

Small filters, Deeper networks

8 layers (AlexNet)
-> 16 - 19 layers (VGG16Net)

Only 3x3 CONV stride 1, pad 1
and 2x2 MAX POOL stride 2

11.7% top 5 error in ILSVRC'13
(ZFNet)
-> 7.3% top 5 error in ILSVRC'14



Exercise

- Alexnet uses 11x11 filters in the first layer which has a receptive field of 11x11. How many layers of filter size 3x3 are needed to achieve the same receptive field. (Assume no pooling layers)
- Is there any reduction in the number of parameters to be trained?

Case Study: VGGNet

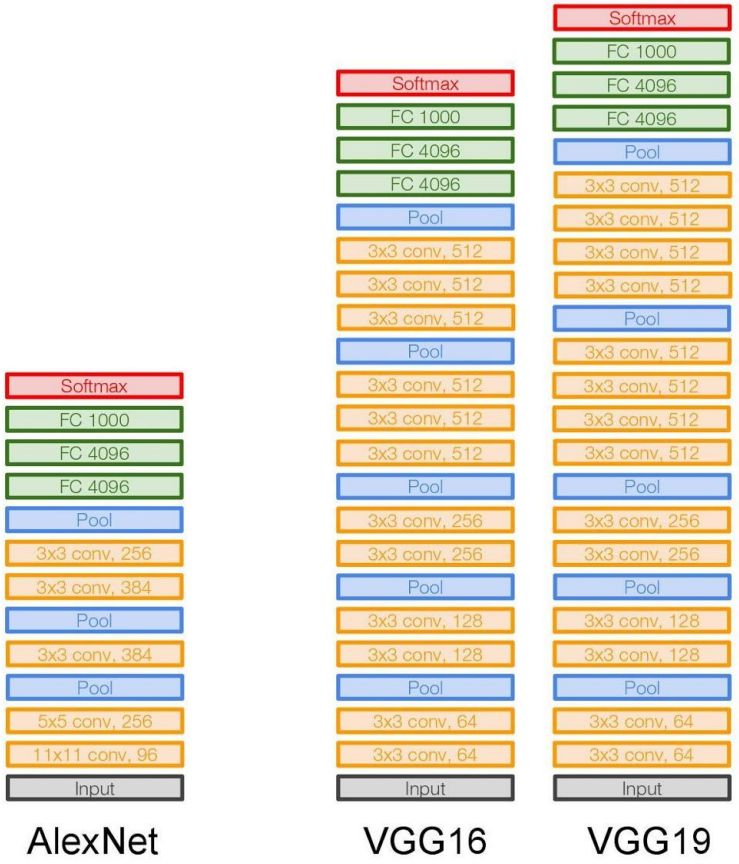
[Simonyan and Zisserman, 2014]

Q: Why use smaller filters? (3x3 conv)

Stack of three 3x3 conv (stride 1) layers has same **effective receptive field** as one 7x7 conv layer

But deeper, more non-linearities

And fewer parameters: $3 * (3^2C^2)$ vs. 7^2C^2 for C channels per layer



(not counting biases)

INPUT: [224x224x3] memory: 224*224*3=150K params: 0
CONV3-64: [224x224x64] memory: **224*224*64=3.2M** params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: **224*224*64=3.2M** params: (3*3*64)*64 = 36,864
POOL2: [112x112x64] memory: 112*112*64=800K params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M params: (3*3*128)*128 = 147,456
POOL2: [56x56x128] memory: 56*56*128=400K params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K params: (3*3*256)*256 = 589,824
POOL2: [28x28x256] memory: 28*28*256=200K params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512] memory: 14*14*512=100K params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512] memory: 7*7*512=25K params: 0
FC: [1x1x4096] memory: 4096 params: 7*7*512*4096 = **102,760,448**
FC: [1x1x4096] memory: 4096 params: 4096*4096 = 16,777,216
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

Note:

Most memory is in early CONV

Most params are in late FC

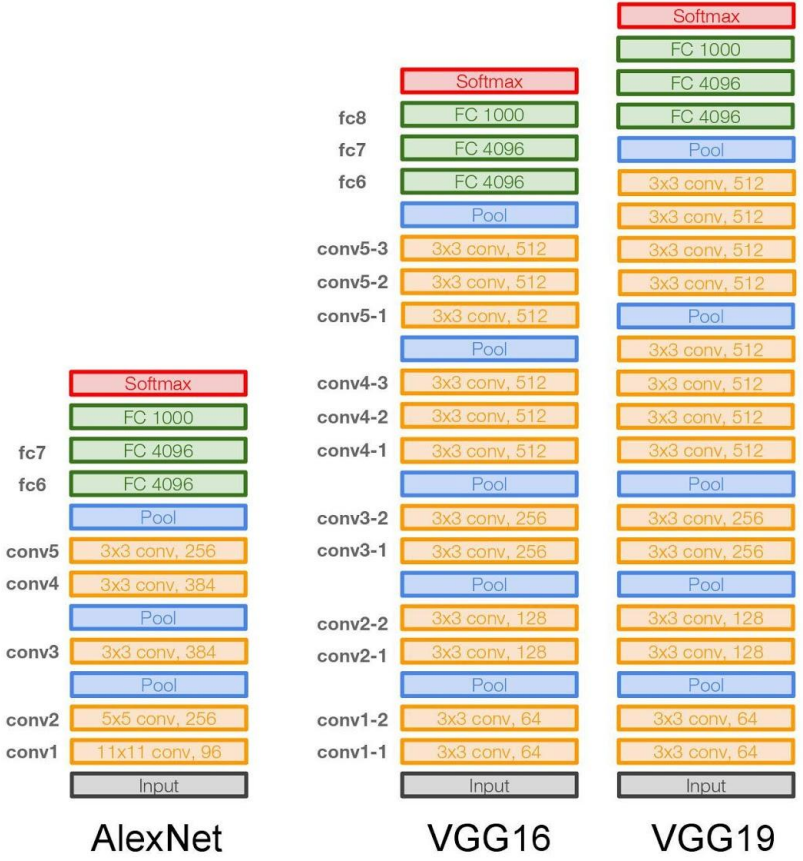
TOTAL memory: 24M * 4 bytes ~= 96MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters

Case Study: VGGNet

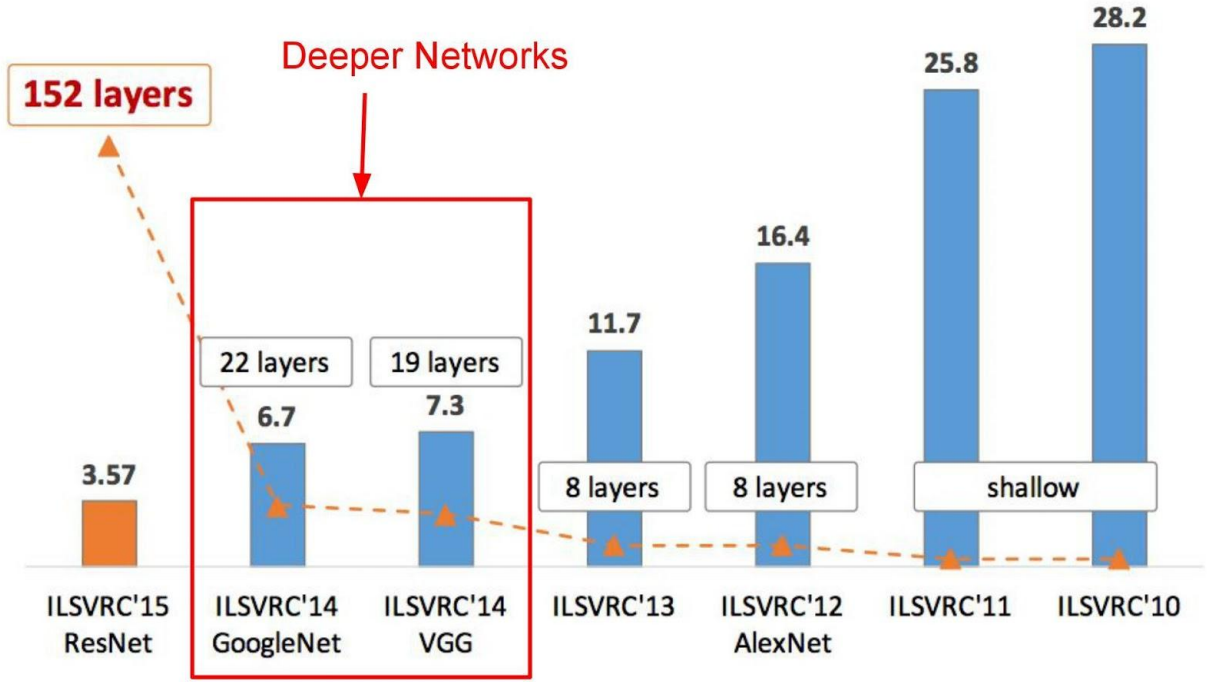
[Simonyan and Zisserman, 2014]

Details:

- ILSVRC'14 2nd in classification, 1st in localization
- Similar training procedure as Krizhevsky 2012
- No Local Response Normalisation (LRN)
- Use VGG16 or VGG19 (VGG19 only slightly better, more memory)
- Use ensembles for best results
- FC7 features generalize well to other tasks



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

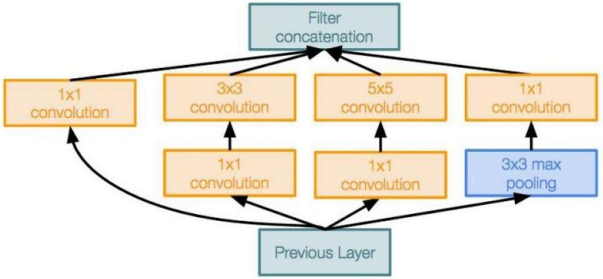


Case Study: GoogLeNet

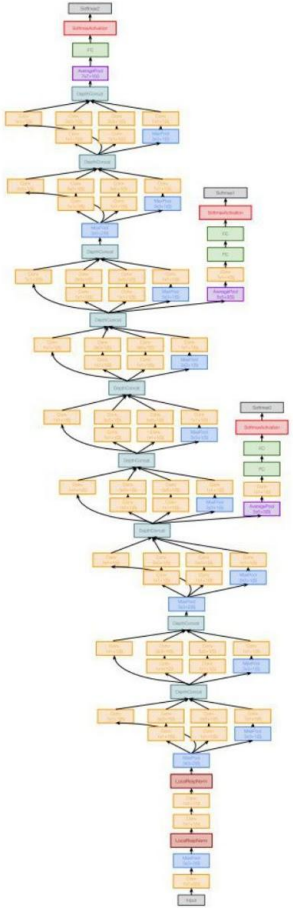
[Szegedy et al., 2014]

Deeper networks, with computational efficiency

- 22 layers
- Efficient “Inception” module
- No FC layers
- Only 5 million parameters!
  12x less than AlexNet
- ILSVRC’14 classification winner
  (6.7% top 5 error)



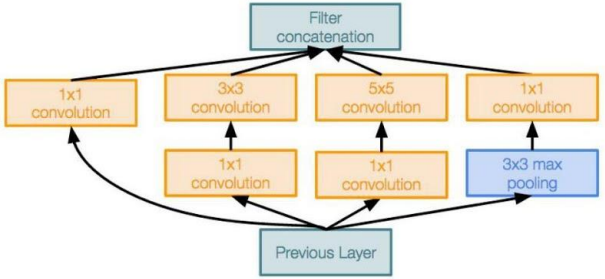
Inception module



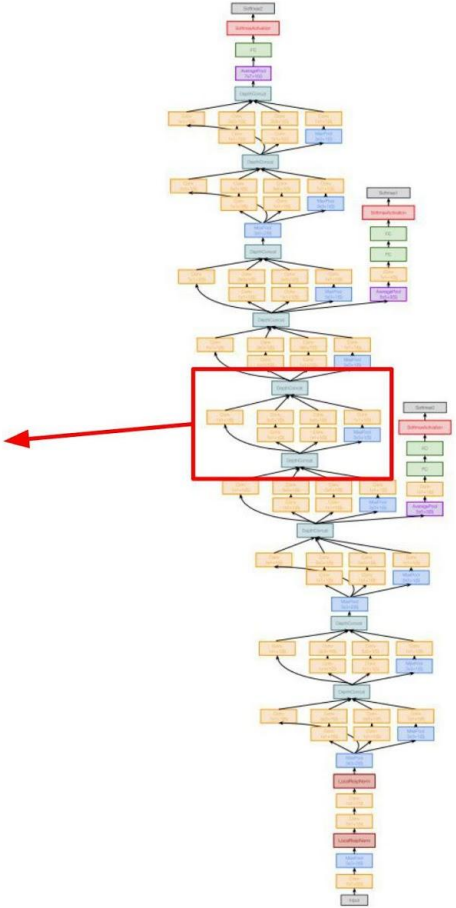
Case Study: GoogLeNet

[Szegedy et al., 2014]

