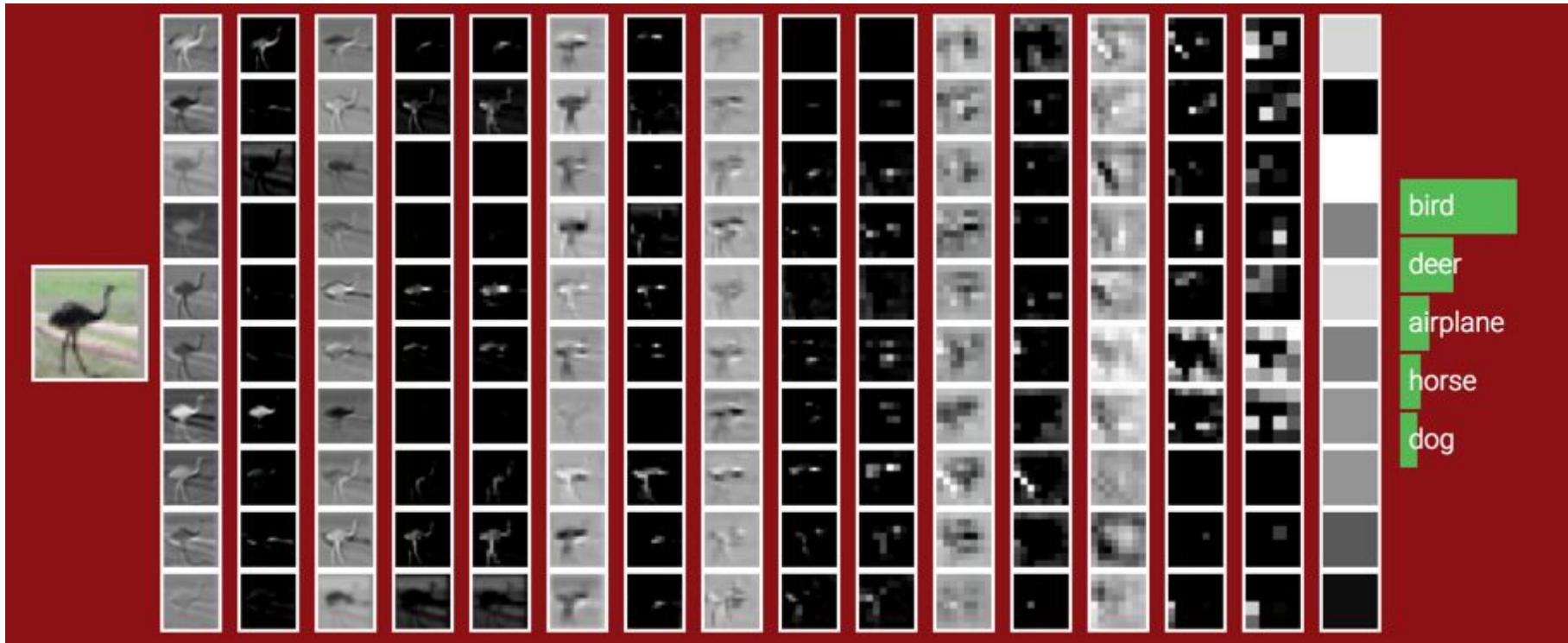
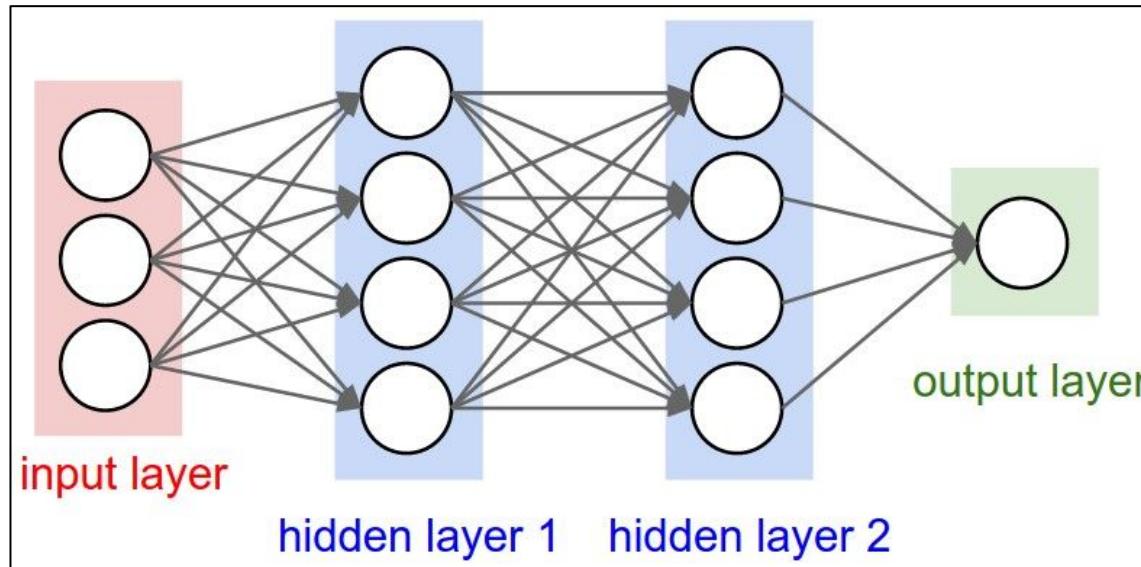


Deep Learning for Computer Vision

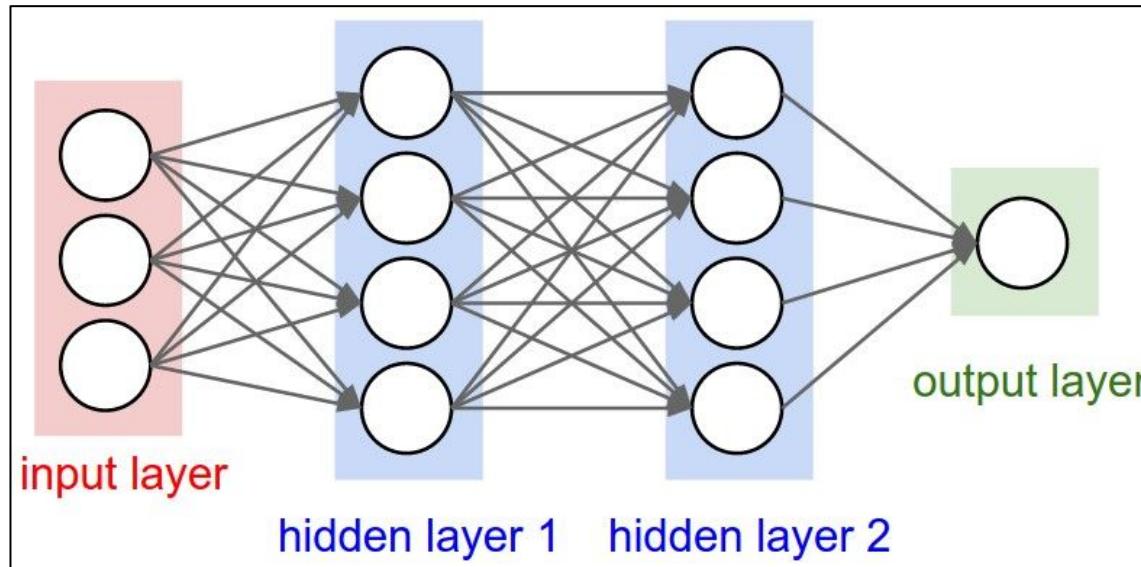


Andrej Karpathy
Bay Area Deep Learning School, 2016

So far...

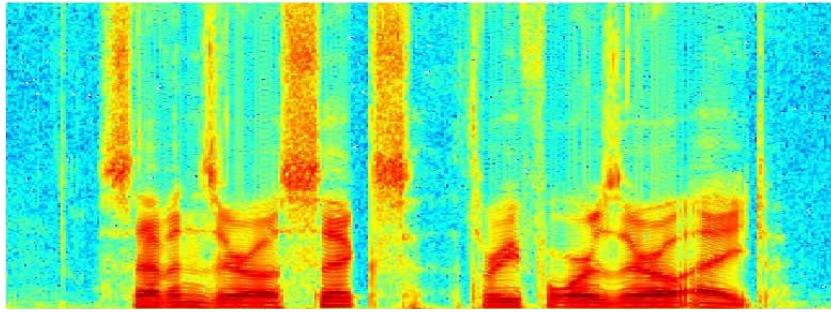


So far...



Some input vector (very few assumptions made).

In many real-world applications input vectors have **structure**.



Spectrograms

"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum."

Text



Images

Convolutional Neural Networks: A pinch of history

Hubel & Wiesel,

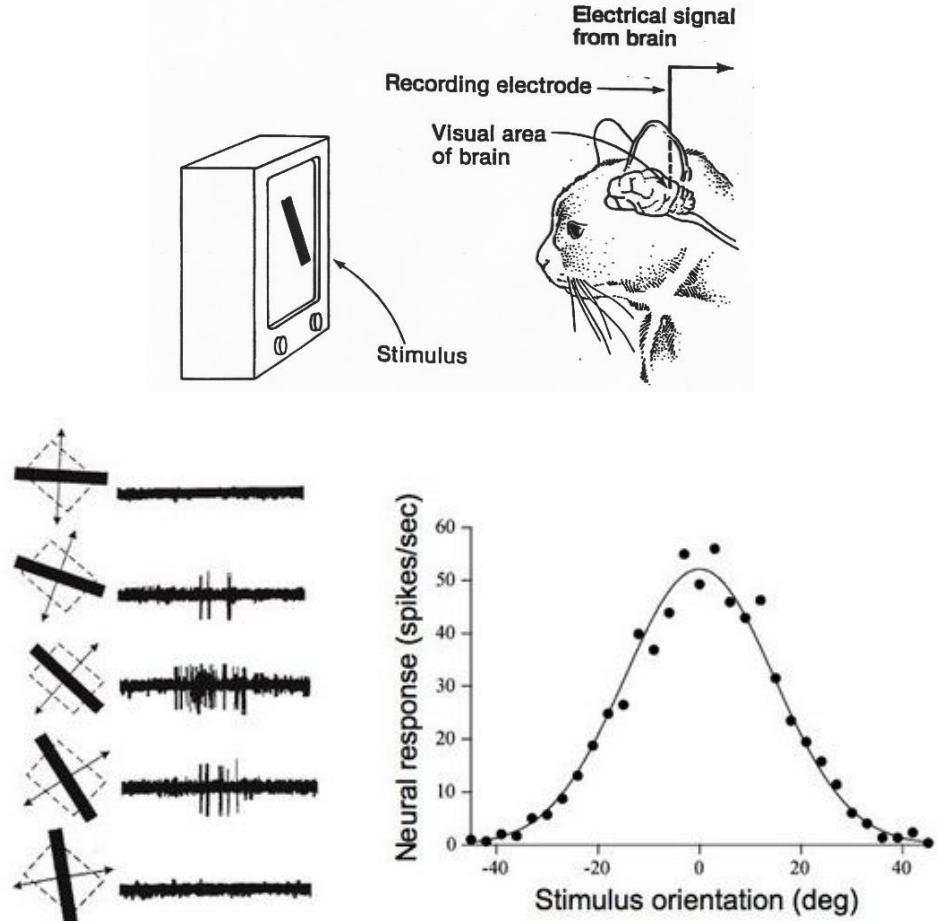
1959

RECEPTIVE FIELDS OF SINGLE
NEURONES IN
THE CAT'S STRIATE CORTEX

1962

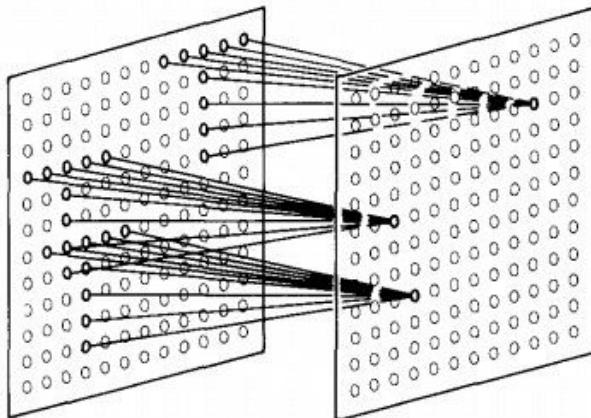
RECEPTIVE FIELDS, BINOCULAR
INTERACTION
AND FUNCTIONAL ARCHITECTURE IN
THE CAT'S VISUAL CORTEX

1968...

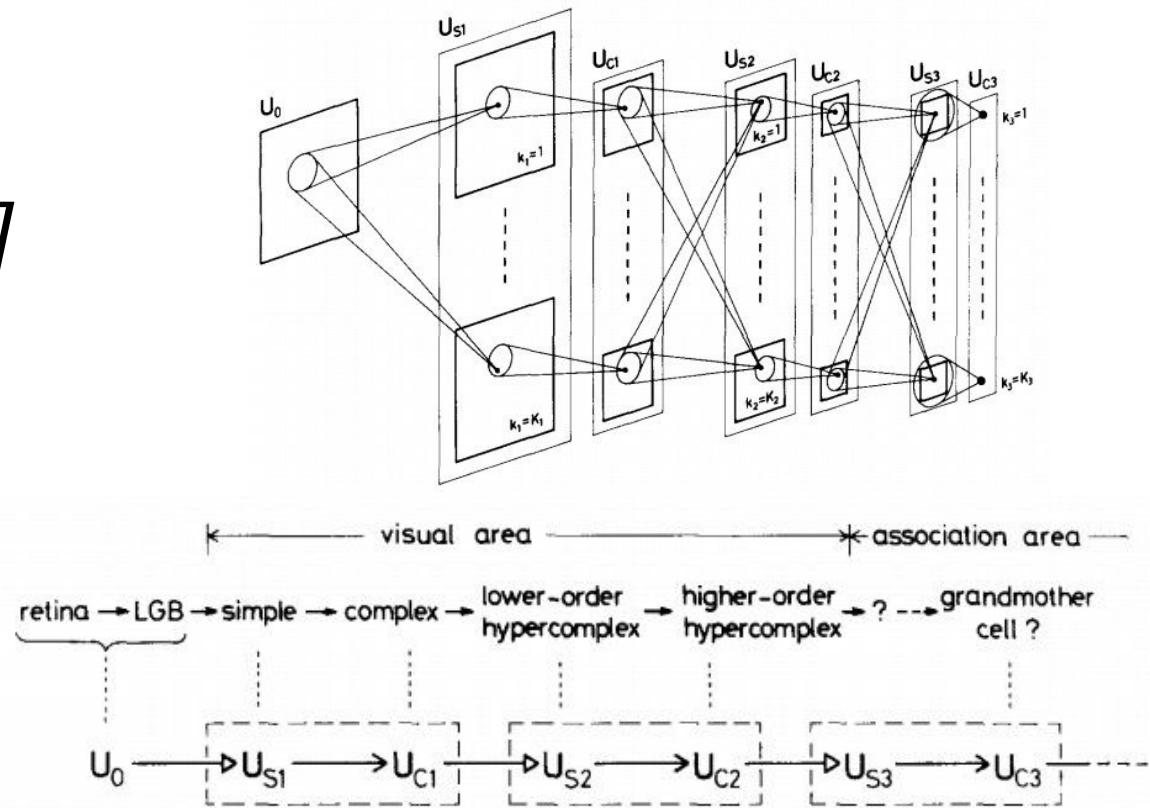


A bit of history:

Neurocognitron [Fukushima 1980]

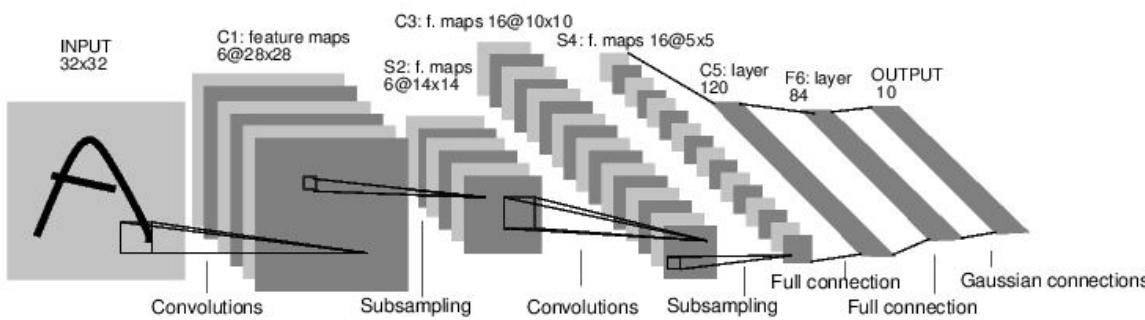


“sandwich” architecture (SCSCSC...)
simple cells: modifiable parameters
complex cells: perform pooling

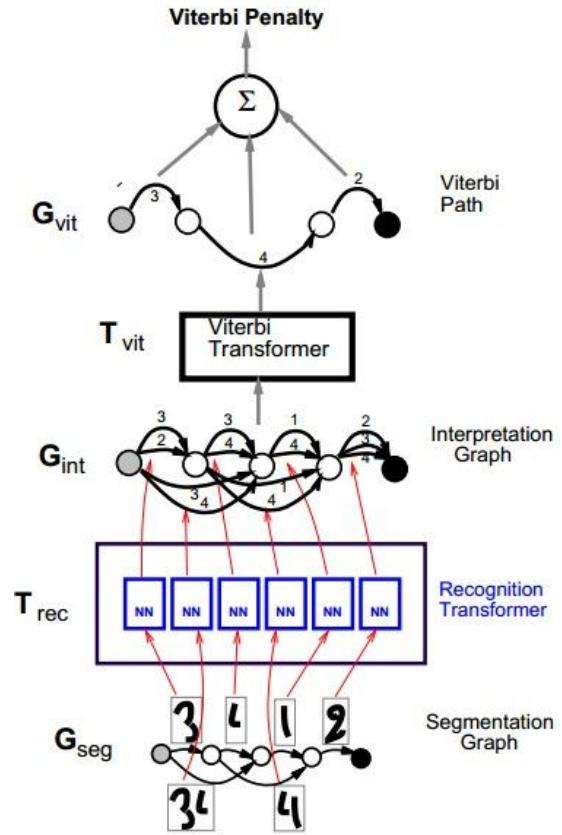


Gradient-based learning applied to document recognition

[LeCun, Bottou, Bengio, Haffner 1998]



LeNet-5



Computer Vision 2011



Computer Vision 2011

4.1. Image Features and Kernels

We selected or designed several state-of-art features that are potentially useful for scene classification. GIST features [21] are proposed specifically for scene recognition tasks. Dense SIFT features are also found to perform very well at the 15-category dataset [17]. We also evaluate sparse SIFTS as used in “Video Google” [27]. HOG features provide excellent performance for object and human recognition tasks [4, 9], so it is interesting to examine their utility for scene recognition. While SIFT is known to be very good at finding repeated image content, the self-similarity descriptor (SSIM) [26] relates images using their internal layout of local self-similarities. Unlike GIST, SIFT, and HOG, which are all local gradient-based approaches, SSIM may provide a distinct, complementary measure of scene layout that is somewhat appearance invariant. As a baseline, we also include Tiny Images [28], color histograms and straight line histograms. To make our color and texton histograms more invariant to scene layout, we also build histograms for specific geometric classes as determined by [13]. The geometric classification of a scene is then itself used as a feature, hopefully being invariant to appearance but responsive to

layout.

GIST: The GIST descriptor [21] computes the output energy of a bank of 24 filters. The filters are Gabor-like filters tuned to 8 orientations at 4 different scales. The square output of each filter is then averaged on a 4×4 grid.

HOG2x2: First, histogram of oriented edges (HOG) descriptors [4] are densely extracted on a regular grid at steps of 8 pixels. HOG features are computed using the code available online provided by [9], which gives a 31-dimension descriptor for each node of the grid. Then, 2×2 neighboring HOG descriptors are stacked together to form a descriptor with 124 dimensions. The stacked descriptors spatially overlap. This 2×2 neighbor stacking is important because the higher feature dimensionality provides more descriptive power. The descriptors are quantized into 300 visual words by k -means. With this visual word representation, three-level spatial histograms are computed on grids of 1×1 , 2×2 and 4×4 . Histogram intersection[17] is used to define the similarity of two histograms at the same pyramid level for two images. The kernel matrices at the three levels are normalized by their respective means, and linearly combined together using equal weights.

Dense SIFT: As with HOG2x2, SIFT descriptors are densely extracted [17] using a flat rather than Gaussian window at two scales (4 and 8 pixel radii) on a regular grid at steps of 5 pixels. The three descriptors are stacked together for each HSV color channels, and quantized into 300 visual words by k -means, and spatial pyramid histograms are used as kernels[17].

Computer Vision 2011

LBP: Local Binary Patterns (LBP) [20] is a powerful texture feature based on occurrence histogram of local binary patterns. We can regard the scene recognition as a texture classification problem of 2D images, and therefore apply this model to our problem. We also try the rotation invariant extension version [2] of LBP to examine whether rotation invariance is suitable for scene recognition.

Sparse SIFT histograms: As in “Video Google” [27], we build SIFT features at Hessian-affine and MSER [19] interest points. We cluster each set of SIFTS, independently, into dictionaries of 1,000 visual words using k -means. An image is represented by two histograms counting the number of sparse SIFTS that fall into each bin. An image is represented by two 1,000 dimension histograms where each SIFT is soft-assigned, as in [22], to its nearest cluster centers. Kernels are computed with χ^2 distance.

SSIM: Self-similarity descriptors [26] are computed on a regular grid at steps of five pixels. Each descriptor is obtained by computing the correlation map of a patch of 5×5 in a window with radius equal to 40 pixels, then quantizing it in 3 radial bins and 10 angular bins, obtaining 30 dimensional descriptor vectors. The descriptors are then quantized into 300 visual words by k -means and we use χ^2 distance on spatial histograms for the kernels.

Tiny Images: The most trivial way to match scenes is to compare them directly in color image space. Reducing the image dimensions drastically makes this approach more computationally feasible and less sensitive to exact align-

ment. This method of image matching has been examined thoroughly by Torralba et al.[28] for the purpose of object recognition and scene classification.

Line Features: We detect straight lines from Canny edges using the method described in Video Compass [15]. For each image we build two histograms based on the statistics of detected lines— one with bins corresponding to line angles and one with bins corresponding to line lengths. We use an RBF kernel to compare these unnormalized histograms. This feature was used in [11].

Texton Histograms: We build a 512 entry universal texton dictionary [18] by clustering responses to a bank of filters with 8 orientations, 2 scales, and 2 elongations. For each image we then build a 512-dimensional histogram by assigning each pixel’s set of filter responses to the nearest texton dictionary entry. We compute kernels from normalized χ^2 distances.

Color Histograms: We build joint histograms of color in CIE $L^*a^*b^*$ color space for each image. Our histograms have 4, 14, and 14 bins in L , a , and b respectively for a total of 784 dimensions. We compute distances between these histograms using χ^2 distance on the normalized histograms.

Geometric Probability Map: We compute the geometric class probabilities for image regions using the method of Hoiem et al. [13]. We use only the ground, vertical, porous, and sky classes because they are more reliably classified. We reduce the probability maps for each class to 8×8 and use an RBF kernel. This feature was used in [11].

Computer Vision 2011

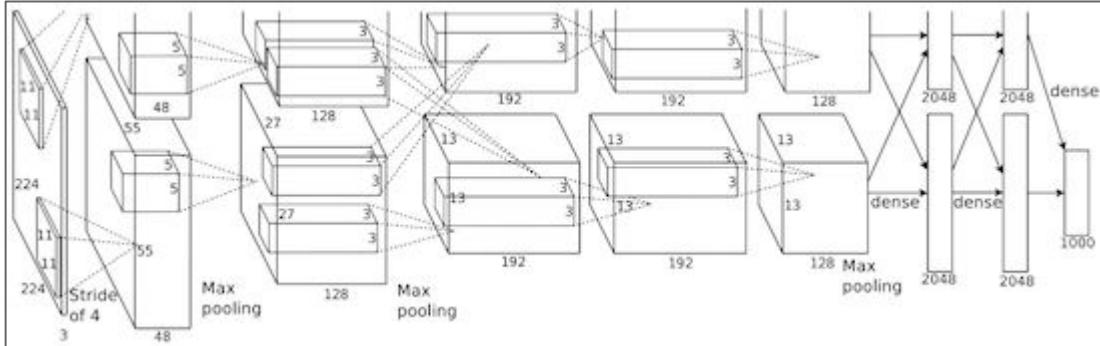
Geometry Specific Histograms: Inspired by “Illumination Context” [16], we build color and texton histograms for each geometric class (ground, vertical, porous, and sky). Specifically, for each color and texture sample, we weight its contribution to each histogram by the probability that it belongs to that geometric class. These eight histograms are compared with χ^2 distance after normalization.

+ code complexity :(

ImageNet Classification with Deep Convolutional Neural Networks

[Krizhevsky, Sutskever, Hinton, 2012]

“AlexNet”

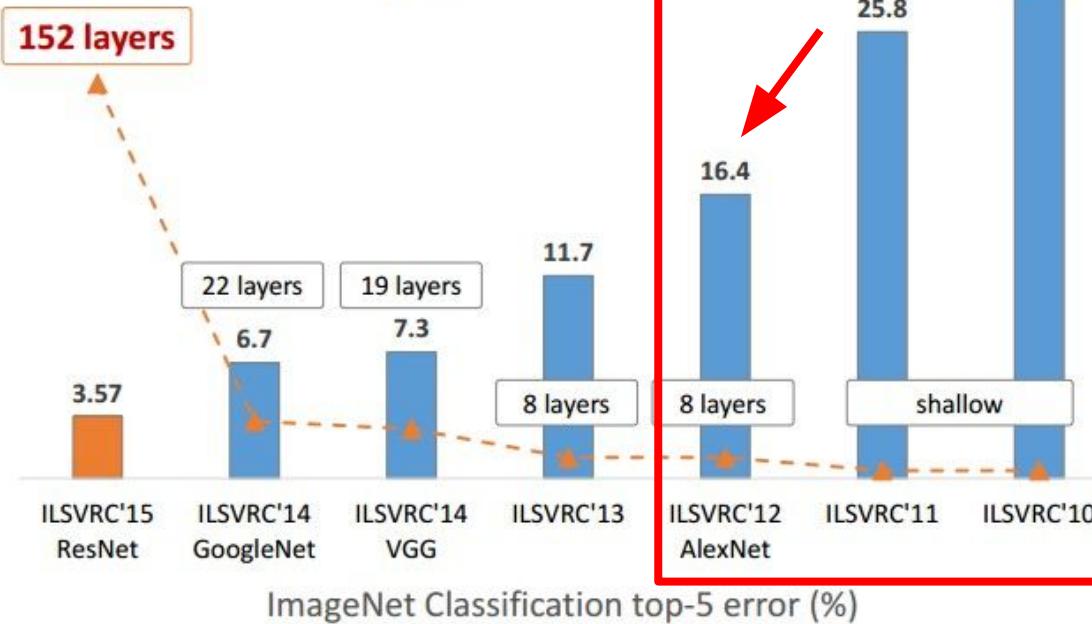


Deng et al.
Russakovsky et al.



NVIDIA et al.

Revolution of Depth



“What I learned from competing against a ConvNet on ImageNet” (karpathy.github.io)

Show answer Show google prediction

hotdog, hot dog, red hot

hotdog, hot dog, red hot

cheeseburger

GoogLeNet predictions:

hotdog, hot dog, red hot

ice cream, icecream

buckeye, horse chestnut, conker

French loaf

cheeseburger

consomme

snack food sandwich

hotdog, hot dog, red hot

hamburger, beefburger, burger

course entree, main course

plate

dessert, sweet, afters frozen dessert

“What I learned from competing against a ConvNet on ImageNet” (karpathy.github.io)



TLDR: Human accuracy is **somewhere 2-5%**.
(depending on how much training or how little life you have)

vector describing
various image statistics

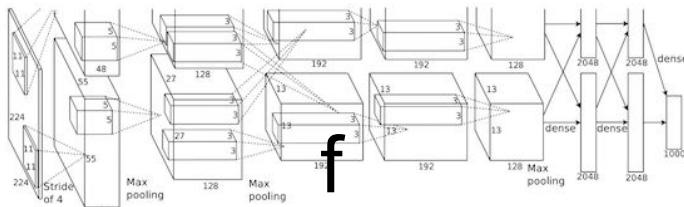


Feature
Extraction

f

← training

1000 numbers,
indicating class scores



← training

1000 numbers,
indicating class scores

layout.

GIST: The GIST descriptor [21] computes the output energy of a bank of 24 filters. The filters are Gabor-like filters tuned to 8 orientations at 4 different scales. The square output of each filter is then averaged on a 4×4 grid.