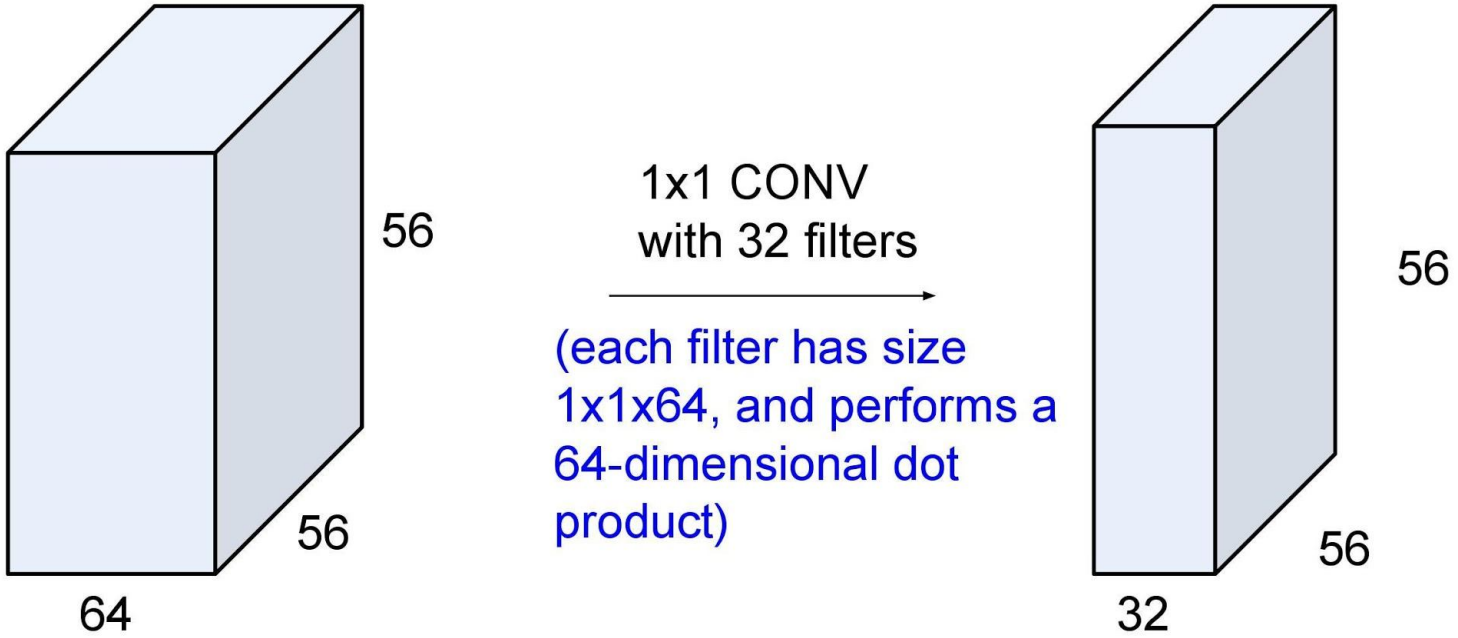
"Inception module": design a good local network topology (network within a network) and then stack these modules on top of each other



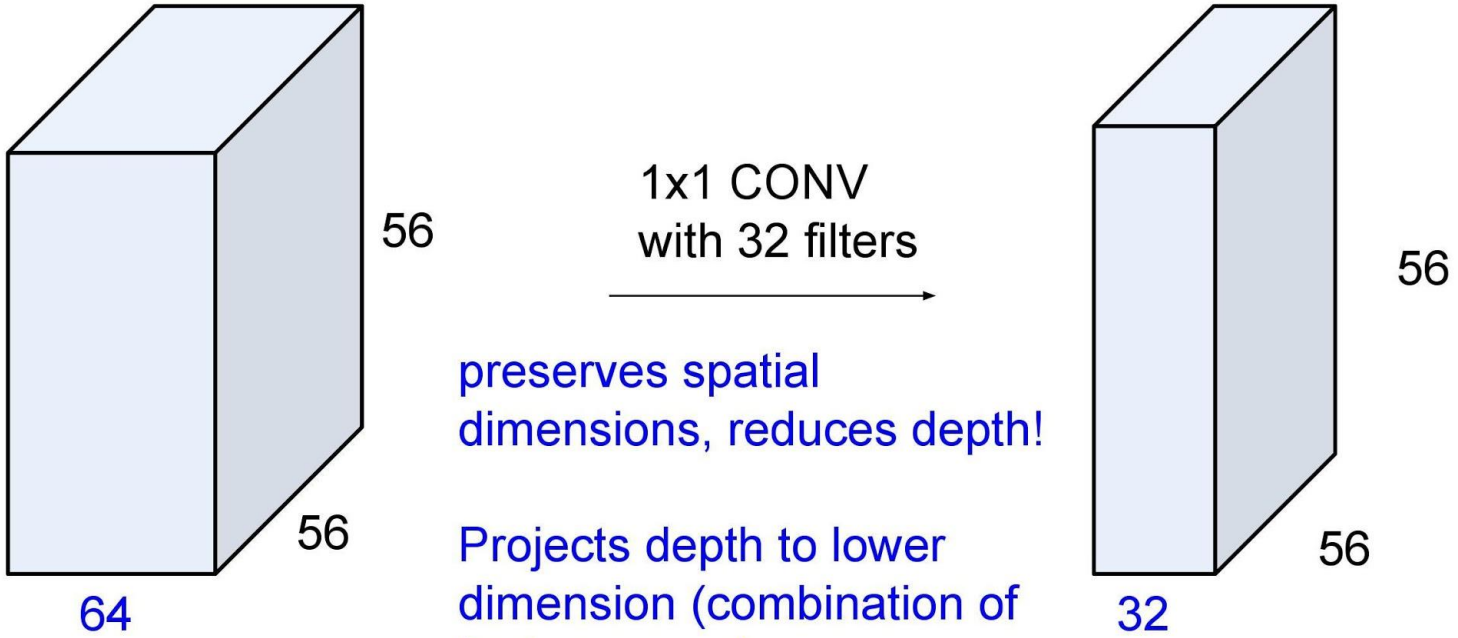
Inception module



1x1 Convolutions

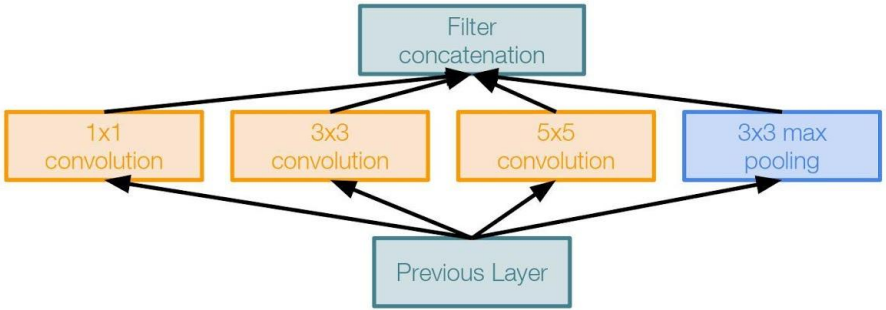


1x1 Convolutions



Case Study: GoogLeNet

[Szegedy et al., 2014]



Naive Inception module

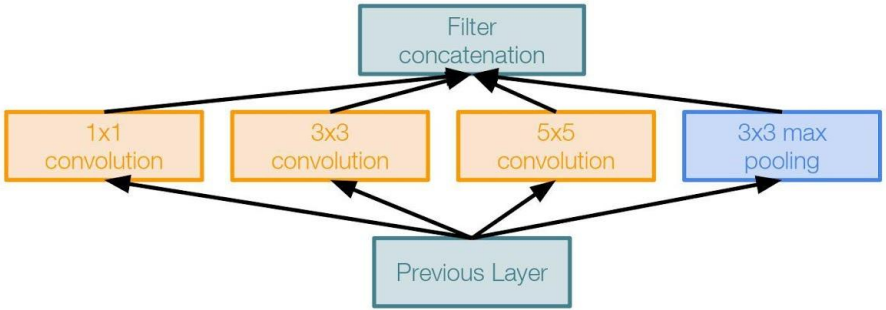
Apply parallel filter operations on the input from previous layer:

- Multiple receptive field sizes for convolution (1x1, 3x3, 5x5)
- Pooling operation (3x3)

Concatenate all filter outputs together depth-wise

Case Study: GoogLeNet

[Szegedy et al., 2014]



Naive Inception module

Apply parallel filter operations on the input from previous layer:

- Multiple receptive field sizes for convolution (1x1, 3x3, 5x5)
- Pooling operation (3x3)

Concatenate all filter outputs together depth-wise

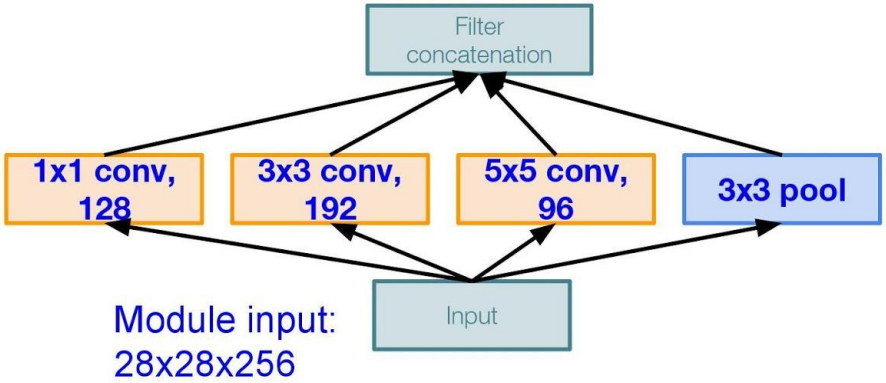
Q: What is the problem with this?
[Hint: Computational complexity]

Case Study: GoogLeNet

[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:



Naive Inception module

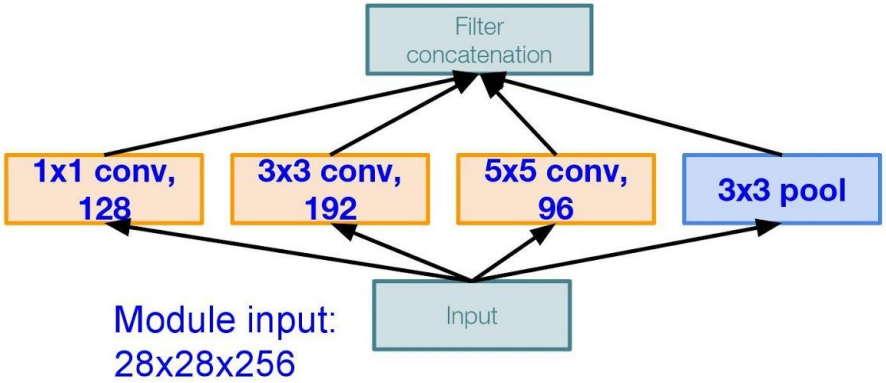
Case Study: GoogLeNet

[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q1: What is the output size of the 1x1 conv, with 128 filters?



Naive Inception module

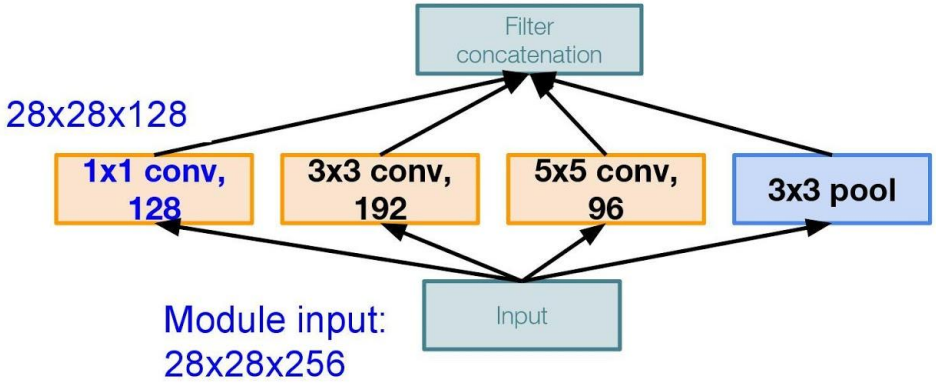
Case Study: GoogLeNet

[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q1: What is the output size of the 1x1 conv, with 128 filters?



Naive Inception module

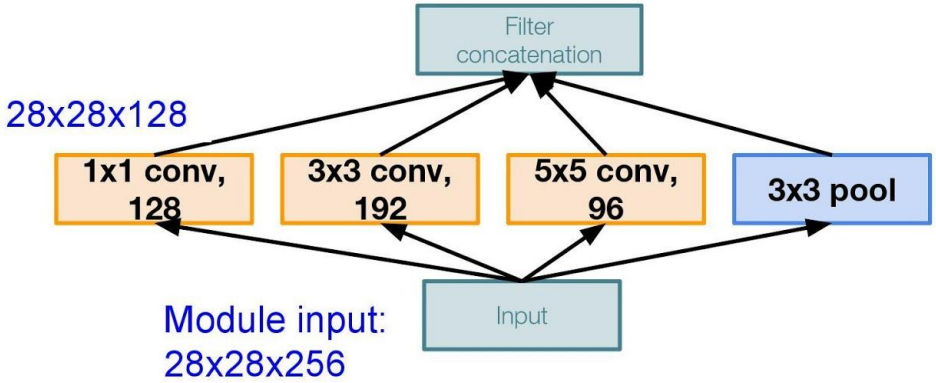
Case Study: GoogLeNet

[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q2: What are the output sizes of all different filter operations?



Naive Inception module

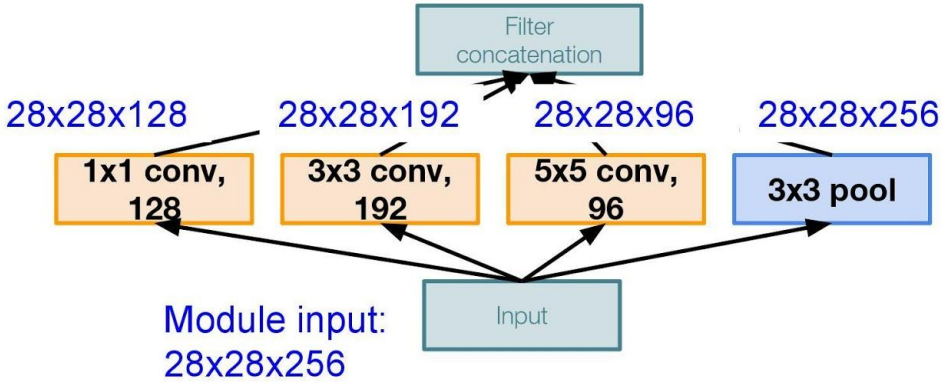
Case Study: GoogLeNet

[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q2: What are the output sizes of all different filter operations?



Naive Inception module

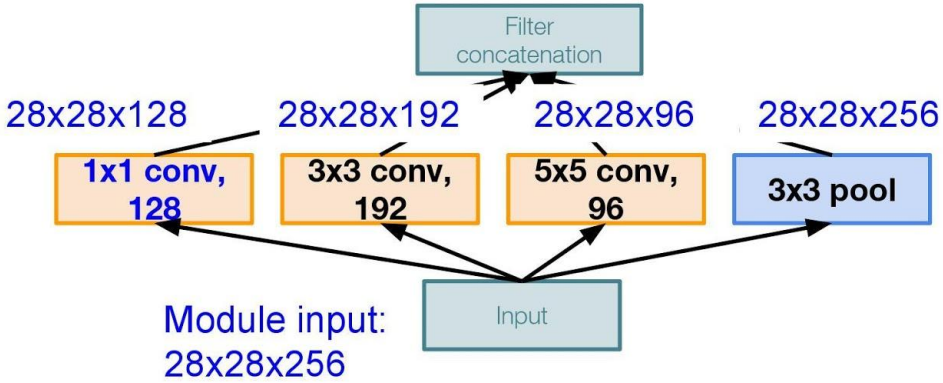
Case Study: GoogLeNet

[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q3:What is output size after filter concatenation?



Naive Inception module

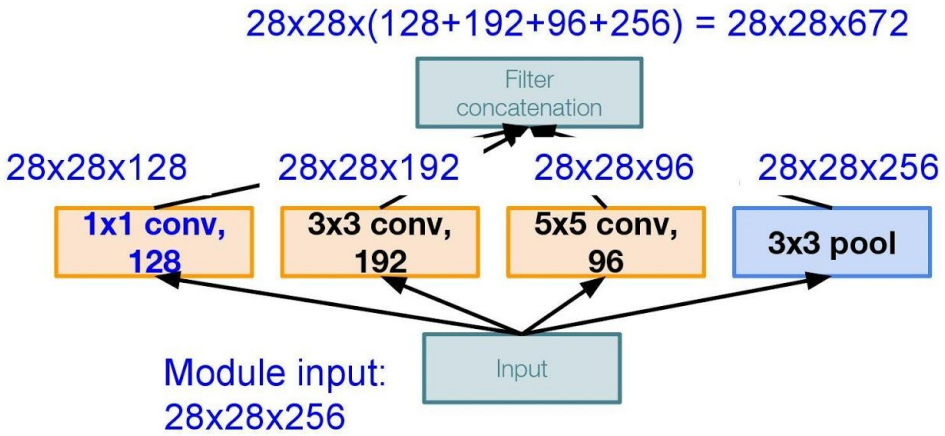
Case Study: GoogLeNet

[Szegedy et al., 2014]

Q: What is the problem with this?
[Hint: Computational complexity]

Example:

Q3:What is output size after filter concatenation?



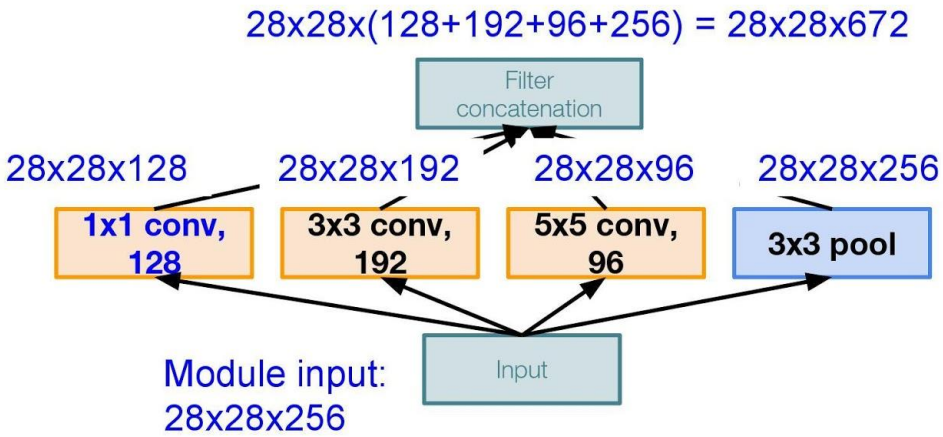
Naive Inception module

Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q3:What is output size after filter concatenation?



Naive Inception module

Q: What is the problem with this?
[Hint: Computational complexity]

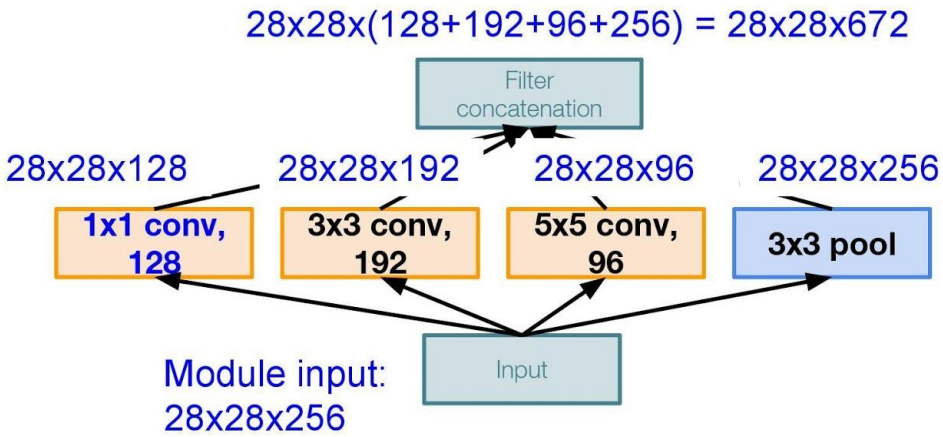
Conv Ops:
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x256
[5x5 conv, 96] 28x28x96x5x5x256
Total: 854M ops

Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q3:What is output size after filter concatenation?



Naive Inception module

Q: What is the problem with this?
[Hint: Computational complexity]

Conv Ops:
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x256
[5x5 conv, 96] 28x28x96x5x5x256
Total: 854M ops

Very expensive compute

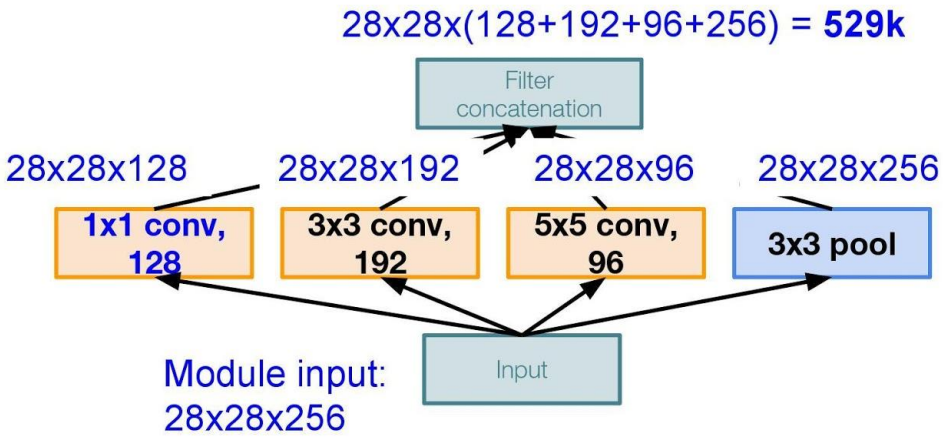
Pooling layer also preserves feature depth, which means total depth after concatenation can only grow at every layer!

Case Study: GoogLeNet

[Szegedy et al., 2014]

Example:

Q3:What is output size after filter concatenation?



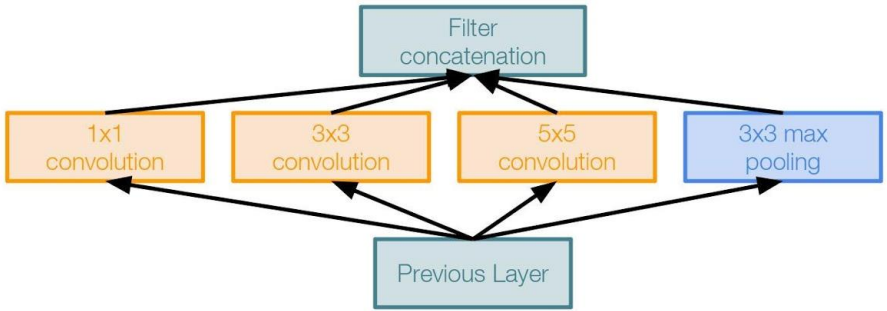
Naive Inception module

Q: What is the problem with this? [Hint: Computational complexity]

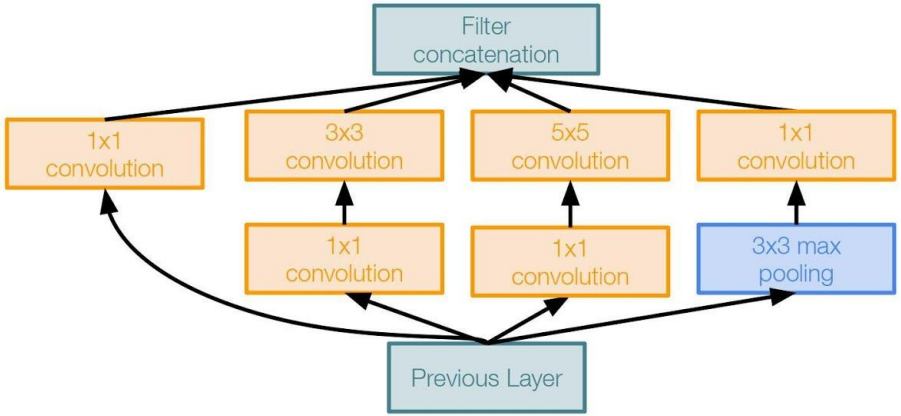
Solution: “bottleneck” layers that use 1x1 convolutions to reduce feature depth

Case Study: GoogLeNet

[Szegedy et al., 2014]



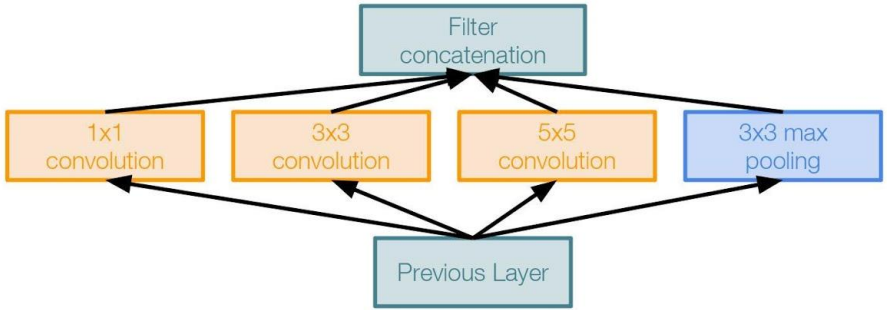
Naive Inception module



Inception module with dimension reduction

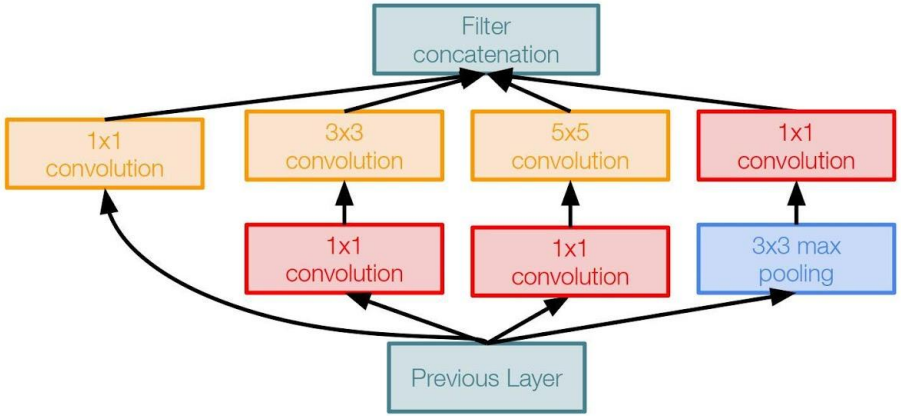
Case Study: GoogLeNet

[Szegedy et al., 2014]