HOGx2: First, histogram of oriented edges (HOG) descriptors [4] are densely extracted on a regular grid at steps of 8 pixels. HOG features are computed using the code available online provided by [9], which gives a 31-dimension descriptor for each node of the grid. Then, 2×2 neighboring HOG descriptors are stacked together to form a descriptor with 124 dimensions. The stacked descriptors spatially overlap. This 2×2 neighbor stacking is important because the higher feature dimensionality provides more descriptive power. The descriptors are quantized into 300 visual words by k -means. With this visual word representation, three-level spatial histograms are computed on grids of 1×1 , 2×2 and 4×4 . Histogram intersection[17] is used to define the similarity of two histograms at the same pyramid level for two images. The kernel matrices at the three levels are normalized by their respective means, and linearly combined together using equal weights.

Dense SIFT: As with HOGx2, SIFT descriptors are densely extracted [17] using a flat rather than Gaussian window at two scales (4 and 8 pixel radii) on a regular grid at steps of 5 pixels. The three descriptors are stacked together for each HSV color channels, and quantized into 300 visual words by k -means, and spatial pyramid histograms are used as kernels[17].

4.1. Image Features and Kernels

We selected or designed several state-of-art features that are potentially useful for scene classification. GIST descriptors [21] are proposed specifically for scene recognition tasks. Dense SIFT features are also found to perform very well at the 15-category dataset [17]. We also evaluate sparse SIFTs as used in "Video Google" [27]. HOG features provide excellent performance for object and human recognition tasks [4, 9], so it is interesting to examine their utility for scene recognition. While SIFT is known to be very good at finding repeated image content, the self-similarity descriptor (SSIM) [26] relates images using their internal layout of local self-similarities. Unlike GIST, SIFT, and HOG, which are all local gradient-based approaches, SSIM may provide a distinct, complementary measure of scene layout that is somewhat appearance invariant. As a baseline, we also include Tiny Images [28], color histograms and straight line histograms. To make our color and texton histograms more invariant to scene layout, we also build histograms for specific geometric classes as determined by [13]. The geometric classification of a scene is then itself used as a feature, hopefully being invariant to appearance but responsive to

LBP: Local Binary Patterns (LBP) [20] is a powerful texture feature based on occurrence histogram of local binary patterns. We can regard the scene recognition as a texture classification problem of 2D images, and therefore apply this model to our problem. We also try the rotation invariant extension version [2] of LBP to examine whether rotation invariance is suitable for scene recognition.

Sparse SIFT histograms: As in "Video Google" [27], we build SIFT features at Hessian-affine and MSER [19] interest points. We cluster each set of SIFTS, independently, into dictionaries of 1,000 visual words using k -means. An image is represented by two histograms counting the number of sparse SIFTS that fall into each bin. An image is represented by two 1,000 dimension histograms where each SIFT is soft-assigned, in [22], to its nearest cluster centers. Kernels are computed with χ^2 distance.

SSIM: Self-similarity descriptors [26] are computed on a regular grid at steps of five pixels. Each descriptor is obtained by computing the correlation map of a patch of 5×5 in a window with radius equal to 40 pixels, then quantizing it in 3 radial bins and 10 angular bins, obtaining 30 dimensional descriptor vectors. The descriptors are then quantized into 300 visual words by k -means and we use χ^2 distance on spatial histograms for the kernels.

Tiny Images: The most trivial way to match scenes is to compare them directly in color image space. Reducing the image dimensions drastically makes this approach more computationally feasible and less sensitive to exact alignment.

This method of image matching has been examined thoroughly by Torralba et al.[28] for the purpose of object recognition and scene classification.

Line Features: We detect straight lines from Canny edges using the method described in Video Compass [15]. For each image we build two histograms based on the statistics of detected lines- one with bins corresponding to line angles and one with bins corresponding to line lengths. We use an RBF kernel to compare these unnormalized histograms. This feature was used in [11].

Texton Histograms: We build a 512 entry universal texture dictionary [18] by clustering responses to a bank of filters with 8 orientations, 2 scales, and 2 elongations. For each image we then build a 512-dimensional histogram by assigning each pixel's set of filtered responses to the nearest texture dictionary entry. We compute kernels from normalized χ^2 distances.

Color Histograms: We build joint histograms of color in CIE $L^*a^*b^*$ color space for each image. Our histograms have 4, 14, and 14 bins in L , a , and b respectively for a total of 784 dimensions. We compute distances between these histograms using χ^2 distance on the normalized histograms.

Geometric Probability Map: We compute the geometric class probabilities for image regions using the method of Hoiem et al. [13]. We use only the ground, vertical, porous, and sky classes because they are more reliably classified. We reduce the probability maps for each class to 8×8 and use an RBF kernel. This feature was used in [11].

Geometry Specific Histograms: Inspired by "Illumination Context" [16], we build color and texton histograms for each geometric class (ground, vertical, porous, and sky). Specifically, for each color and texture sample, we weight its contribution to each histogram by the probability that it belongs to that geometric class. These eight histograms are compared with χ^2 distance after normalization.

"Run the image through 20 layers of 3×3 convolutions and train the filters with SGD."*

* to the first order

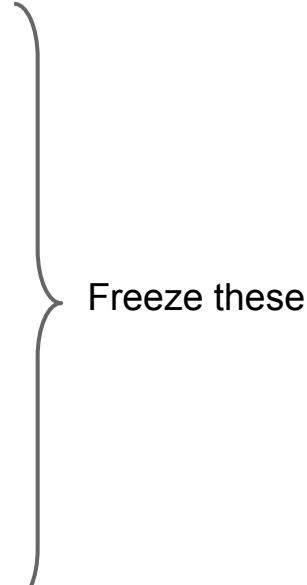
Transfer Learning



1. Train on Imagenet



2. Small dataset:
feature extractor



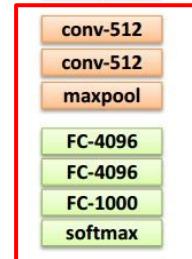
Train this



3. Medium dataset:
finetuning

more data = retrain more of the network (or all of it)

Freeze these



Train this

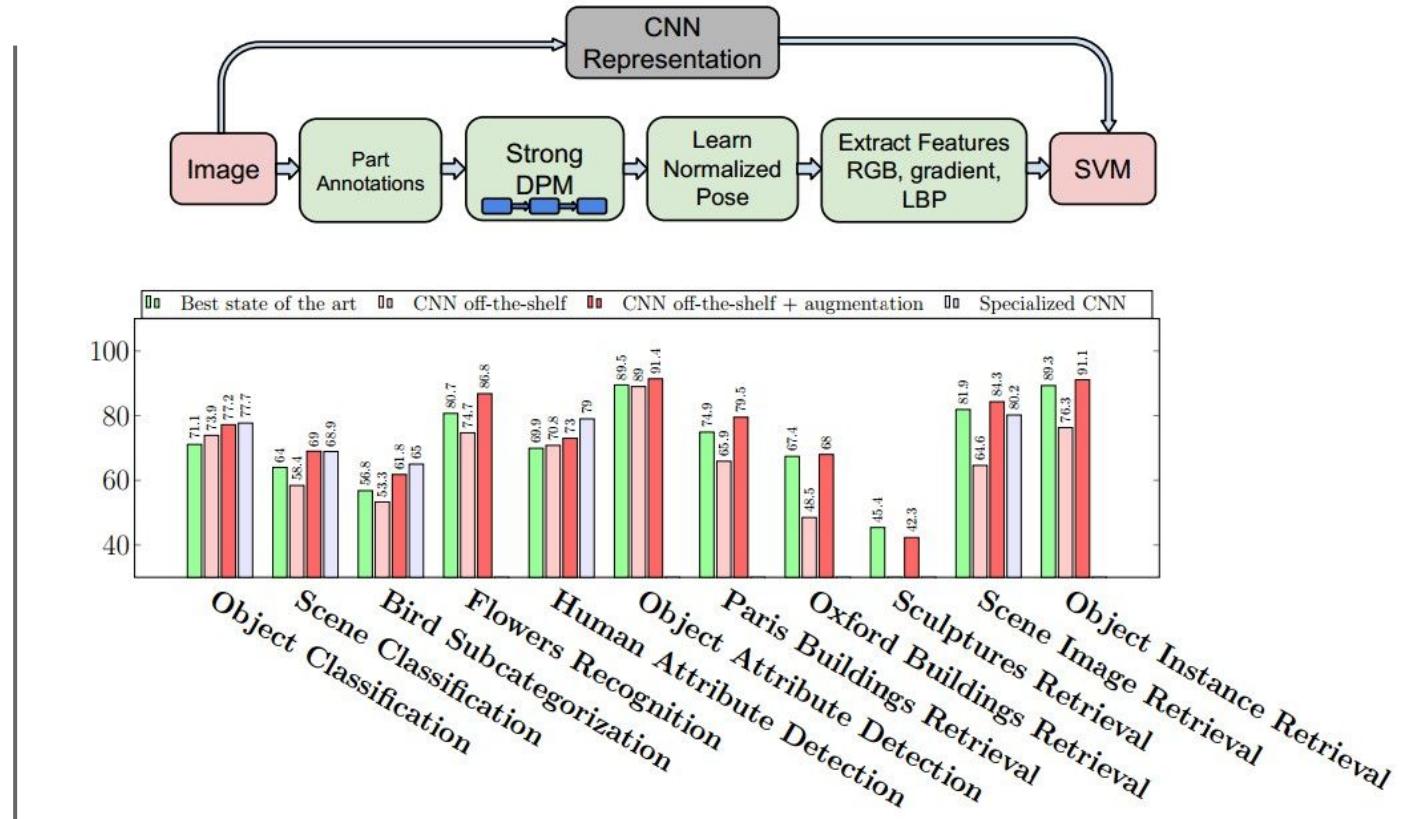
Transfer Learning

CNN Features off-the-shelf: an Astounding Baseline for Recognition
[Razavian et al, 2014]

DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition
[Donahue*, Jia*, et al., 2013]

	DeCAF ₆	DeCAF ₇
LogReg	40.94 ± 0.3	40.84 ± 0.3
SVM	39.36 ± 0.3	40.66 ± 0.3

Xiao et al. (2010) 38.0



The power is easily accessible.

e.g. with keras.io

```
sudo pip install keras
```

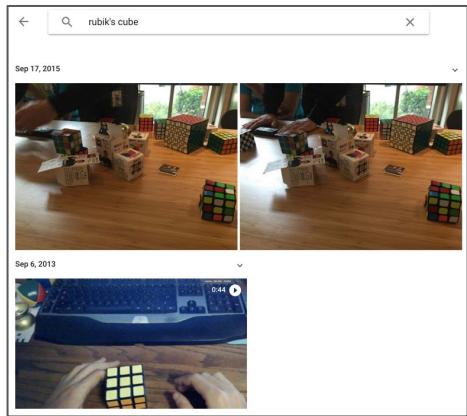
```
from keras.applications.resnet50 import ResNet50
from keras.preprocessing import image
from keras.applications.resnet50 import preprocess_input, decode_predictions

model = ResNet50(weights='imagenet')

img_path = 'elephant.jpg'
img = image.load_img(img_path, target_size=(224, 224))
x = image.img_to_array(img)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)

preds = model.predict(x)
print('Predicted:', decode_predictions(preds))
# print: [[u'n02504458', u'African_elephant']]
```

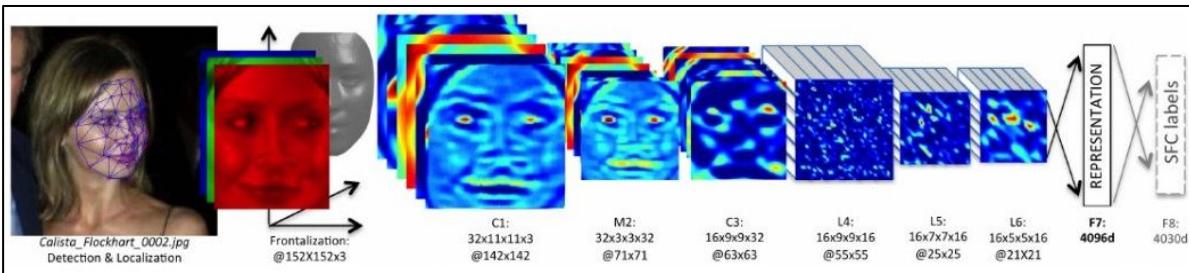
ConvNets are everywhere...



e.g. Google Photos search



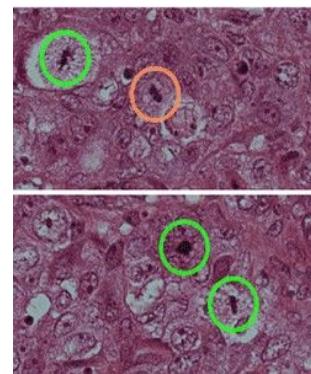
[Goodfellow et al. 2014]



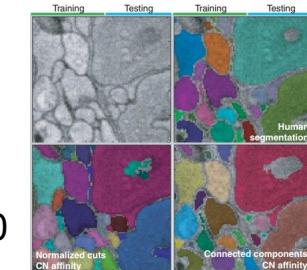
Face Verification, Taigman et al. 2014 (FAIR)



Self-driving cars



Ciresan et al. 2013



Turaga et al 2010

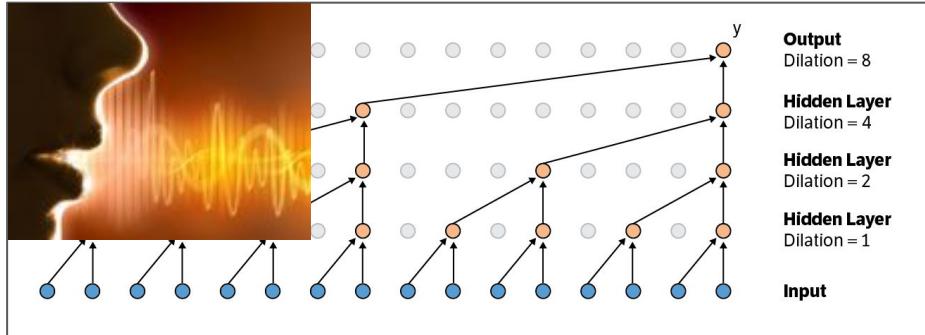
ConvNets are everywhere...



Whale recognition, Kaggle Challenge



Satellite image analysis
Mnih and Hinton, 2010



WaveNet, van den Oord et al. 2016

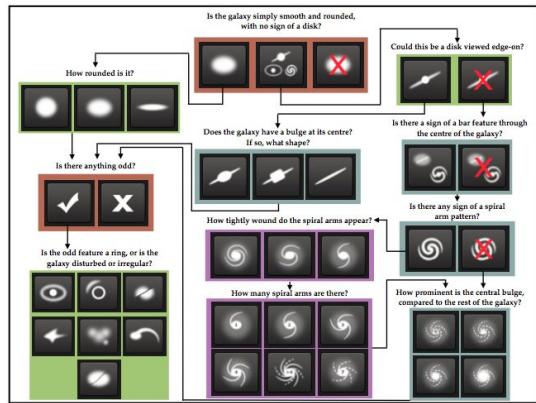


Figure 1. Flowchart of the classification tasks for GZ2, beginning at the top centre. Tasks are colour-coded by their relative depths in the decision tree. Tasks outlined in brown are asked of every galaxy. Tasks outlined in green, blue, and purple are (respectively) one, two or three steps below branching points in the decision tree. Table 2 describes the responses that correspond to the icons in this diagram.