Naive Inception module

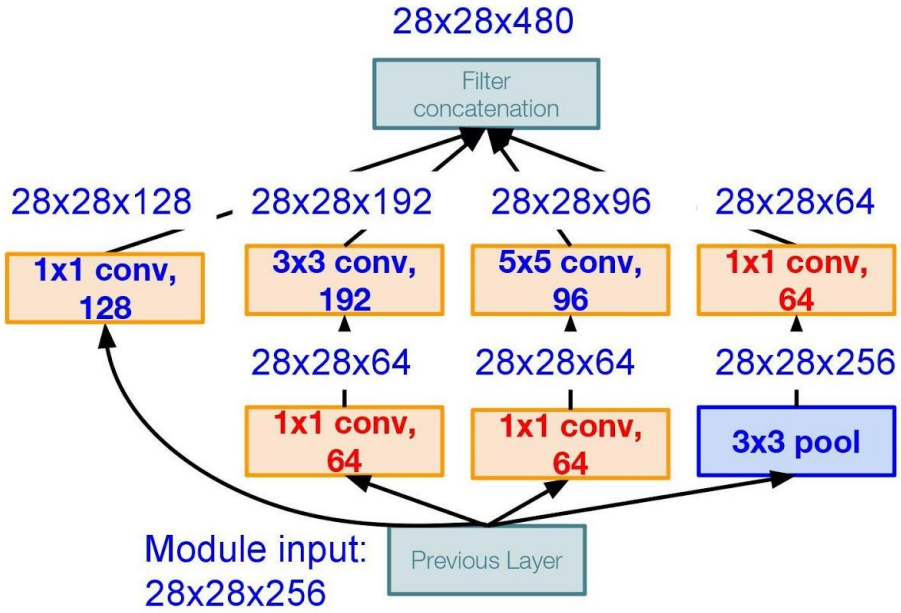
1x1 conv "bottleneck" layers



Inception module with dimension reduction

Case Study: GoogLeNet

[Szegedy et al., 2014]



Inception module with dimension reduction

Using same parallel layers as naive example, and adding "1x1 conv, 64 filter" bottlenecks:

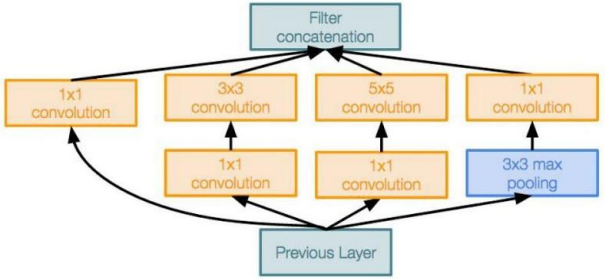
Conv Ops:
[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 64] 28x28x64x1x1x256
[1x1 conv, 128] 28x28x128x1x1x256
[3x3 conv, 192] 28x28x192x3x3x64
[5x5 conv, 96] 28x28x96x5x5x64
[1x1 conv, 64] 28x28x64x1x1x256
Total: 358M ops

Compared to 854M ops for naive version
Bottleneck can also reduce depth after pooling layer

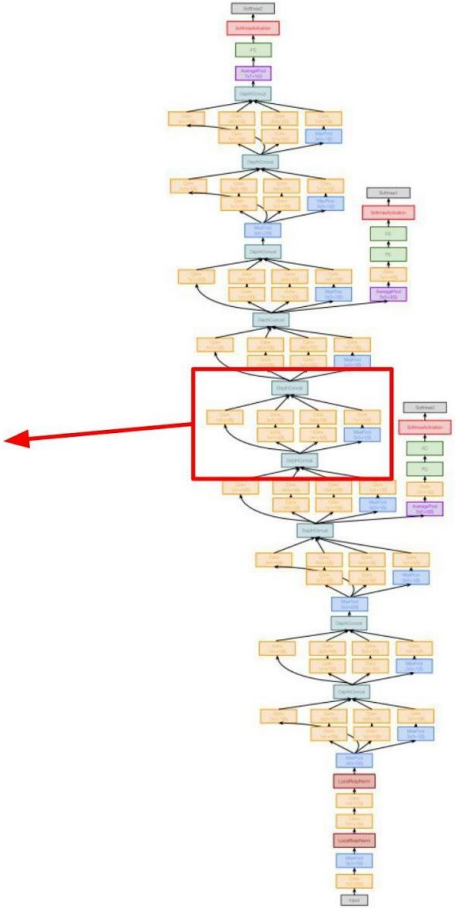
Case Study: GoogLeNet

[Szegedy et al., 2014]

Stack Inception modules with dimension reduction on top of each other



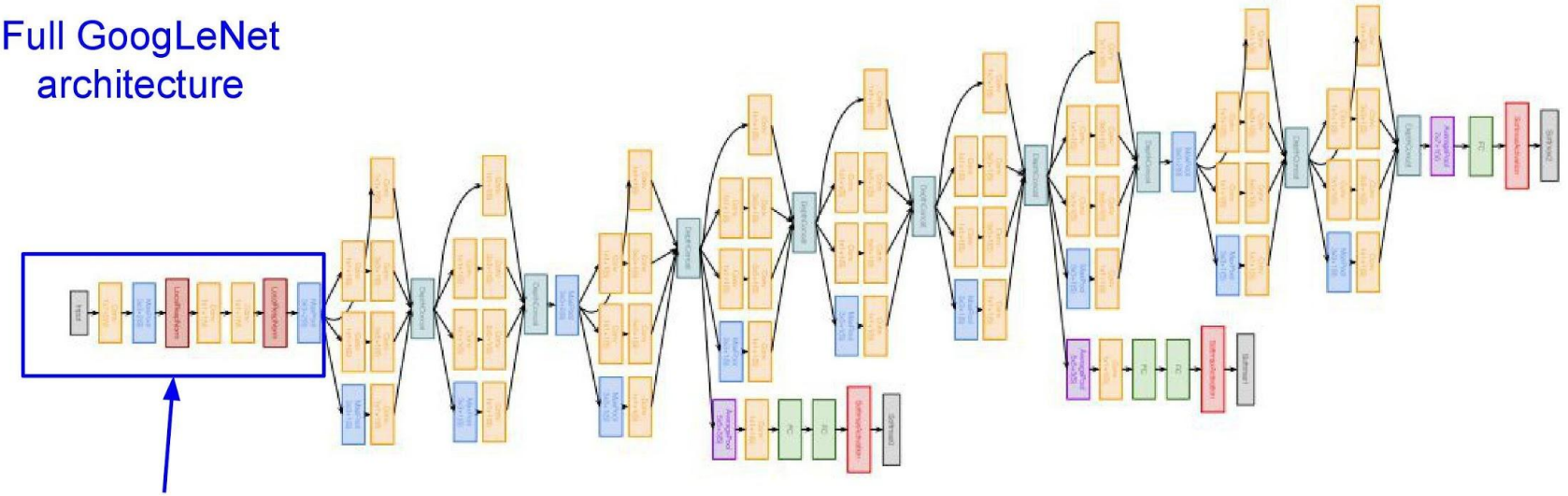
Inception module



Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

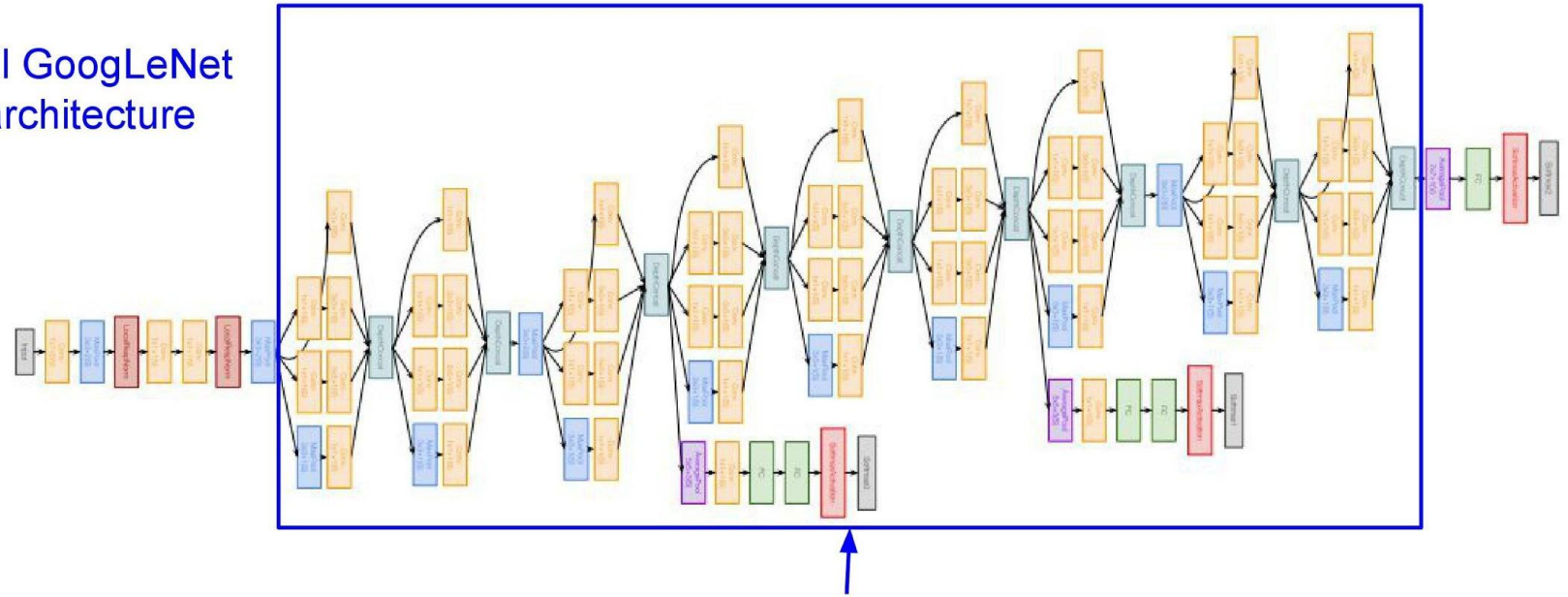


Stem Network: Conv-Pool- 2x Conv-Pool

Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

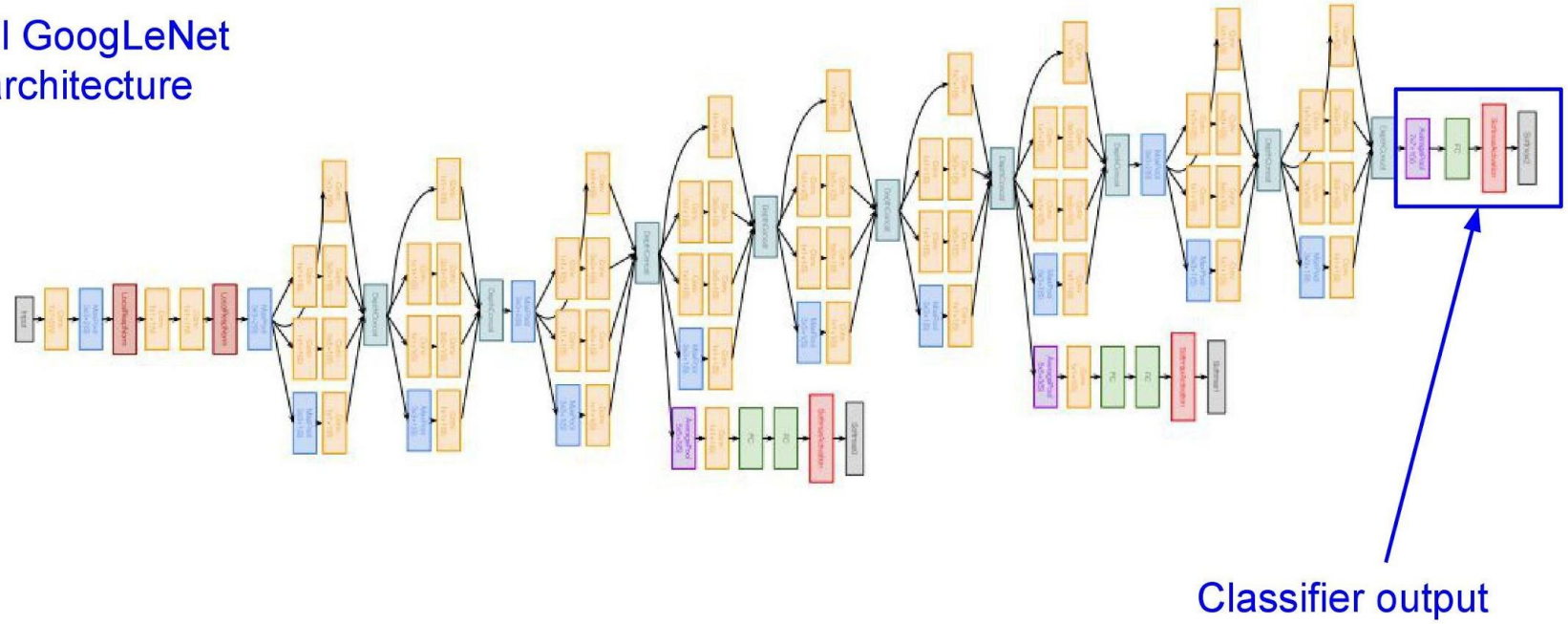


Stacked Inception Modules

Case Study: GoogLeNet

[Szegedy et al., 2014]