Galaxy Challenge Dielman et al. 2015

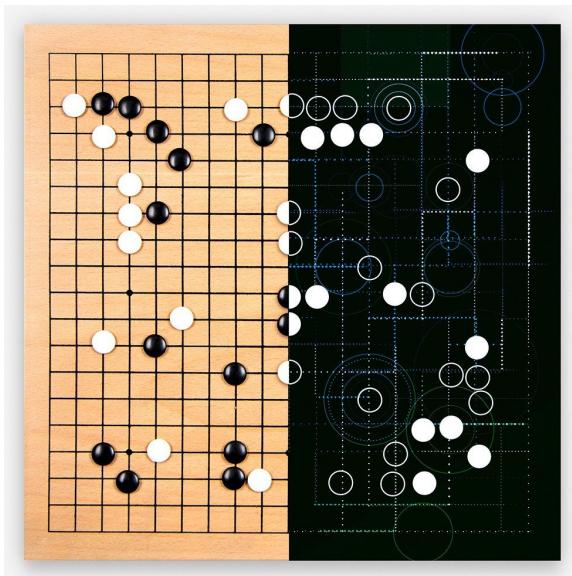


Image captioning, Vinyals et al. 2015

ConvNets are everywhere...



ATARI game playing, Mnih 2013



VizDoom

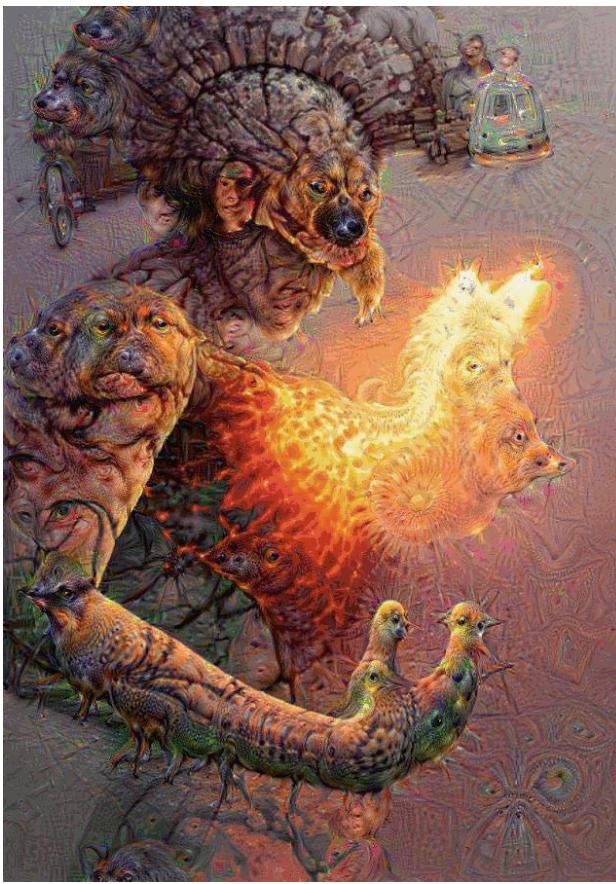
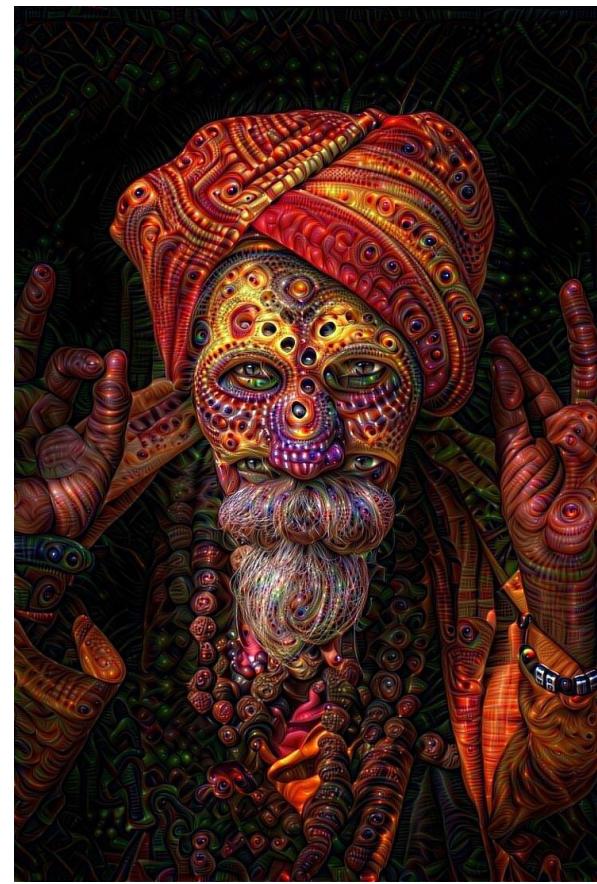


StarCraft

AlphaGo, Silver et al 2016

....

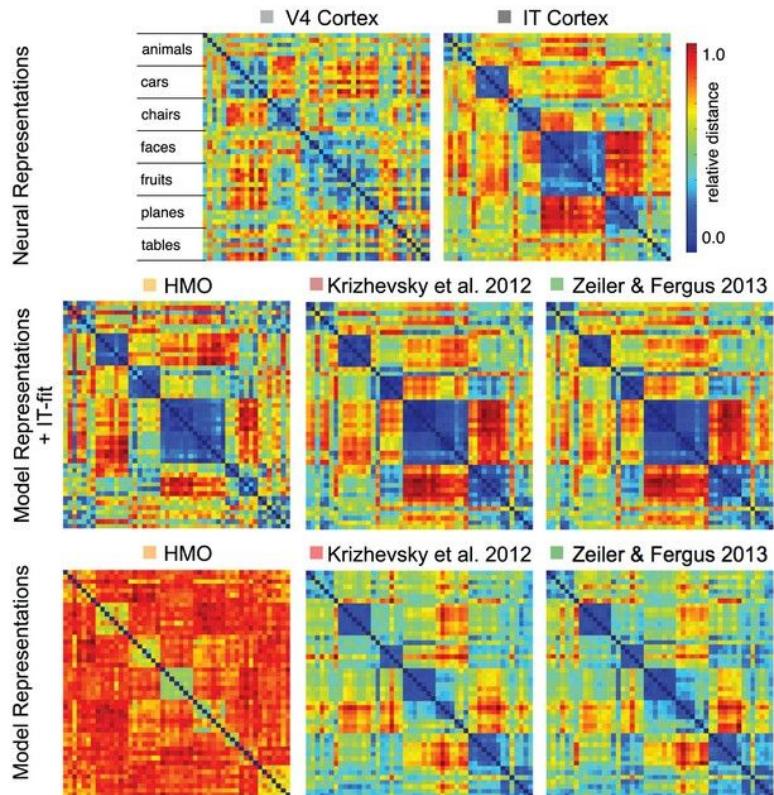
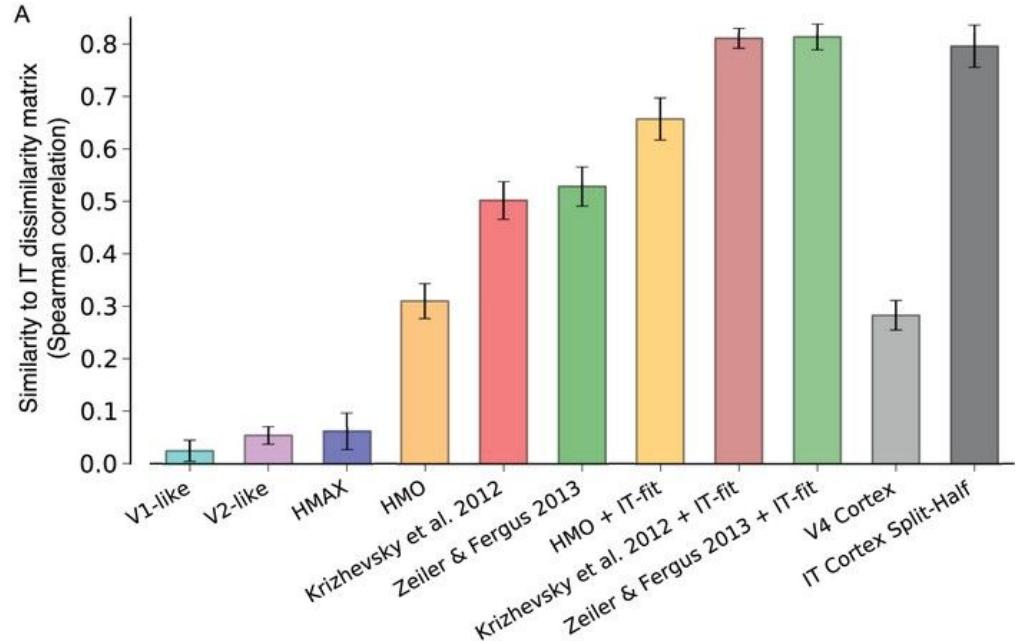
ConvNets are everywhere...



NeuralStyle, Gatys et al. 2015
deepart.io, Prisma, etc.

DeepDream [reddit.com/r/deepdream](https://www.reddit.com/r/deepdream)

ConvNets \longleftrightarrow Visual Cortex

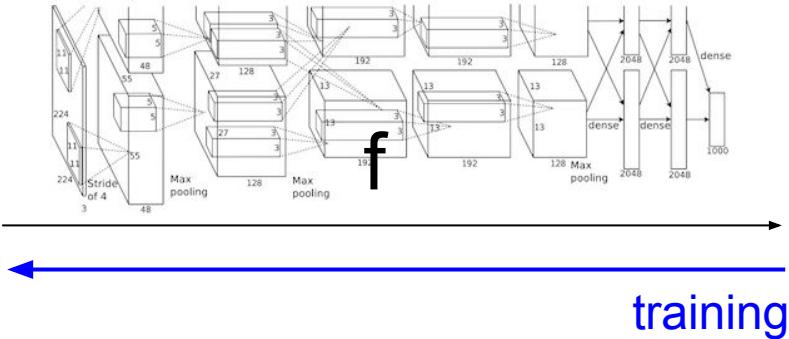


*Deep Neural Networks Rival the Representation of Primate IT Cortex for Core Visual Object Recognition
[Cadieu et al., 2014]*

</history>
</context>

Convolutional Neural Networks

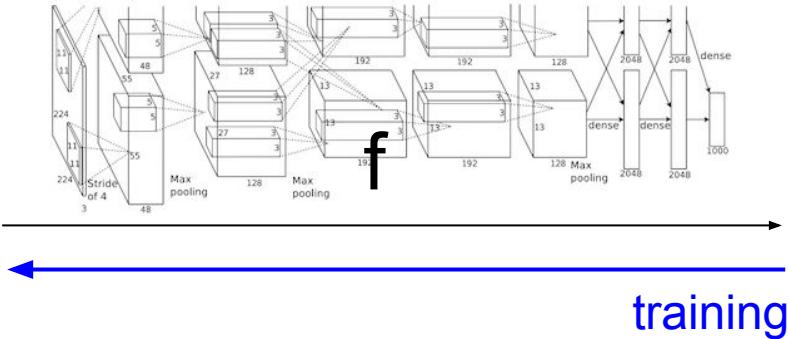
<explanation>



1000 numbers,
indicating class scores

Only two basic operations are involved throughout:

1. Dot products $w^T x$
2. Max operations $\max(\cdot)$



1000 numbers,
indicating class scores

Only two basic operations are involved throughout:

1. Dot products $w^T x$
2. Max operations $\max(\cdot)$

parameters
(~10M of them)

preview:

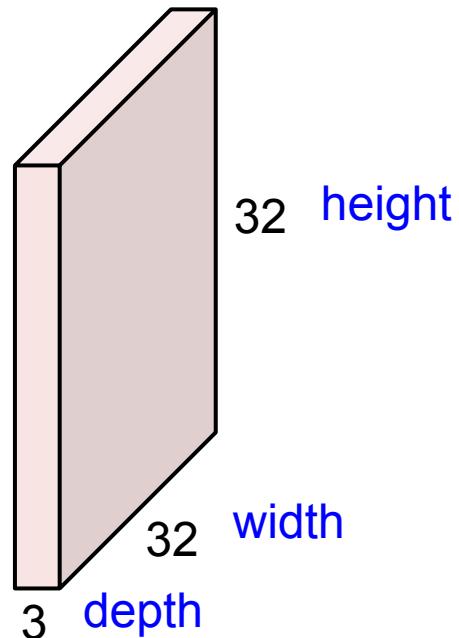


e.g. 200K numbers

→ e.g. 10 numbers

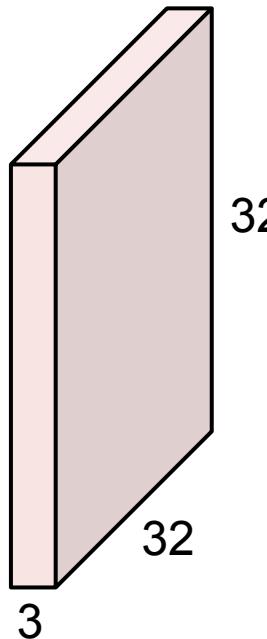
Convolution Layer

32x32x3 image

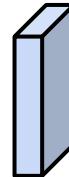


Convolution Layer

32x32x3 image



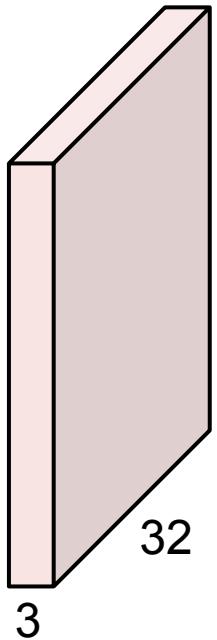
5x5x3 filter



Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

Convolution Layer

32x32x3 image



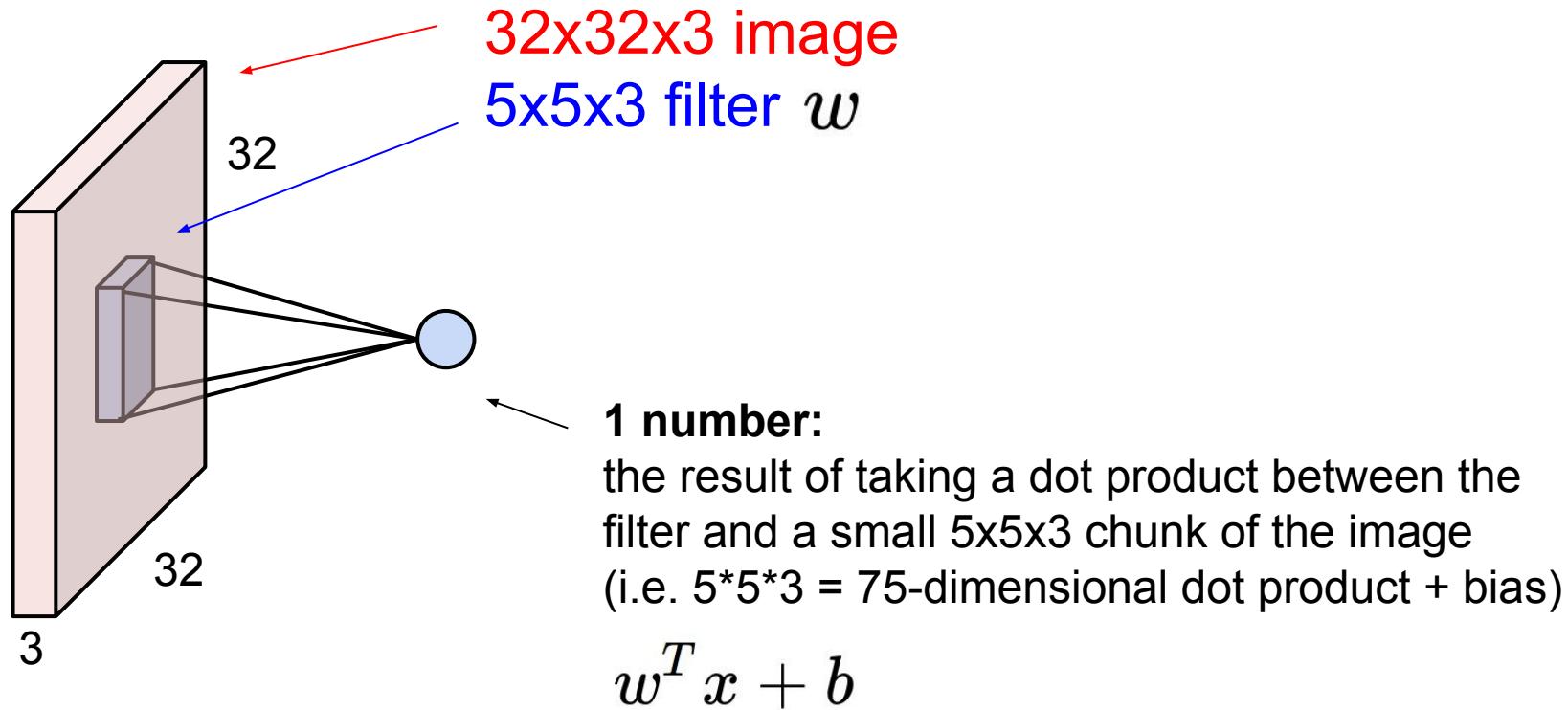
5x5x3 filter



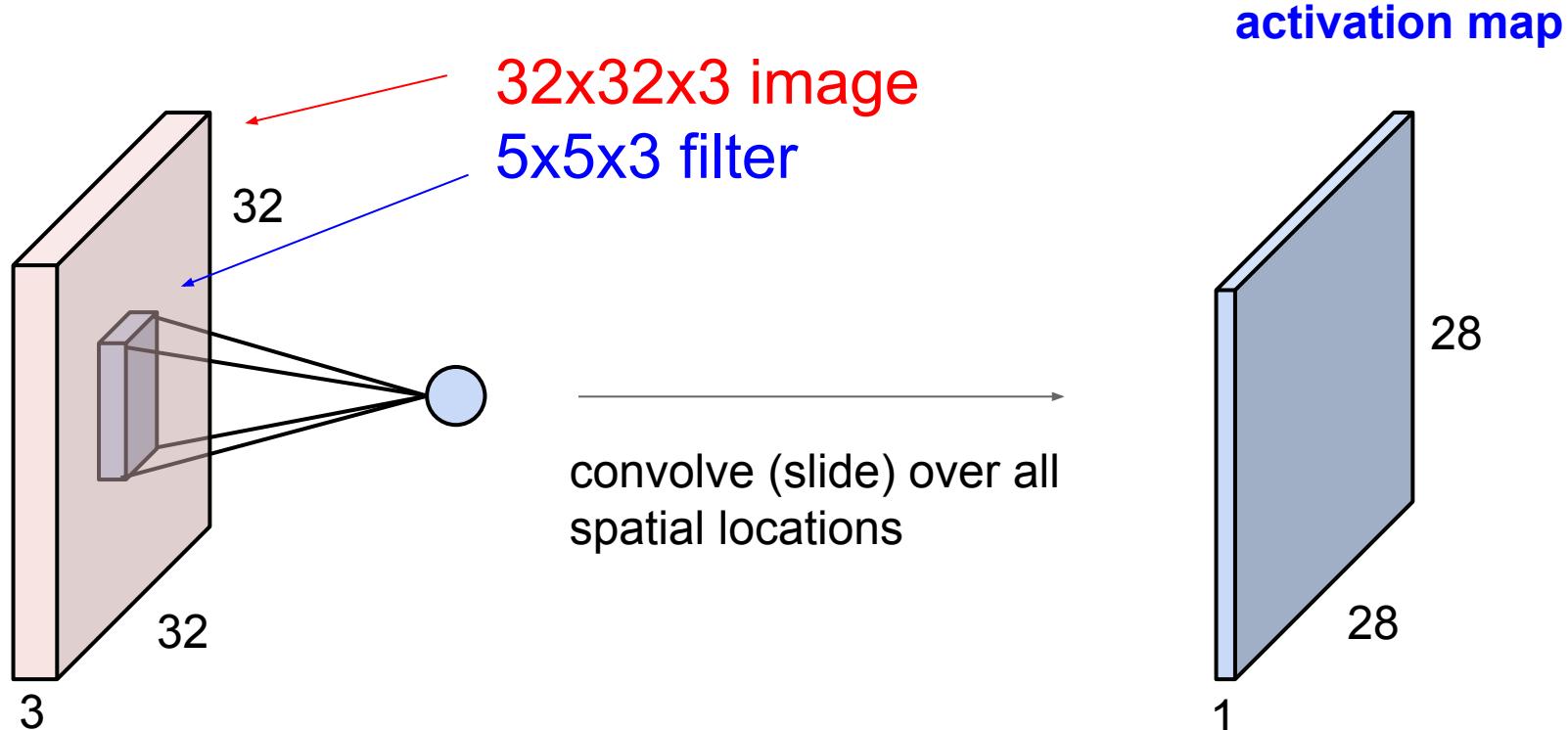
Filters always extend the full depth of the input volume

Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

Convolution Layer

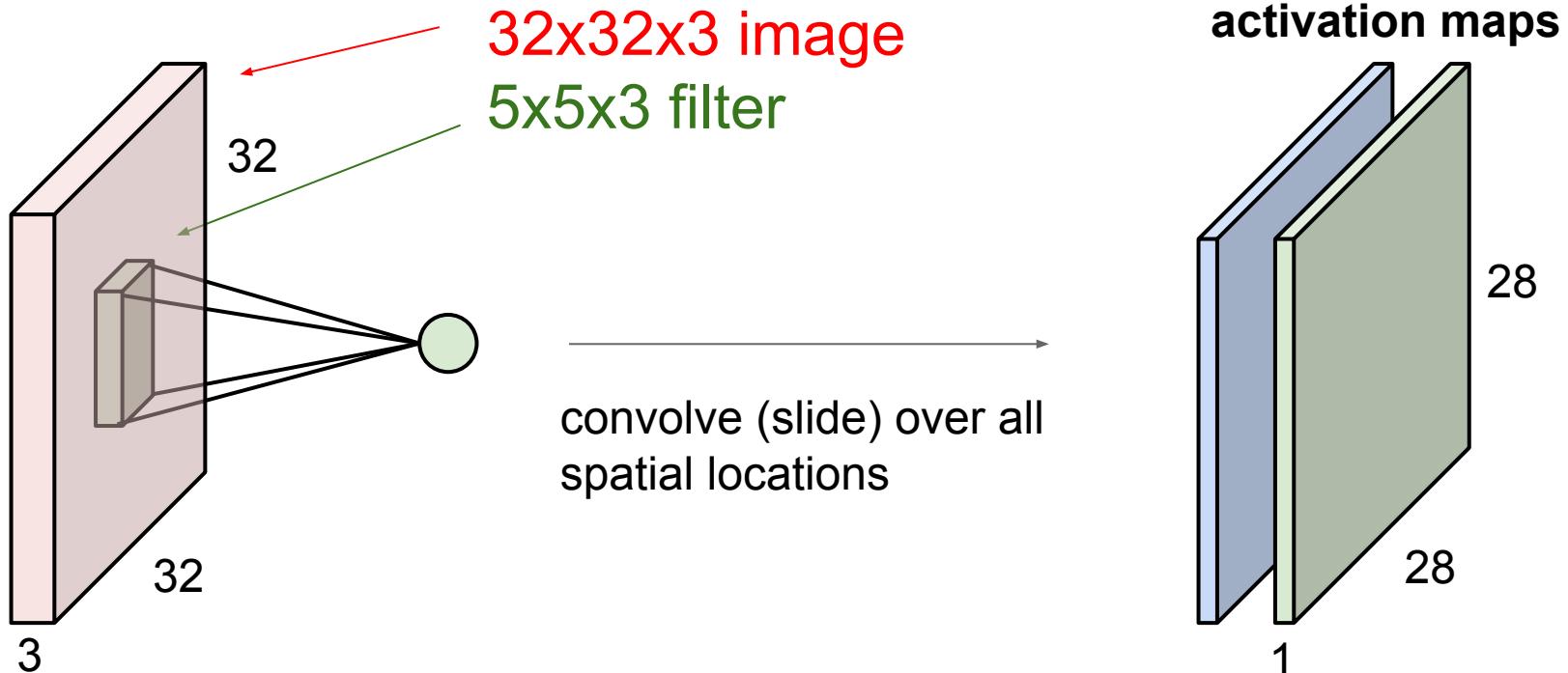


Convolution Layer

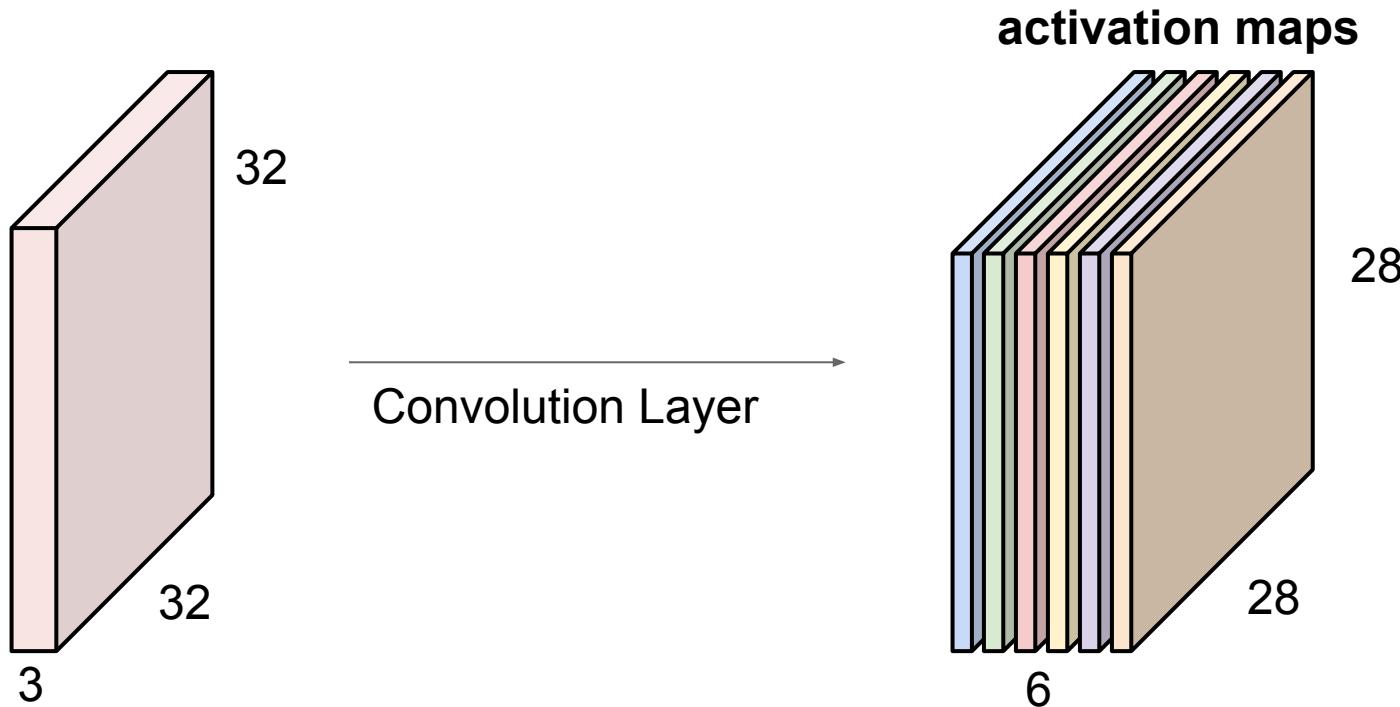


Convolution Layer

consider a second, green filter

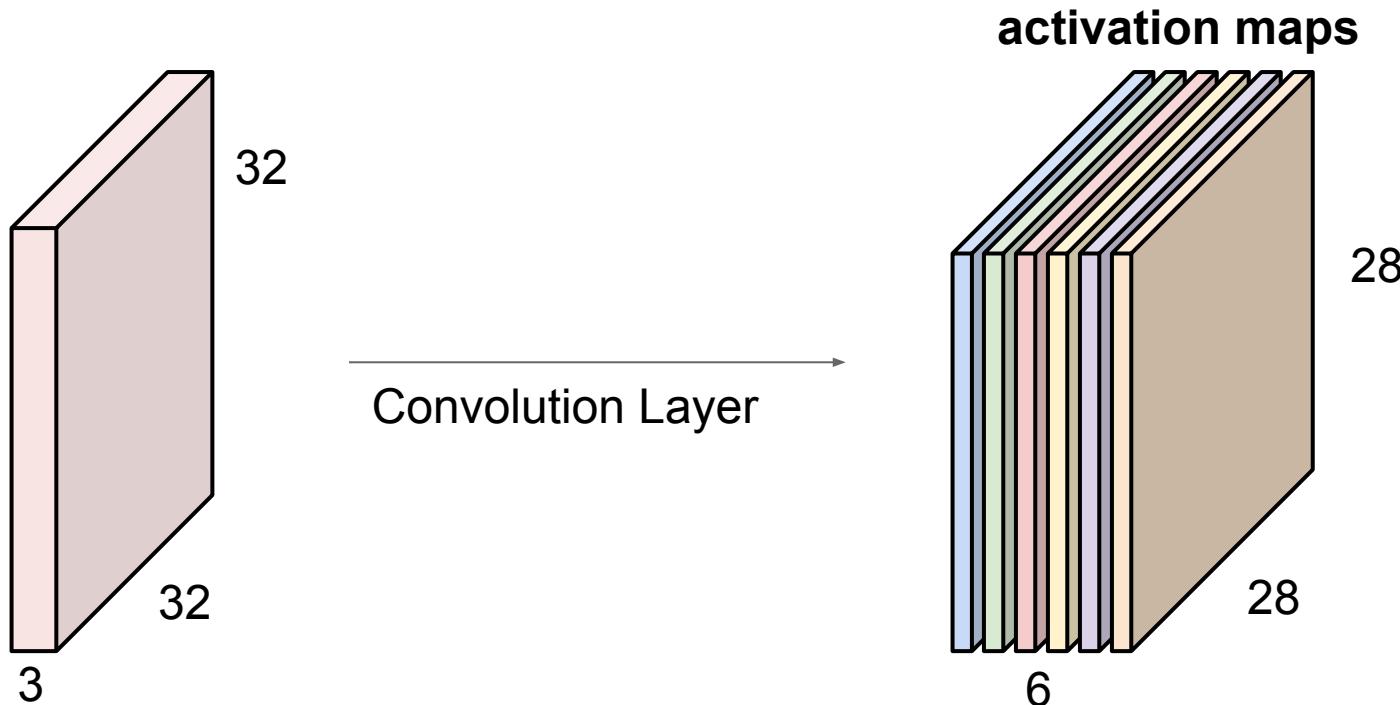


For example, if we had 6 5×5 filters, we'll get 6 separate activation maps:



We stack these up to get a “new image” of size $28 \times 28 \times 6$!