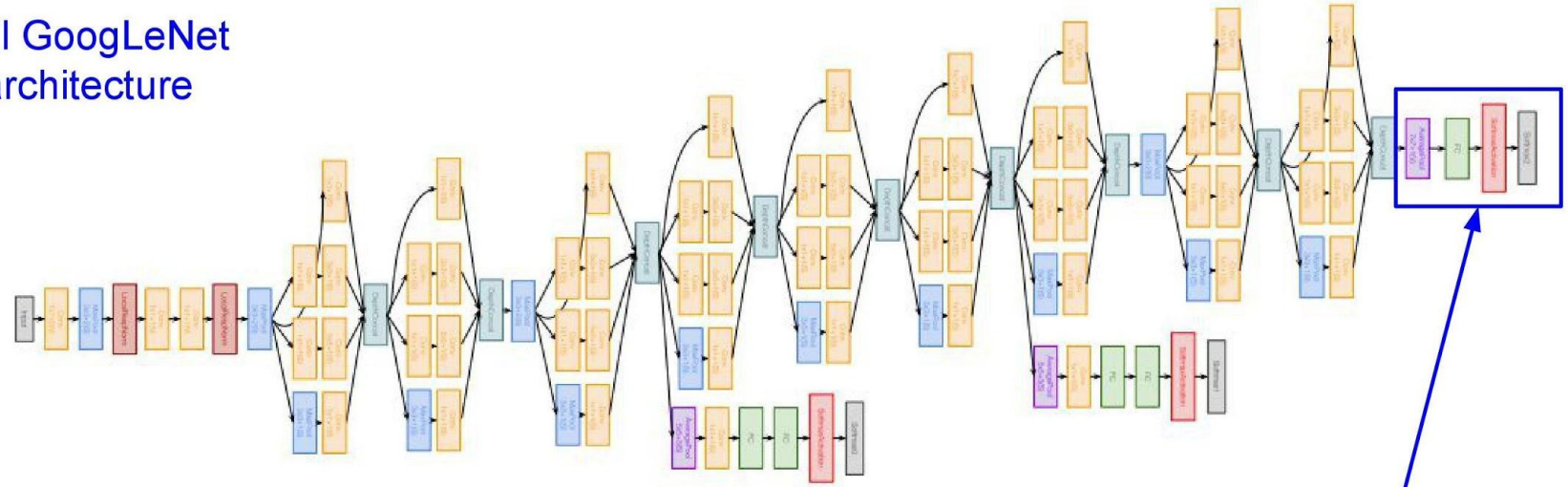
Full GoogLeNet architecture



Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

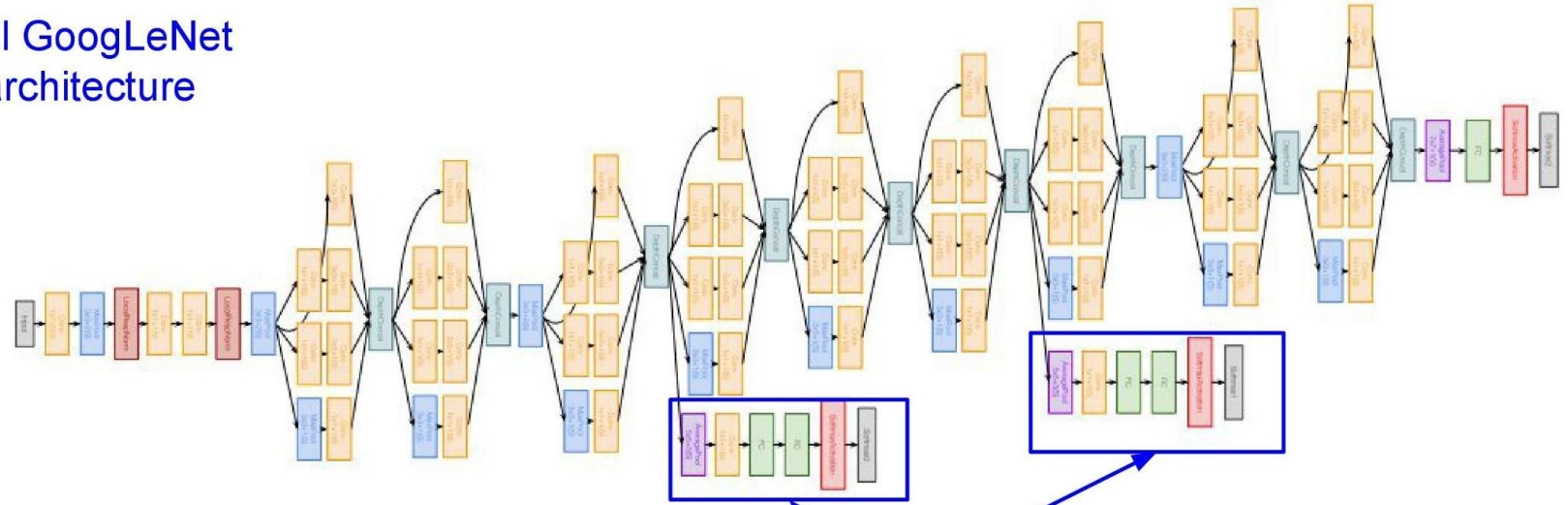


Classifier output
(removed expensive FC layers!)

Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture

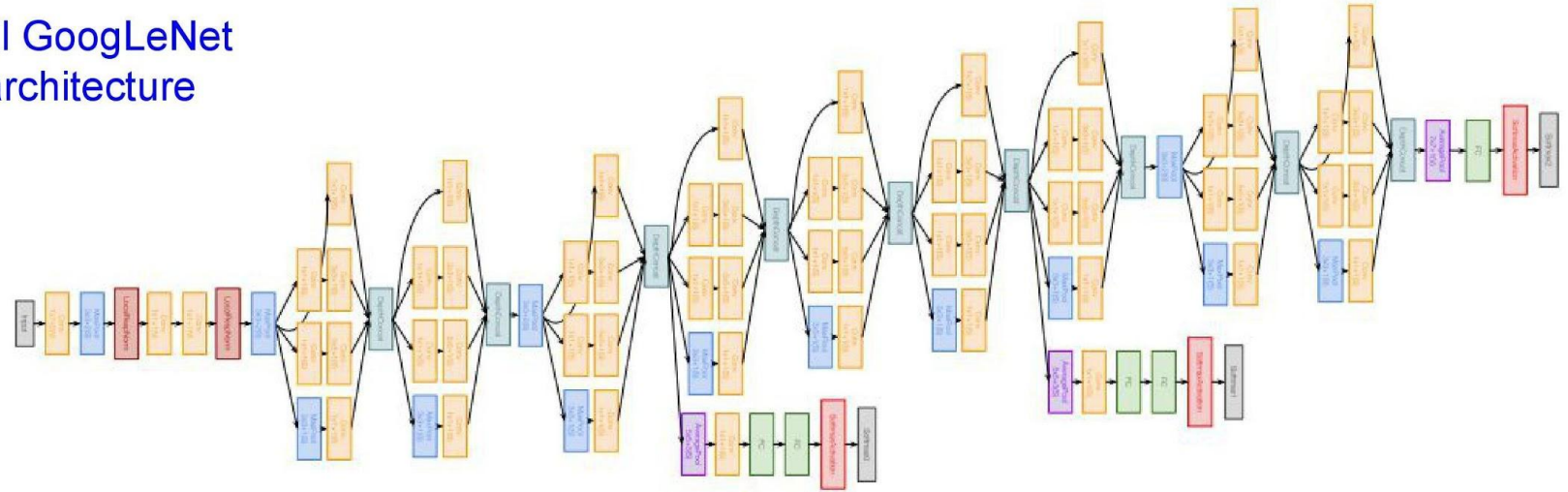


Auxiliary classification outputs to inject additional gradient at lower layers
(AvgPool-1x1Conv-FC-FC-Softmax)

Case Study: GoogLeNet

[Szegedy et al., 2014]

Full GoogLeNet architecture



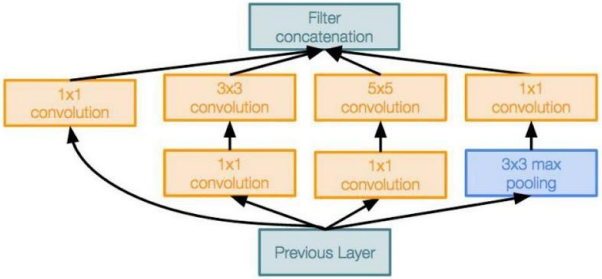
22 total layers with weights (including each parallel layer in an Inception module)

Case Study: GoogLeNet

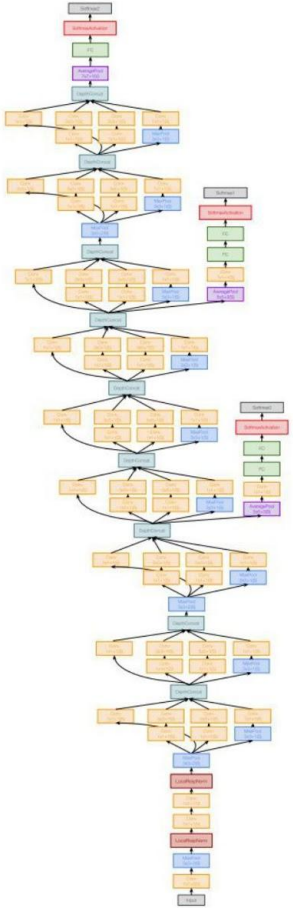
[Szegedy et al., 2014]

Deeper networks, with computational efficiency

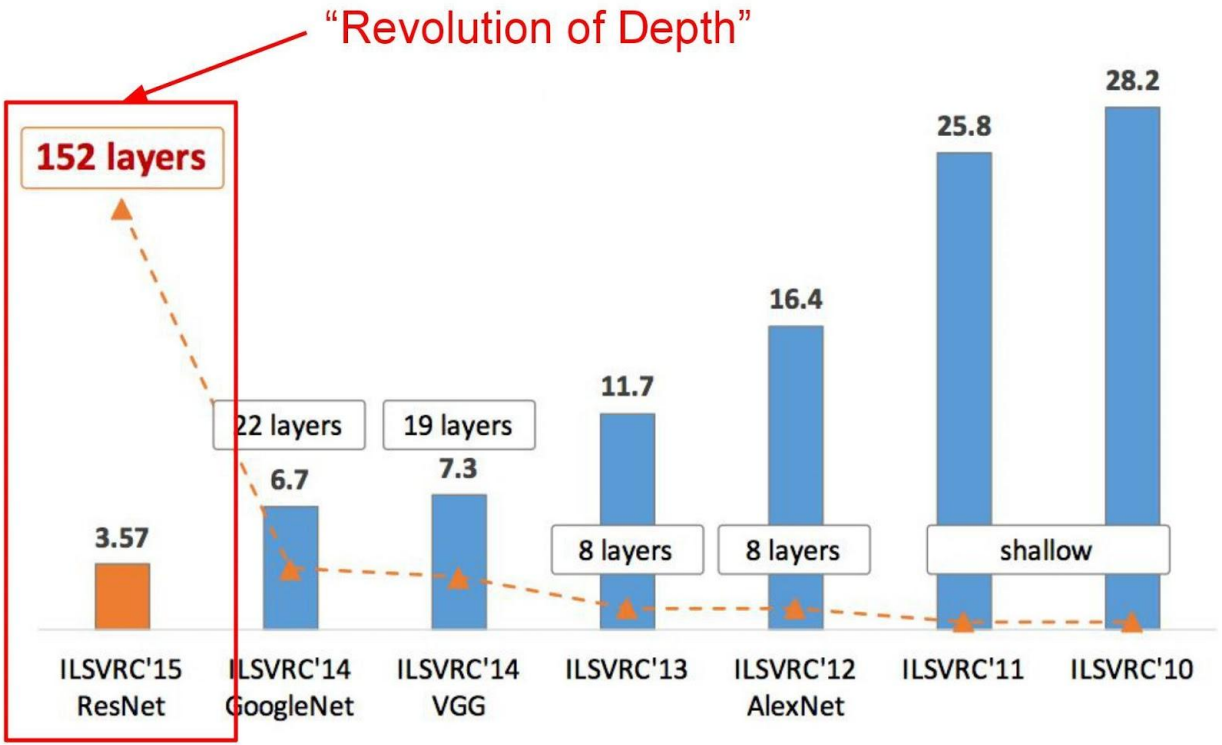
- 22 layers
- Efficient “Inception” module
- No FC layers
- 12x less params than AlexNet
- ILSVRC’14 classification winner (6.7% top 5 error)



Inception module



ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

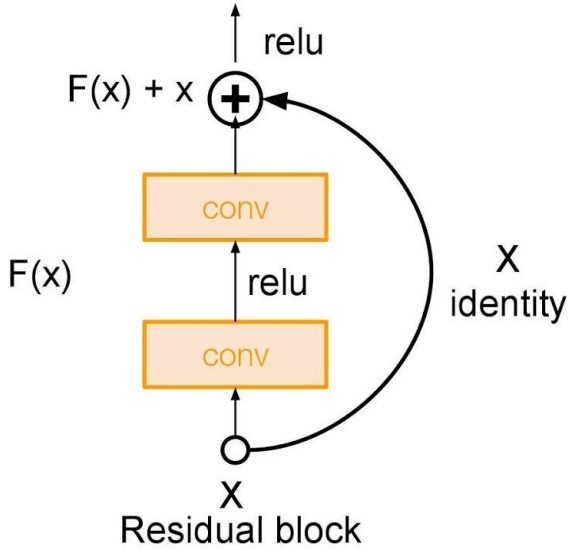


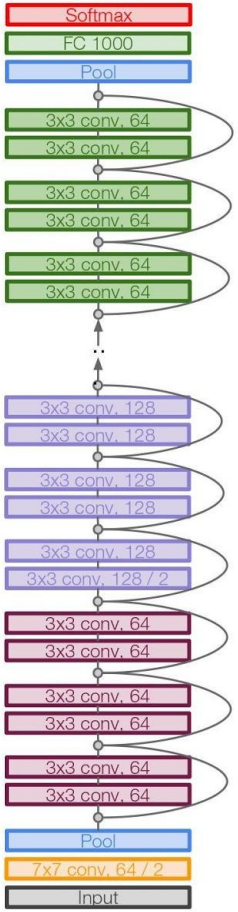
Case Study: ResNet

[He et al., 2015]

Very deep networks using residual connections

- 152-layer model for ImageNet
- ILSVRC'15 classification winner (3.57% top 5 error)
- Swept all classification and detection competitions in ILSVRC'15 and COCO'15!





Case Study: ResNet

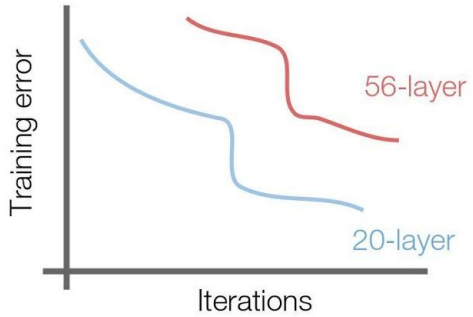
[He et al., 2015]

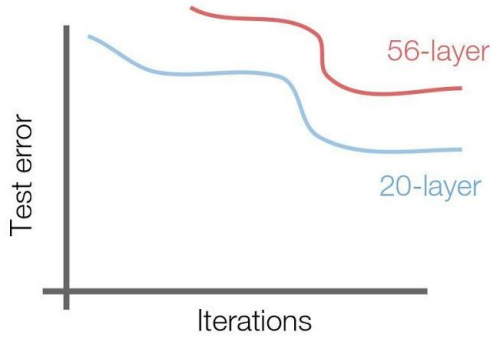
What happens when we continue stacking deeper layers on a “plain” convolutional neural network?

Case Study: ResNet

[He et al., 2015]

What happens when we continue stacking deeper layers on a “plain” convolutional neural network?





Q: What’s strange about these training and test curves?
[Hint: look at the order of the curves]

Case Study: ResNet

[He et al., 2015]

Hypothesis: the problem is an *optimization* problem, deeper models are harder to optimize

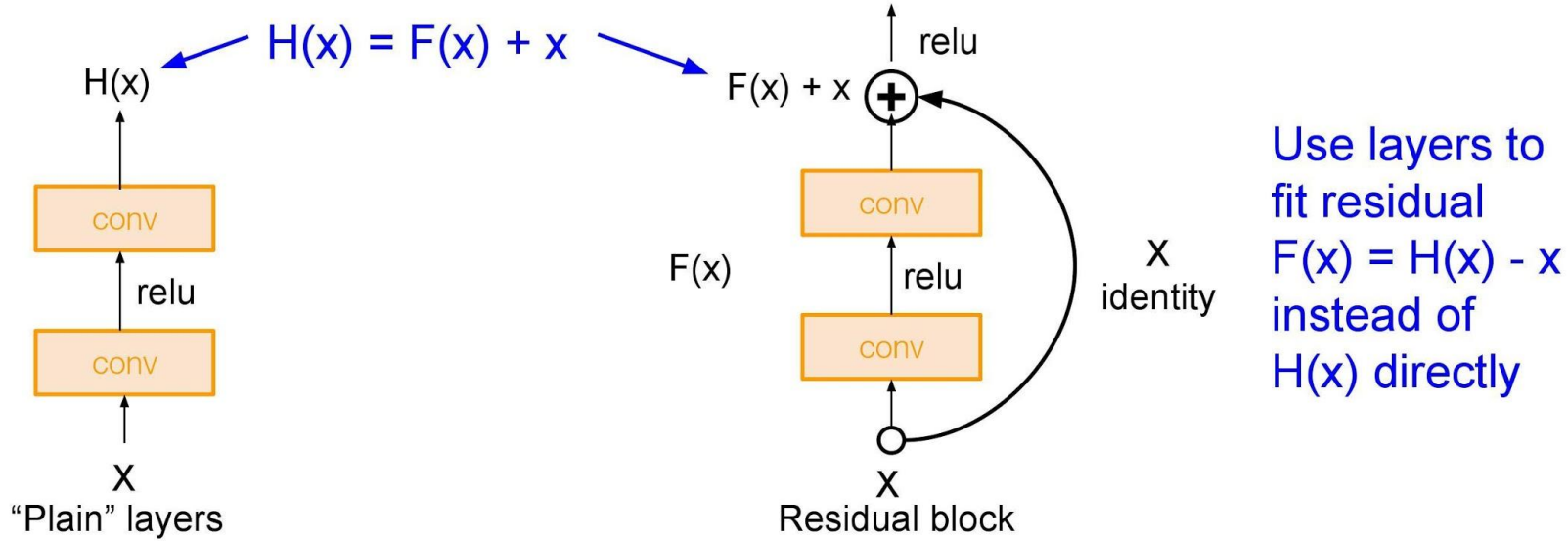
The deeper model should be able to perform at least as well as the shallower model.

A solution by construction is copying the learned layers from the shallower model and setting additional layers to identity mapping.

Case Study: ResNet

[He et al., 2015]

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping



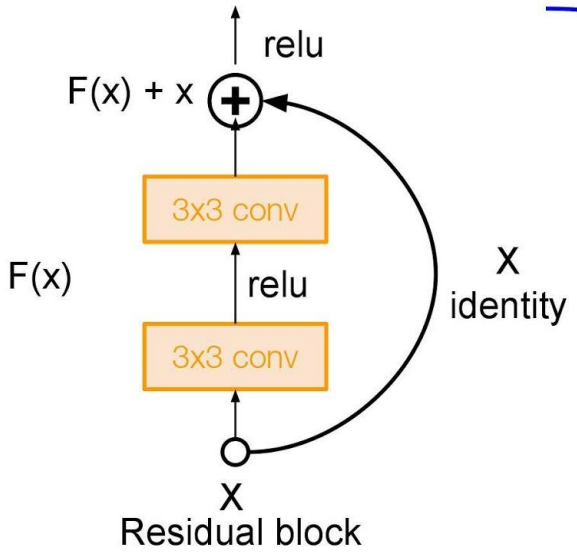
Use layers to fit residual F(x) = H(x) - x instead of H(x) directly

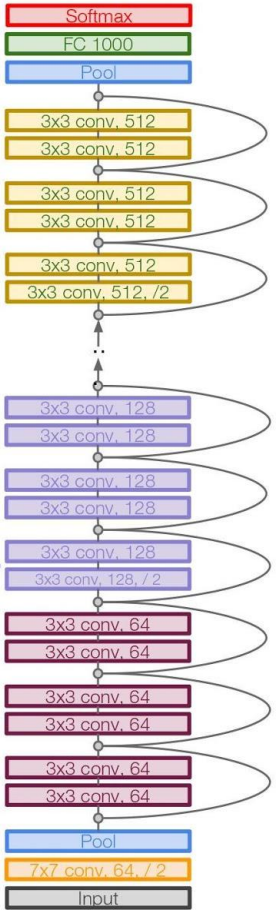
Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

- Stack residual blocks
- Every residual block has two 3x3 conv layers



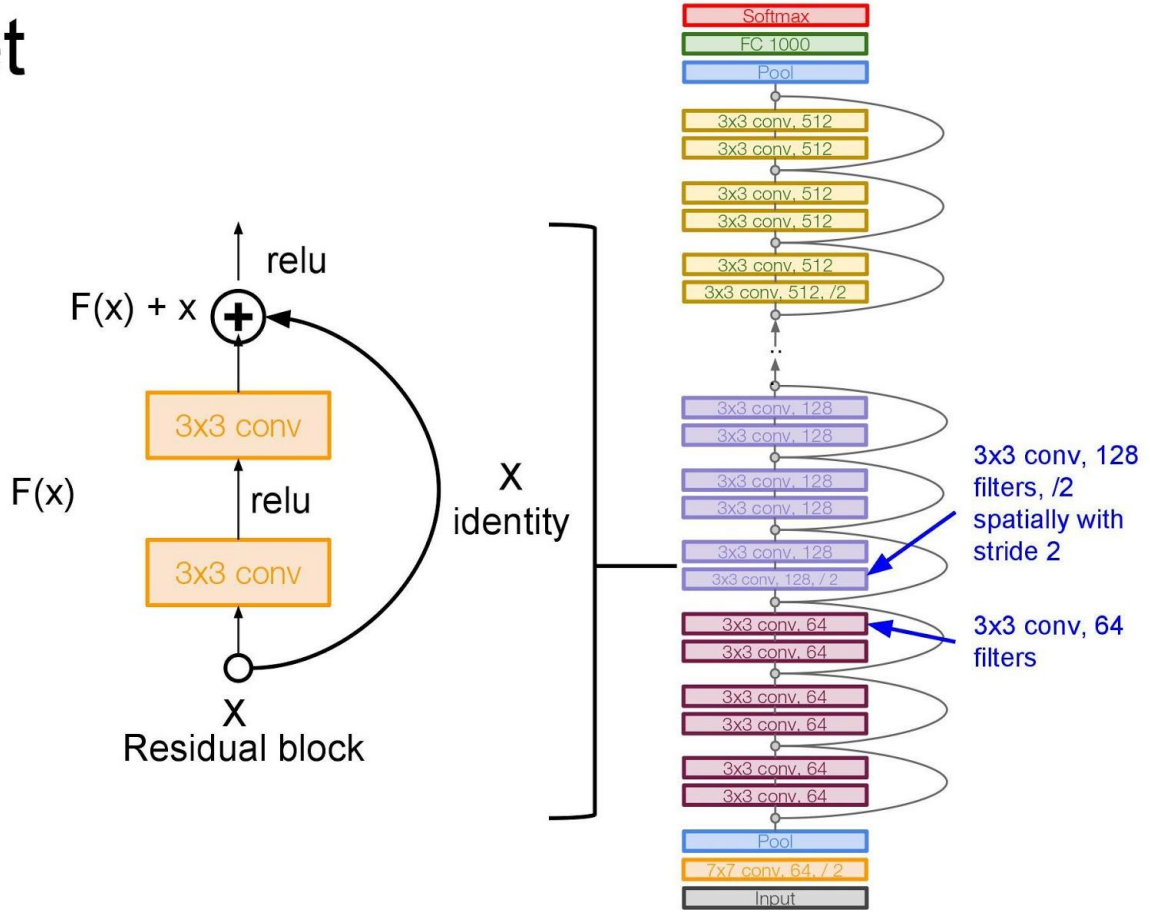


Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)