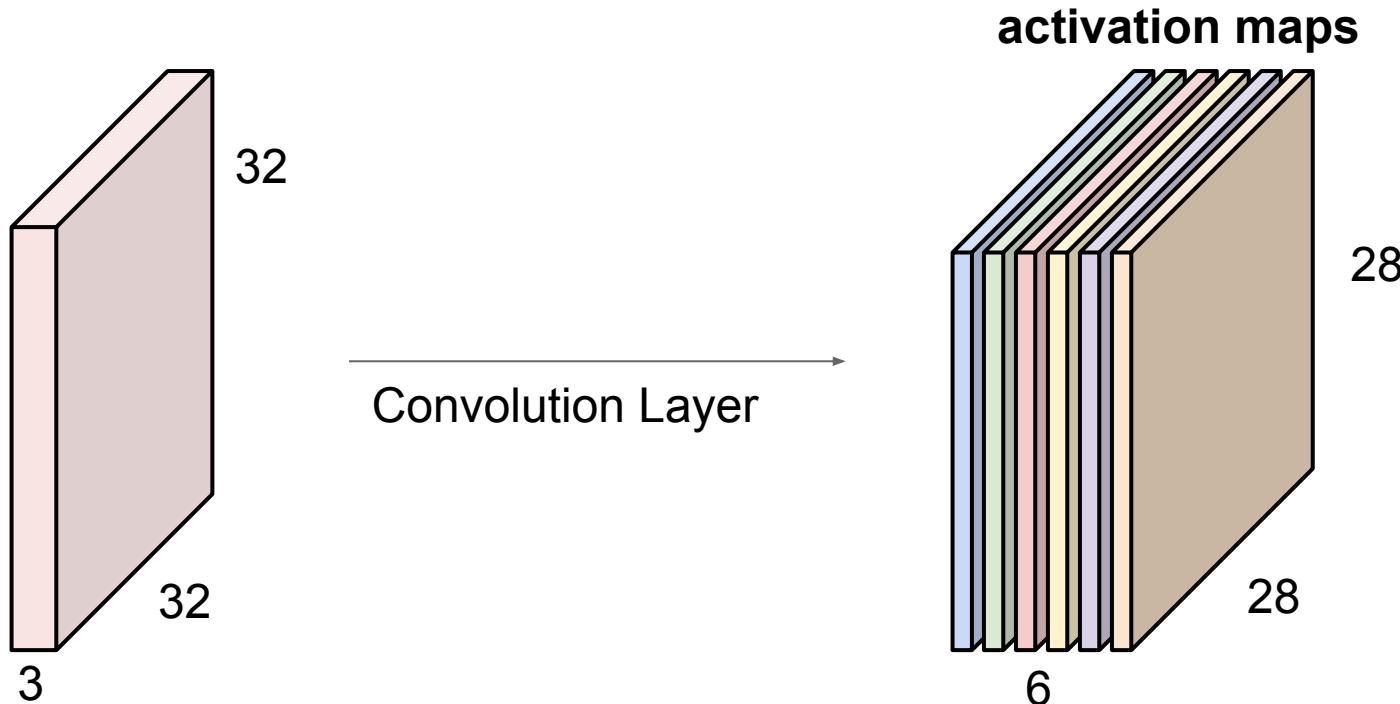
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We processed [32x32x3] volume into [28x28x6] volume.

Q: how many parameters would this be if we used a fully connected layer instead?

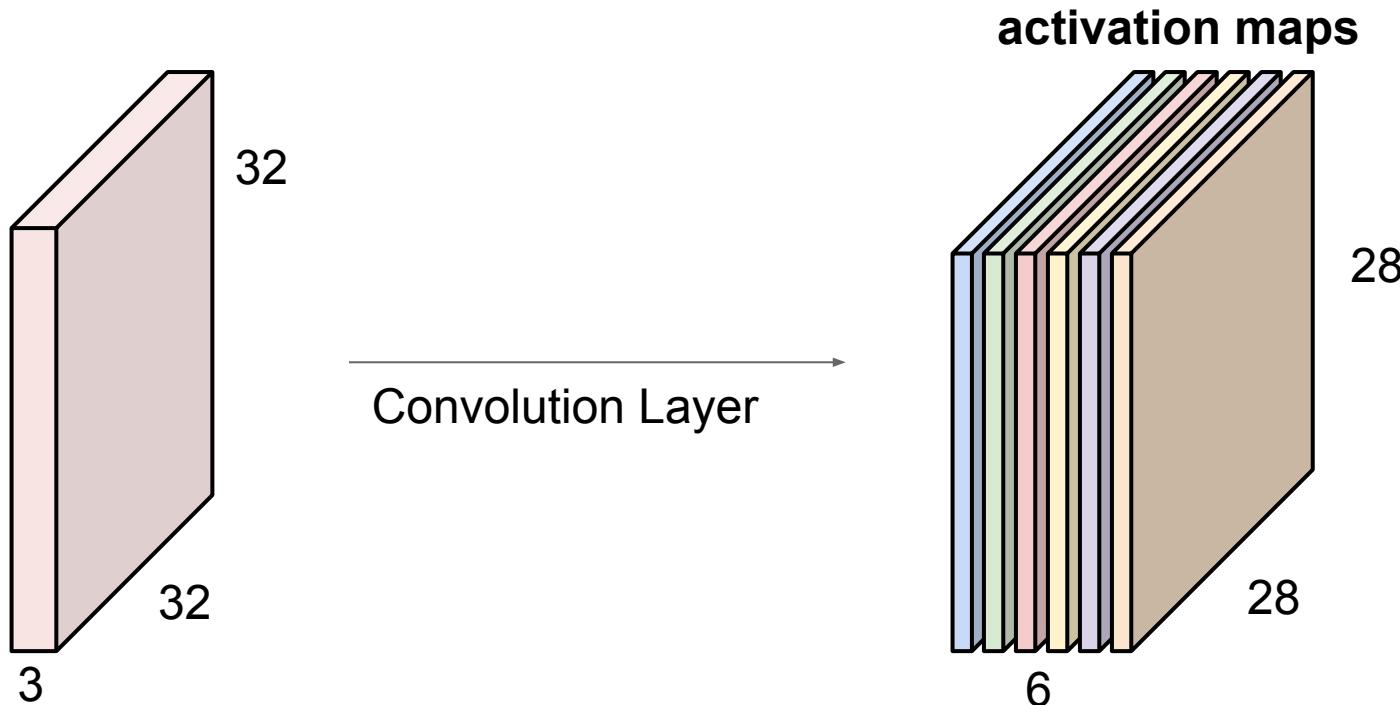
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We processed [32x32x3] volume into [28x28x6] volume.

Q: how many parameters would this be if we used a fully connected layer instead?
A: $(32*32*3)*(28*28*6) = 14.5M$ parameters, ~14.5M multiplies

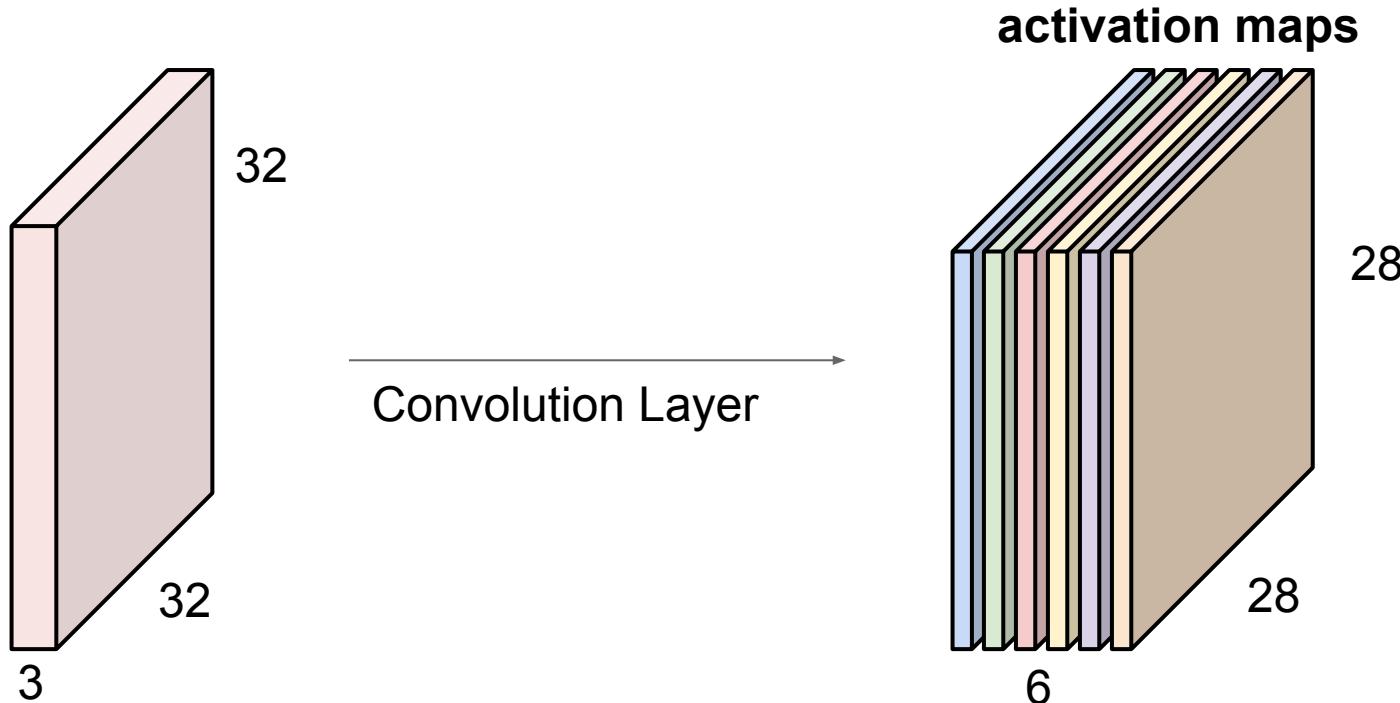
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We processed [32x32x3] volume into [28x28x6] volume.

Q: how many parameters are used instead?

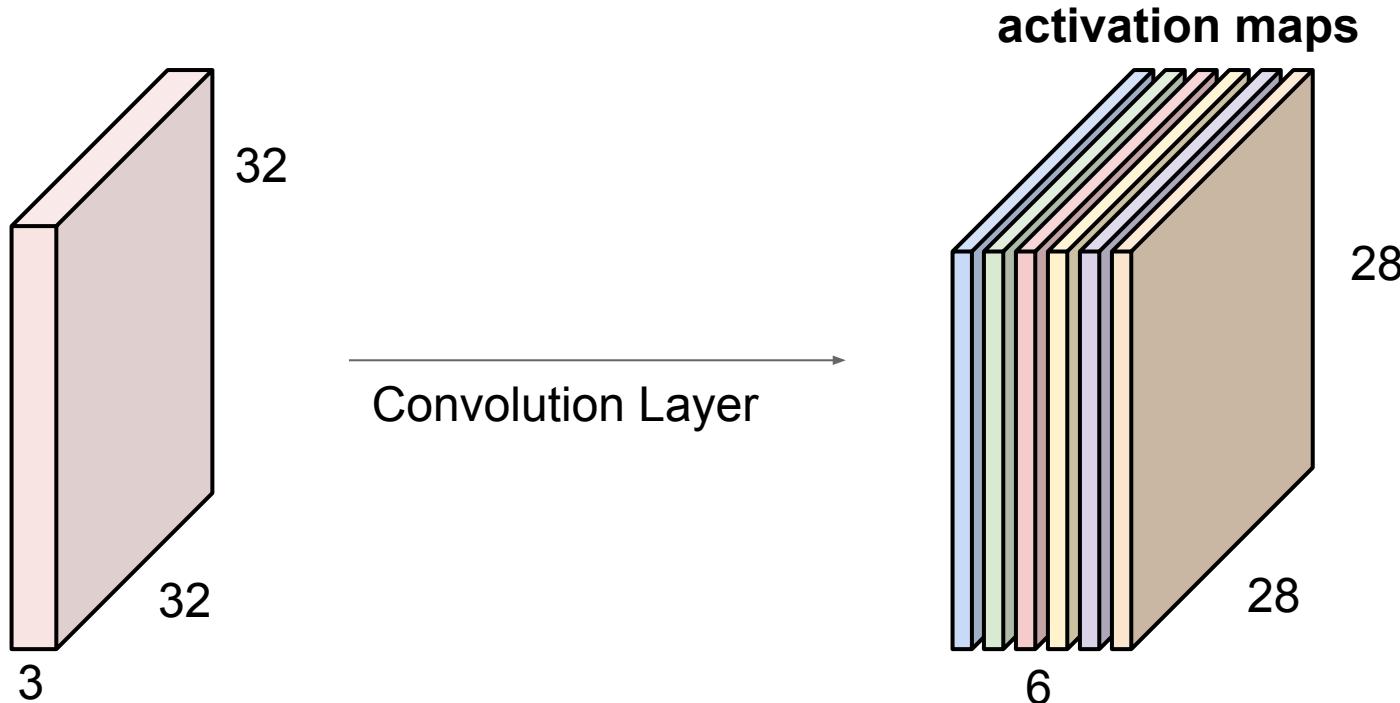
For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We processed [32x32x3] volume into [28x28x6] volume.