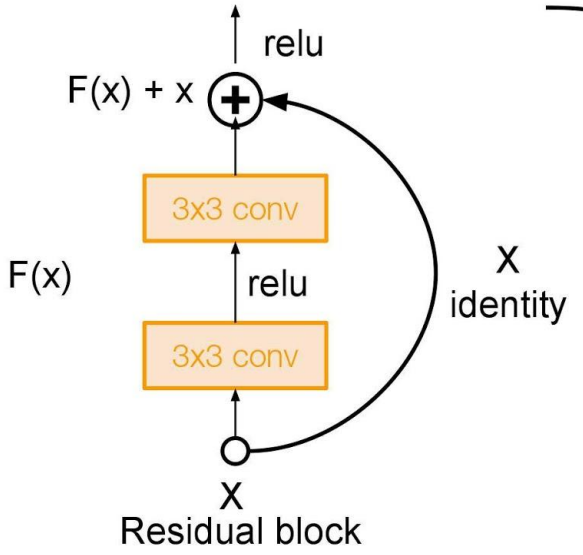


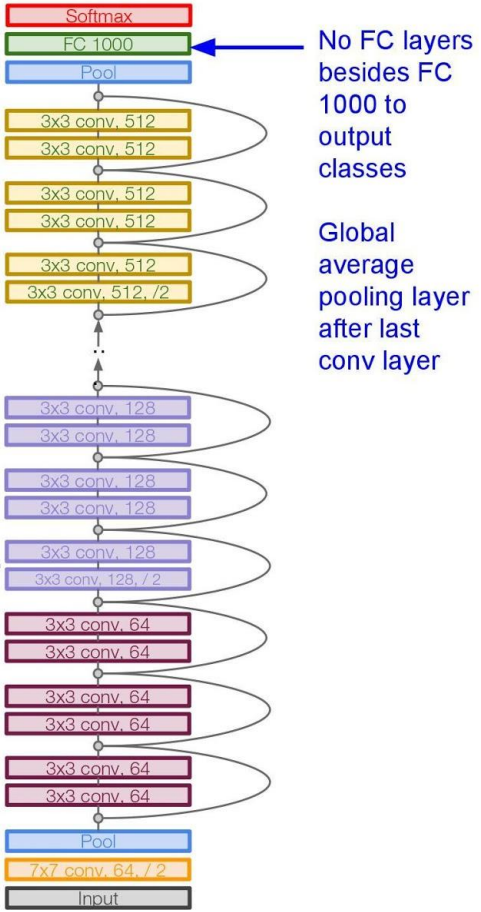
Case Study: ResNet

[He et al., 2015]

Full ResNet architecture:

- Stack residual blocks
- Every residual block has two 3x3 conv layers
- Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
- Additional conv layer at the beginning
- No FC layers at the end (only FC 1000 to output classes)

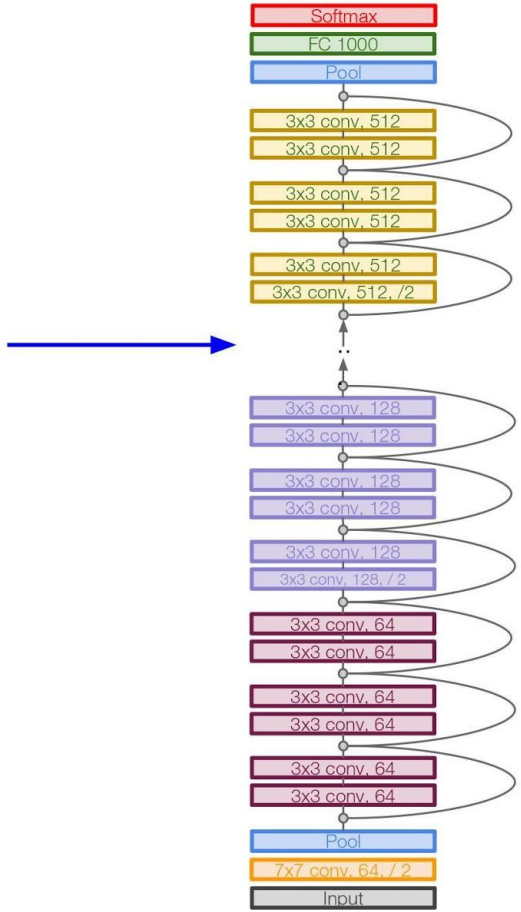




Case Study: ResNet

[He et al., 2015]

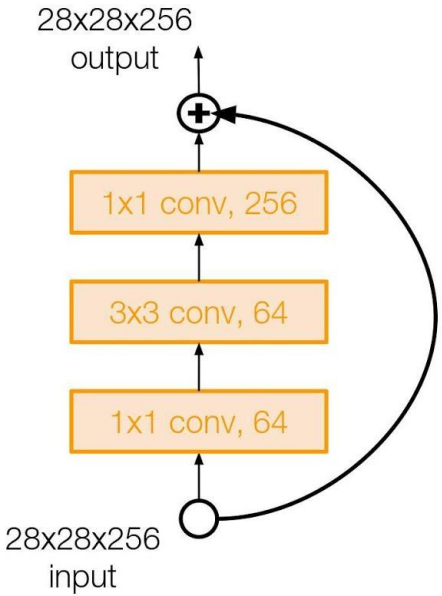
Total depths of 34, 50, 101, or 152 layers for ImageNet



Case Study: ResNet

[He et al., 2015]

For deeper networks (ResNet-50+), use “bottleneck” layer to improve efficiency (similar to GoogLeNet)



Case Study: ResNet

[He et al., 2015]

Training ResNet in practice:

- Batch Normalization after every CONV layer
- Xavier/2 initialization from He et al.
- SGD + Momentum (0.9)
- Learning rate: 0.1, divided by 10 when validation error plateaus
- Mini-batch size 256
- Weight decay of 1e-5
- No dropout used

Case Study: ResNet

[He et al., 2015]

Experimental Results

- Able to train very deep networks without degrading (152 layers on ImageNet, 1202 on Cifar)
- Deeper networks now achieve lowing training error as expected
- Swept 1st place in all ILSVRC and COCO 2015 competitions

MSRA @ ILSVRC & COCO 2015 Competitions

- **1st places in all five main tracks**
  - ImageNet Classification: "*Ultra-deep*" (quote Yann) 152-layer nets
  - ImageNet Detection: 16% better than 2nd
  - ImageNet Localization: 27% better than 2nd
  - COCO Detection: 11% better than 2nd
  - COCO Segmentation: 12% better than 2nd

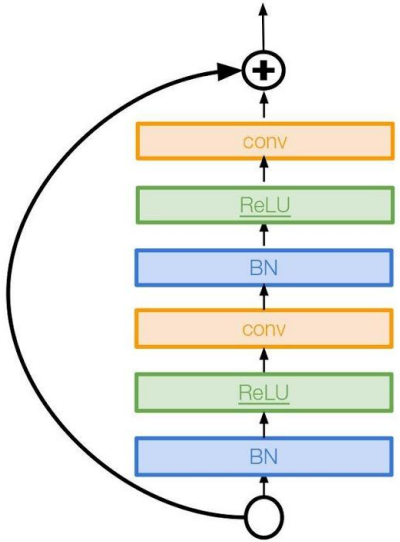
Other architectures to know…

Improving ResNets...

Identity Mappings in Deep Residual Networks

[He et al. 2016]

- Improved ResNet block design from creators of ResNet
- Creates a more direct path for propagating information throughout network (moves activation to residual mapping pathway)
- Gives better performance

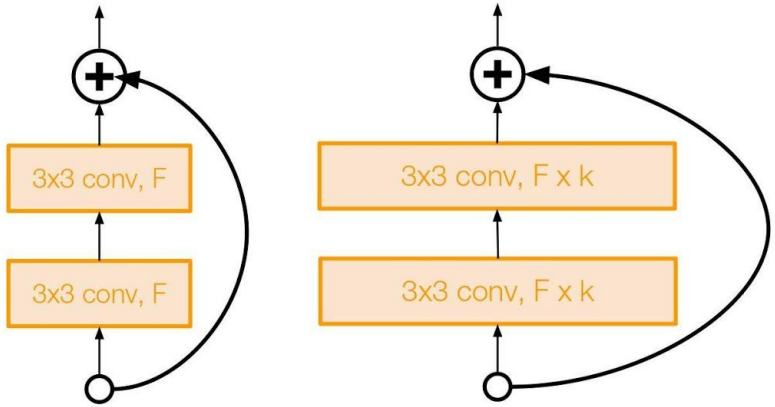


Improving ResNets...

Wide Residual Networks

[Zagoruyko et al. 2016]

- Argues that residuals are the important factor, not depth
- User wider residual blocks (F x k filters instead of F filters in each layer)
- 50-layer wide ResNet outperforms 152-layer original ResNet
- Increasing width instead of depth more computationally efficient (parallelizable)



Basic residual block

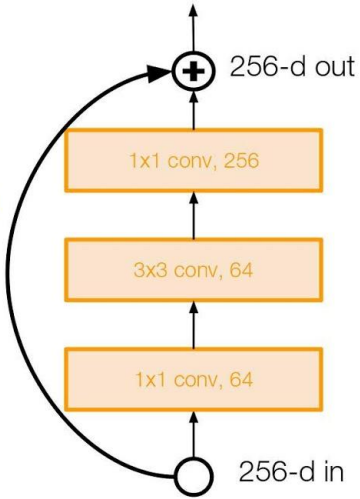
Wide residual block

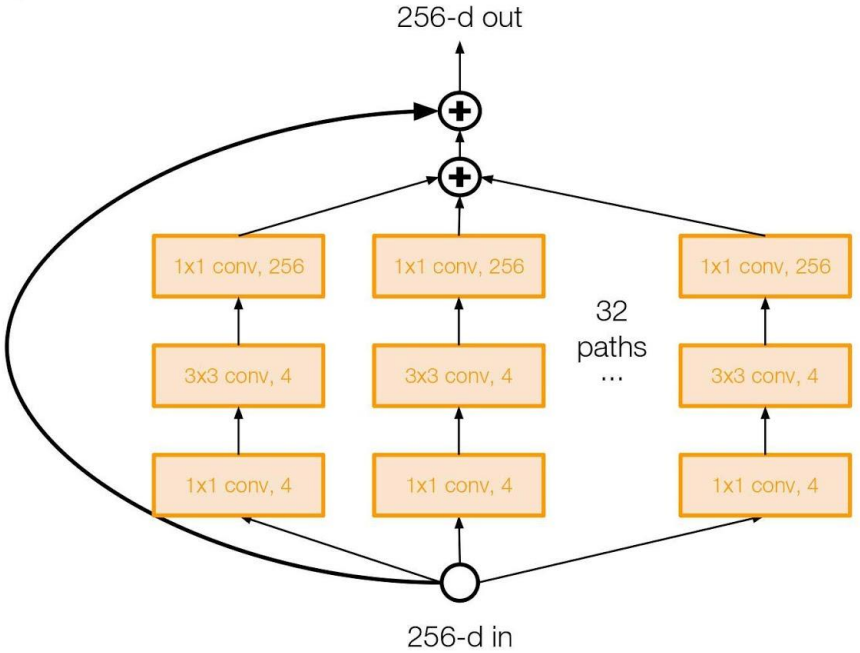
Improving ResNets...

Aggregated Residual Transformations for Deep Neural Networks (ResNeXt)

[Xie et al. 2016]

- Also from creators of ResNet
- Increases width of residual block through multiple parallel pathways ("cardinality")
- Parallel pathways similar in spirit to Inception module



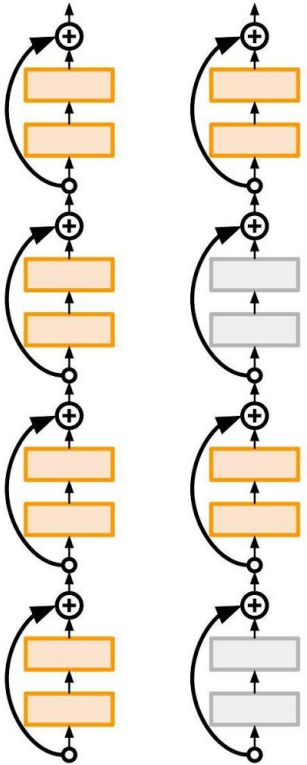


Improving ResNets...

Deep Networks with Stochastic Depth

[Huang et al. 2016]

- Motivation: reduce vanishing gradients and training time through short networks during training
- Randomly drop a subset of layers during each training pass
- Bypass with identity function
- Use full deep network at test time

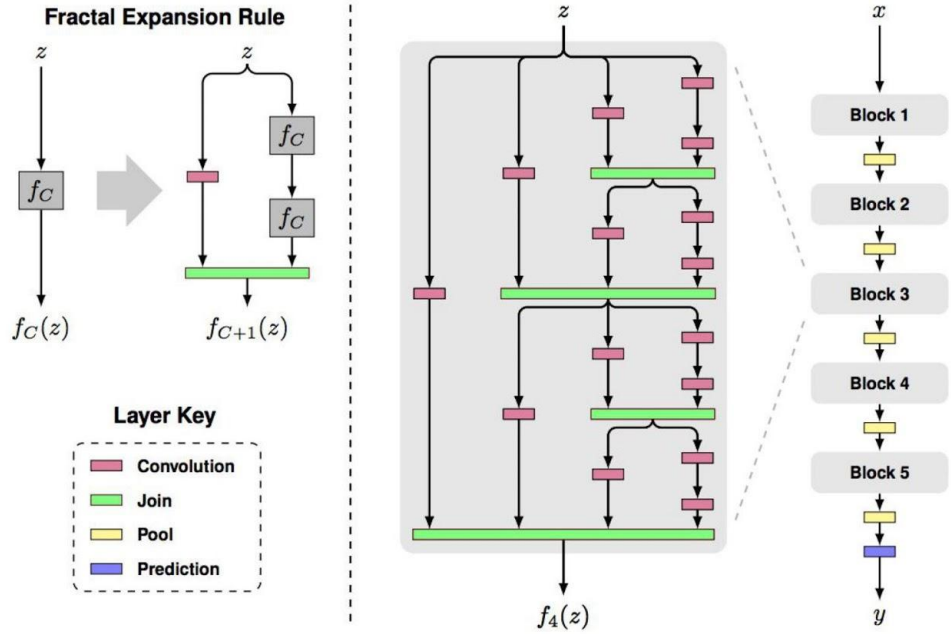


Beyond ResNets...

FractalNet: Ultra-Deep Neural Networks without Residuals

[Larsson et al. 2017]

- Argues that key is transitioning effectively from shallow to deep and residual representations are not necessary
- Fractal architecture with both shallow and deep paths to output
- Trained with dropping out sub-paths
- Full network at test time



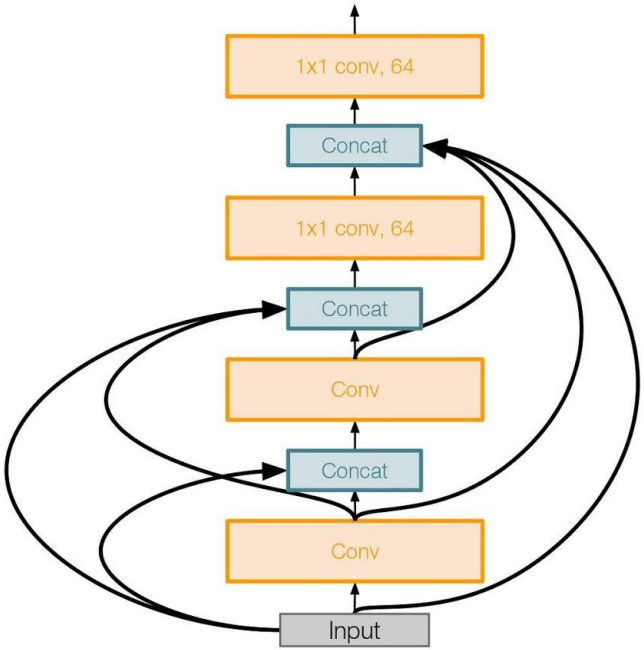
Figures copyright Larsson et al., 2017. Reproduced with permission.

Beyond ResNets...

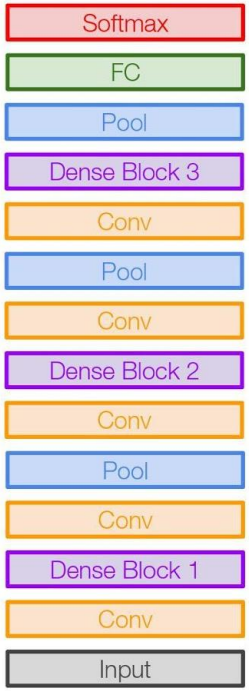
Densely Connected Convolutional Networks

[Huang et al. 2017]

- Dense blocks where each layer is connected to every other layer in feedforward fashion
- Alleviates vanishing gradient, strengthens feature propagation, encourages feature reuse



Dense Block



Dual Path Networks (DPN)

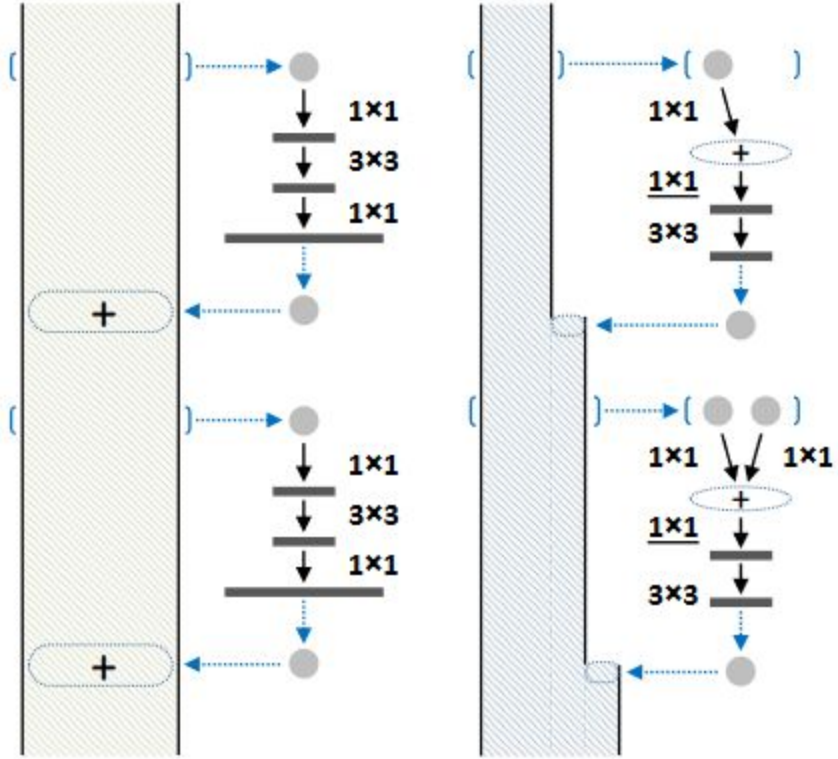
arXiv Preprint

https://arxiv.org/abs/1707.01629

Code & Trained Models

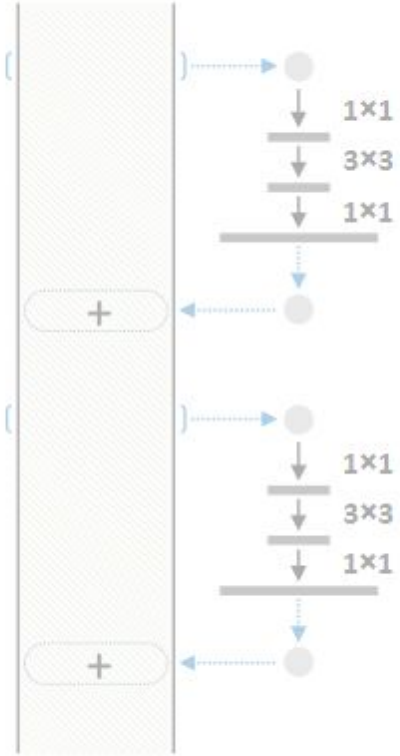
https://github.com/cypw/DPNs

Motivation



(a) Residual Network (b) Densely Connected Network [1]

Analysis



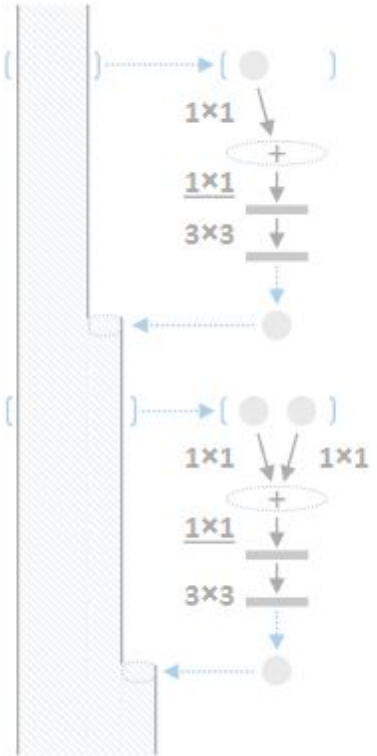
(a) Residual Network

- Residual Networks are essentially Densely Connected Networks but with shared connections.

- Advantage:

  - ResNet:
    features refinement (reuse feature)

  - DenseNet:
    keep exploring new features



(b) Densely Connected Network

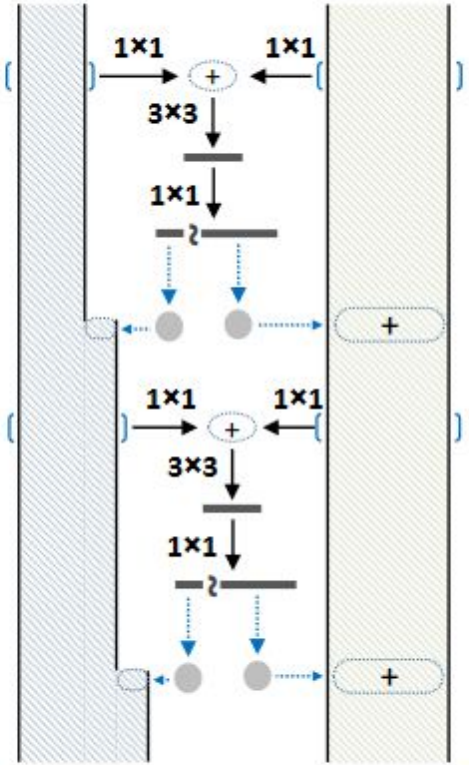
Dual Path Architecture

When managing a company:

Employees need to keep improving the skills (Feature refinement)

Also need to hire freshman to the company (Feature exploration)

Dual Path Architecture

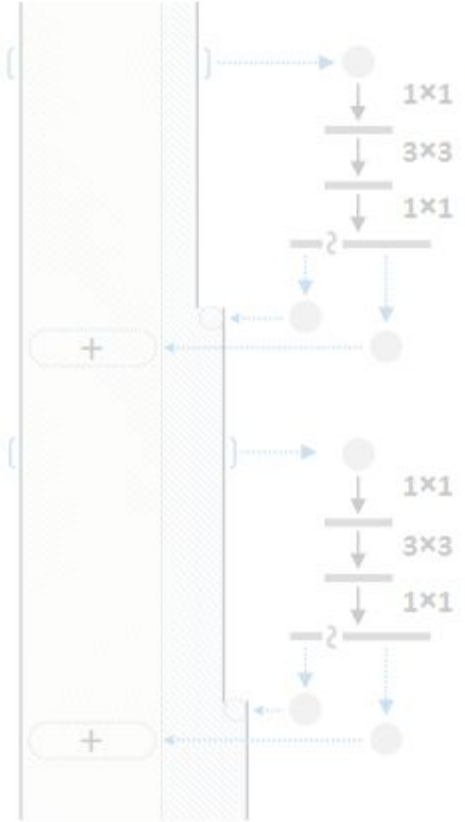


(d) Dual Path Architecture

Explore New Features

Feature Refinement
(Reuse Features)

Dual Path Networks



(e) DPN

- Three DPNs are designed:

DPN-92, DPN-98, DPN-131

depth=92 depth=98 depth=131

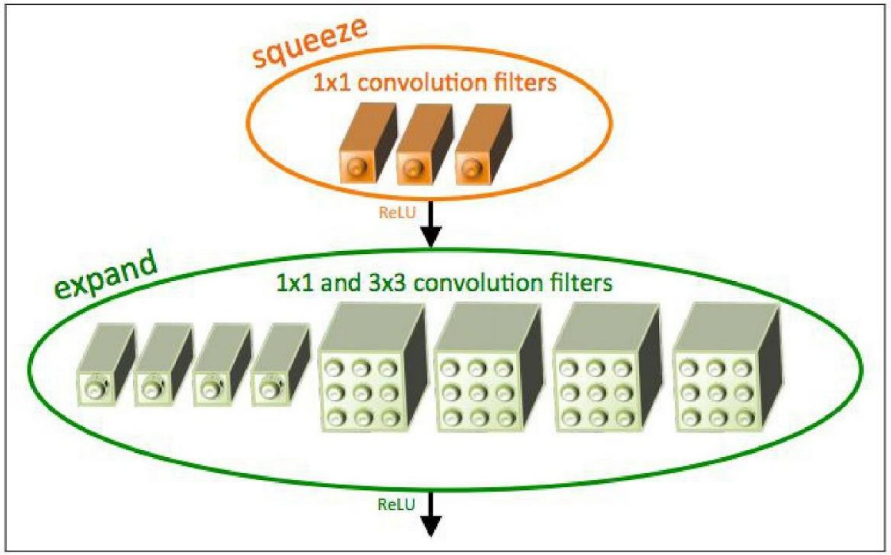
ResNeXt-101 (64x4d) [2]		DPN-98	
320 MB	Model Size	236 MB	- 26%
15.5	GFLOPs	11.7	- 25%
12.1 GB	GPU Memory	11.1 GB	- 8%
20.4 / 5.3	Top 1 / Top 5 Error.	20.2 / 5.2	

Efficient networks...

SqueezeNet: AlexNet-level Accuracy With 50x Fewer Parameters and <0.5Mb Model Size

[Iandola et al. 2017]

- Fire modules consisting of a 'squeeze' layer with 1x1 filters feeding an 'expand' layer with 1x1 and 3x3 filters
- AlexNet level accuracy on ImageNet with 50x fewer parameters
- Can compress to 510x smaller than AlexNet (0.5Mb)



Summary: CNN Architectures

- VGG, GoogLeNet, ResNet all in wide use, available in model zoos
- ResNet current best default
- Trend towards extremely deep networks
- Significant research centers around design of layer / skip connections and improving gradient flow
- Even more recent trend towards examining necessity of depth vs. width and residual connections