Q: how many parameters are used instead? --- And how many multiplies?
A: $(5 \times 5 \times 3) \times 6 = 450$ parameters

For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



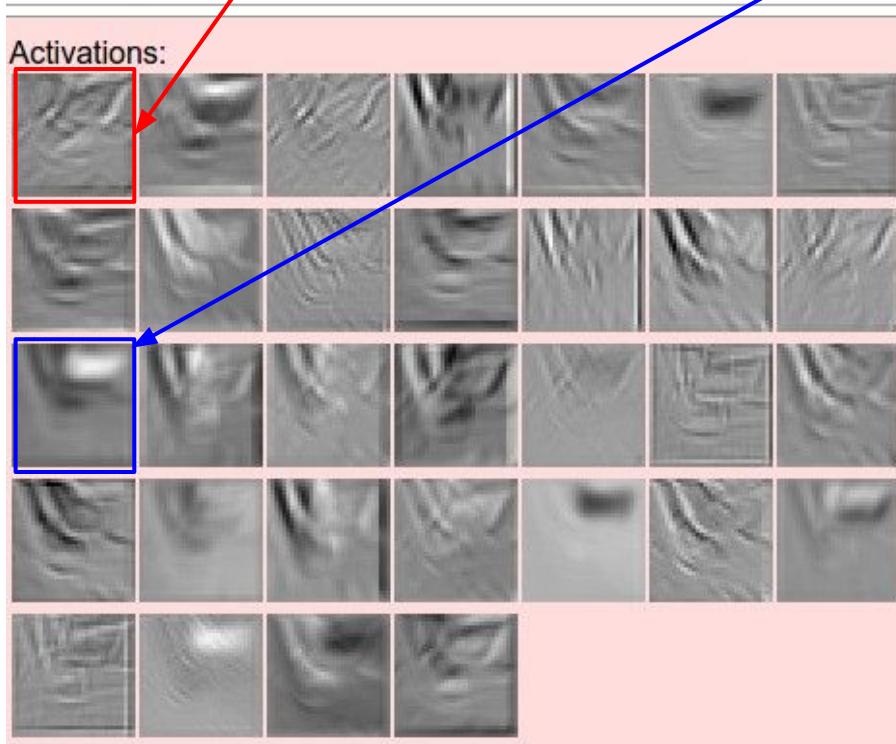
We processed [32x32x3] volume into [28x28x6] volume.

Q: how many parameters are used instead?

A: $(5 \times 5 \times 3) \times 6 = 450$ parameters, $(5 \times 5 \times 3) \times (28 \times 28 \times 6) = \sim 350K$ multiplies



one filter =>
one activation map



example 5x5 filters
(32 total)

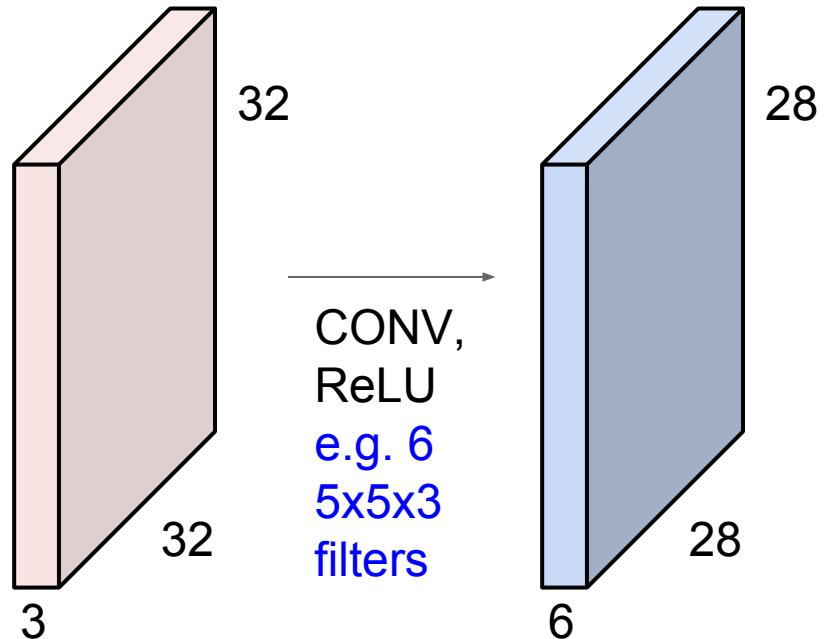
We call the layer convolutional
because it is related to convolution
of two signals:

$$f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1, n_2] \cdot g[x - n_1, y - n_2]$$

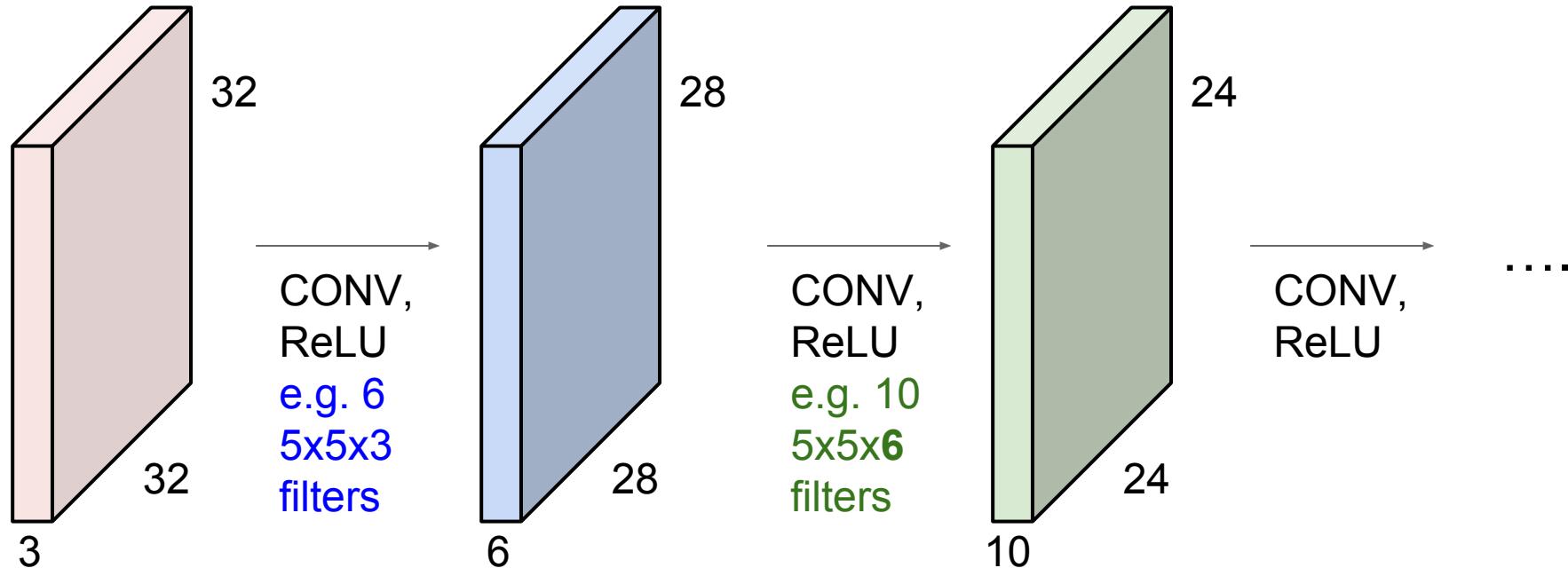


elementwise multiplication and sum of
a filter and the signal (image)

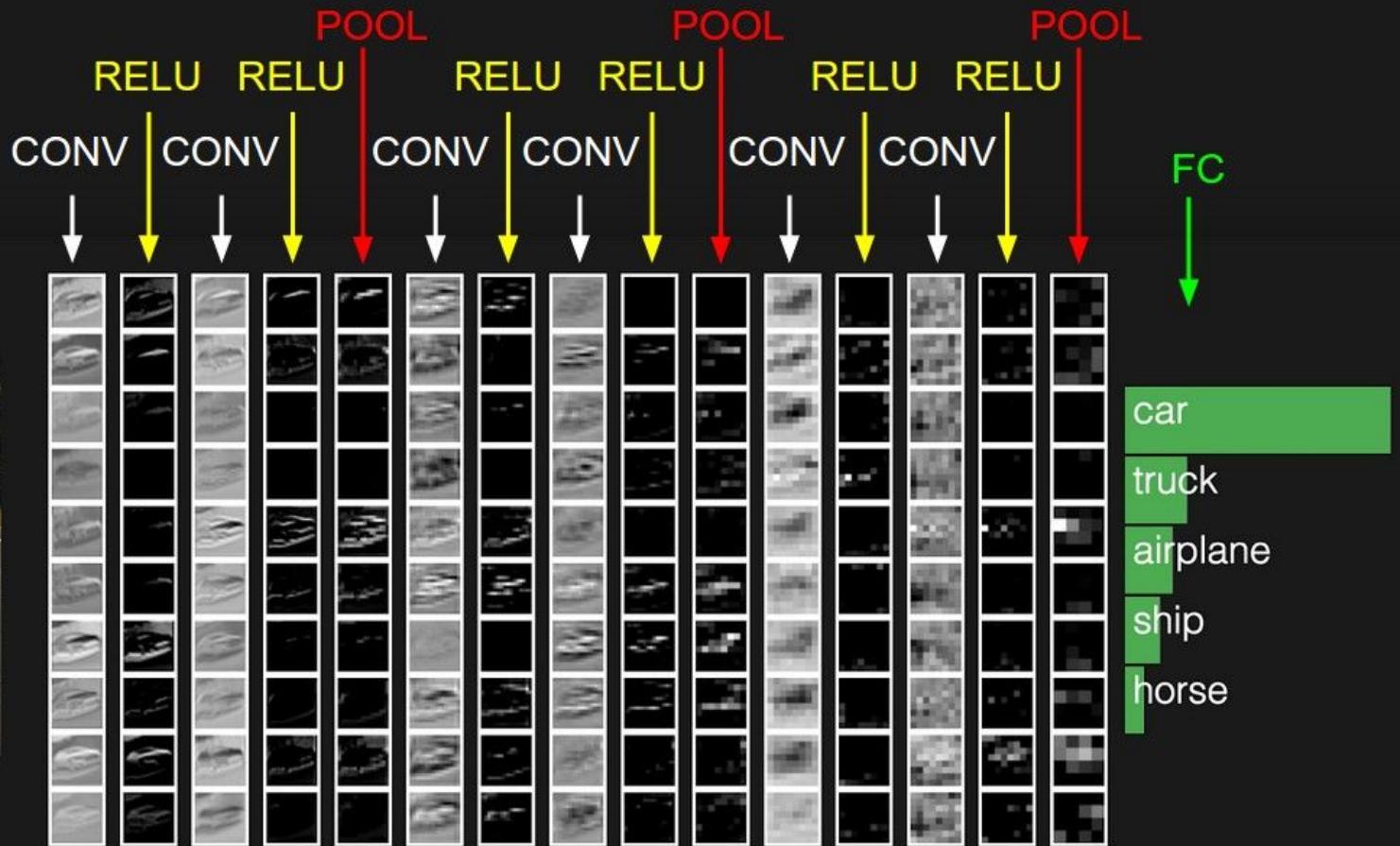
Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions



Preview: ConvNet is a sequence of Convolutional Layers, interspersed with activation functions

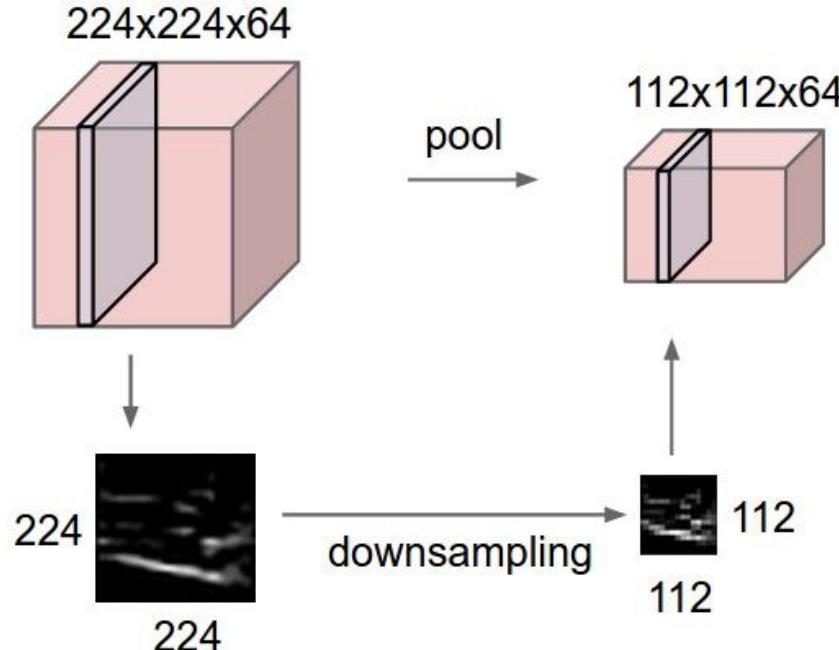


two more layers to go: POOL/FC



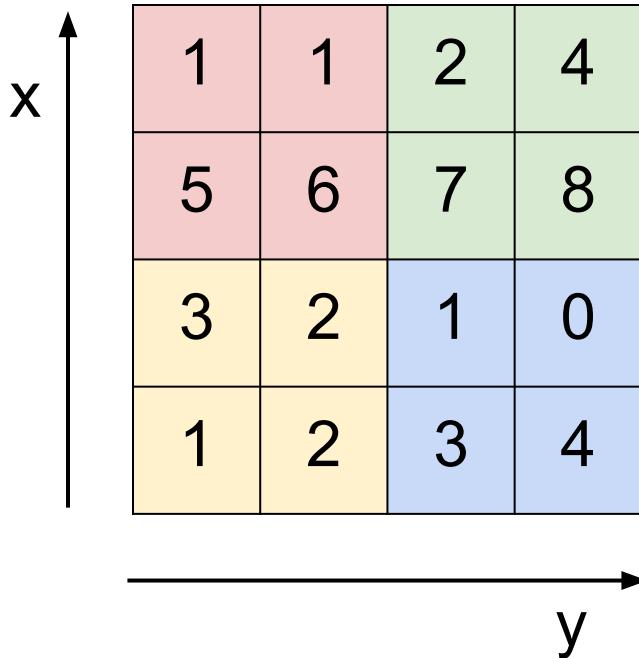
Pooling layer

- makes the representations smaller and more manageable
- operates over each activation map independently:



MAX POOLING

Single depth slice



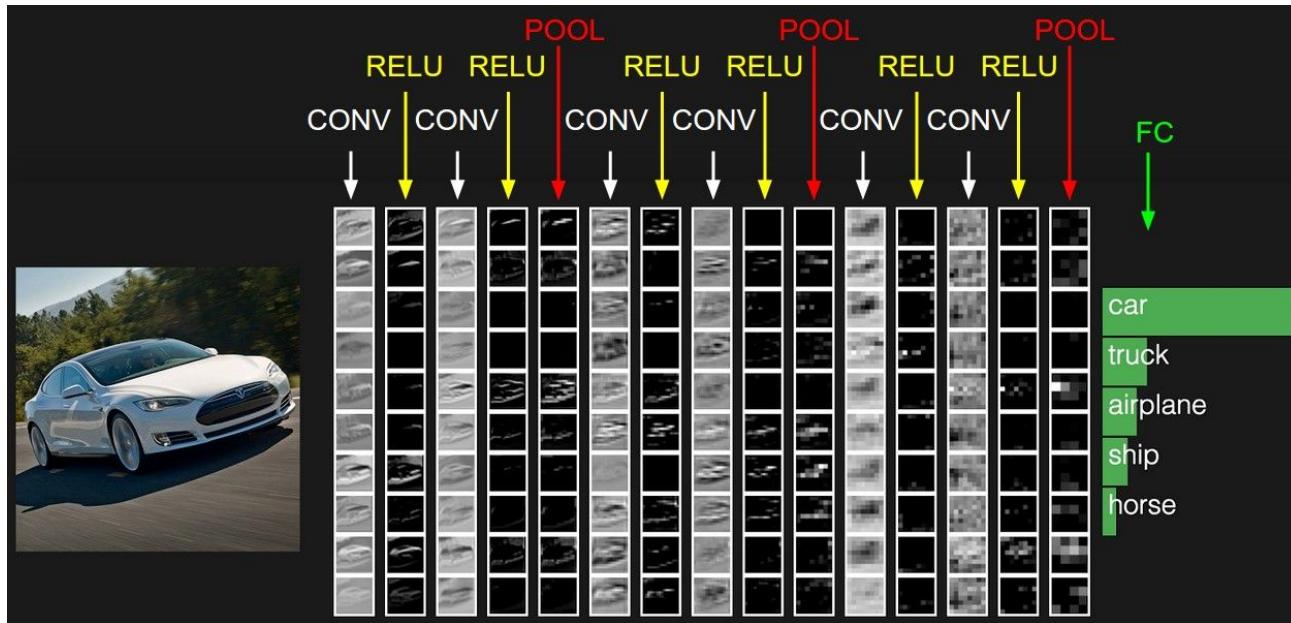
max pool with 2x2 filters
and stride 2

A 2x2 grid representing the output of the max pooling operation. It contains the maximum values from each 2x2 receptive field in the input tensor.

6	8
3	4

Fully Connected Layer (FC layer)

- Contains neurons that connect to the entire input volume, as in ordinary Neural Networks



[ConvNetJS demo: training on CIFAR-10]

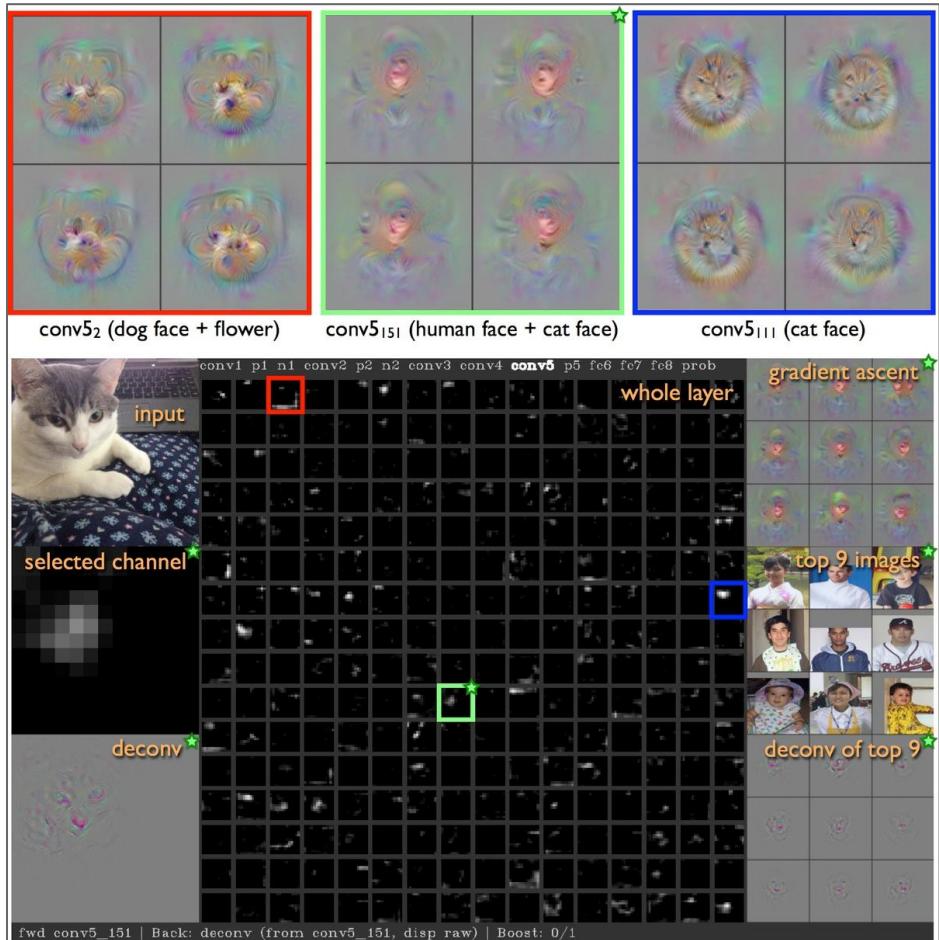
<http://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html>

Visualizing Activations

<http://yosinski.com/deepvis>

YouTube video

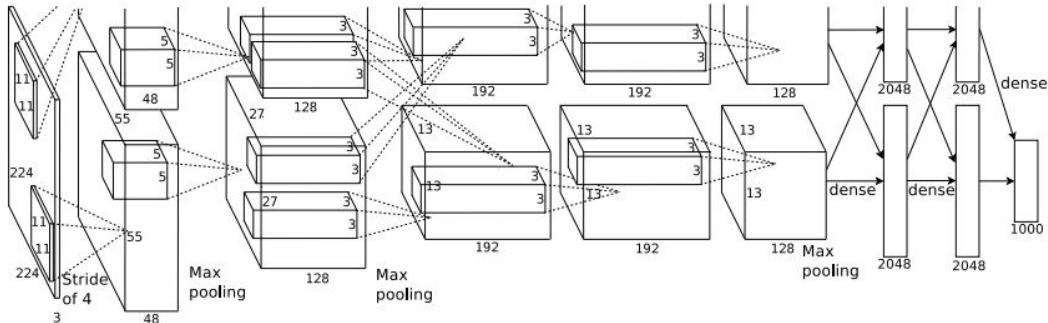
<https://www.youtube.com/watch?v=AgkfIQ4IGaM>
(4min)



Convolutional Neural Networks: Case Study

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

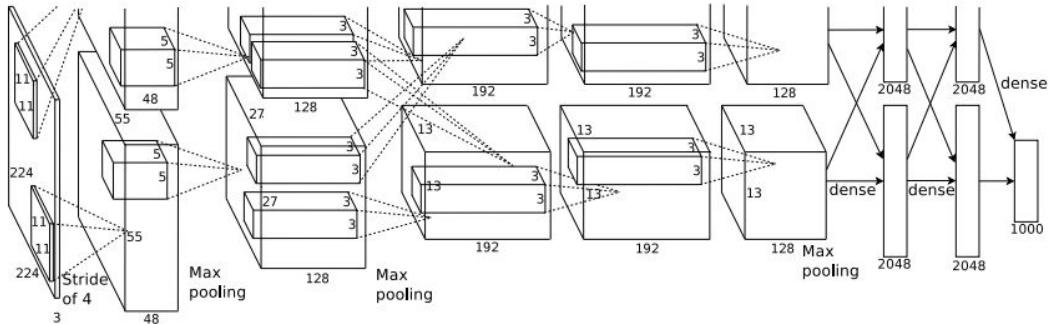
First layer (CONV1): 96 11x11 filters applied at stride 4

=>

Q: what is the output volume size? Hint: $(227-11)/4+1 = 55$

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

First layer (CONV1): 96 11x11 filters applied at stride 4

=>

Output volume **[55x55x96]**

Q: What is the total number of parameters in this layer?

Case Study: AlexNet

[Krizhevsky et al. 2012]

Input: 227x227x3 images

First layer (CONV1): 96 11x11 filters applied at stride 4

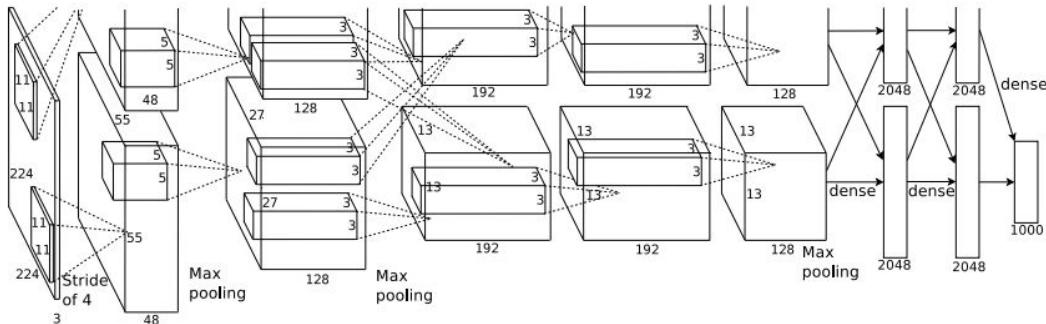
=>

Output volume **[55x55x96]**

Parameters: $(11 \times 11 \times 3) \times 96 = 35K$

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

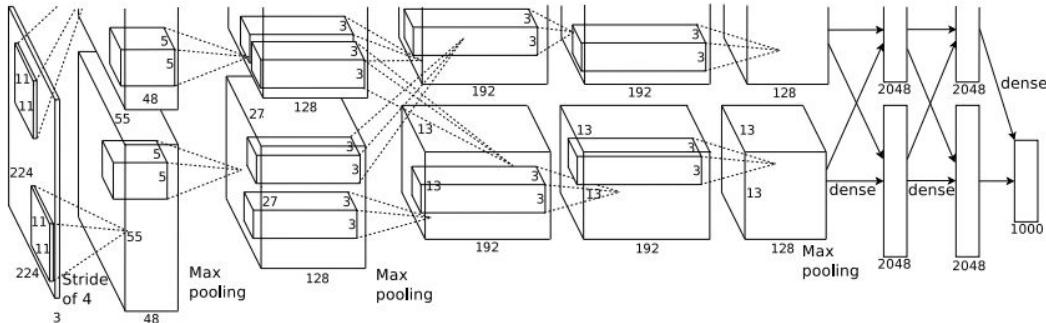
After CONV1: 55x55x96

Second layer (POOL1): 3x3 filters applied at stride 2

Q: what is the output volume size? Hint: $(55-3)/2+1 = 27$

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

After CONV1: 55x55x96

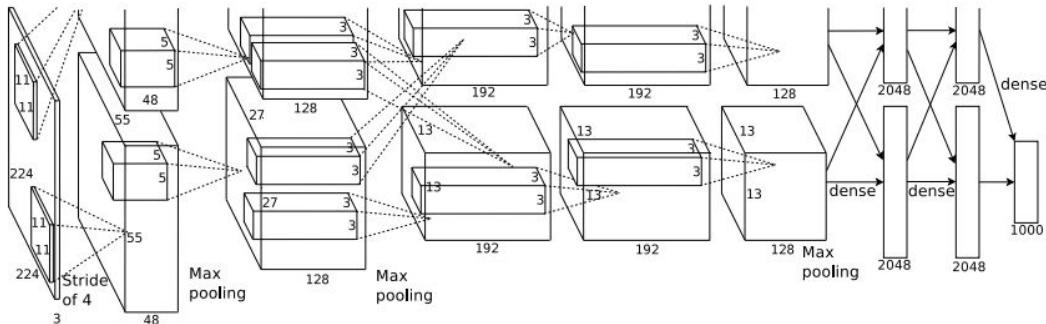
Second layer (POOL1): 3x3 filters applied at stride 2

Output volume: 27x27x96

Q: what is the number of parameters in this layer?

Case Study: AlexNet

[Krizhevsky et al. 2012]



Input: 227x227x3 images

After CONV1: 55x55x96

Second layer (POOL1): 3x3 filters applied at stride 2

Output volume: 27x27x96

Parameters: 0!

Case Study: AlexNet

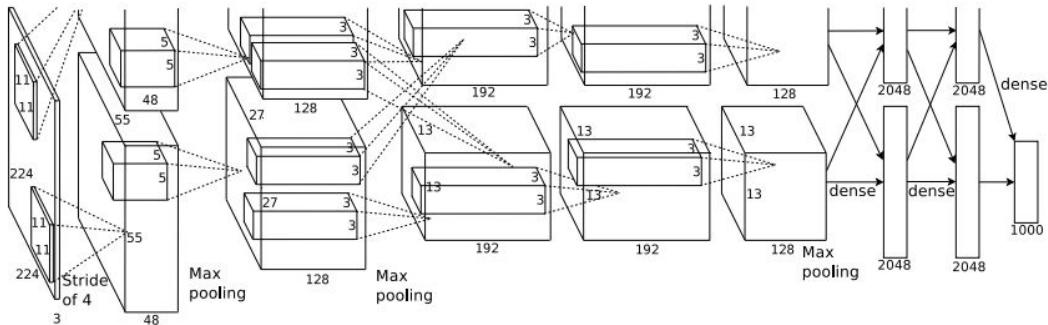
[Krizhevsky et al. 2012]

Input: 227x227x3 images

After CONV1: 55x55x96

After POOL1: 27x27x96

...



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

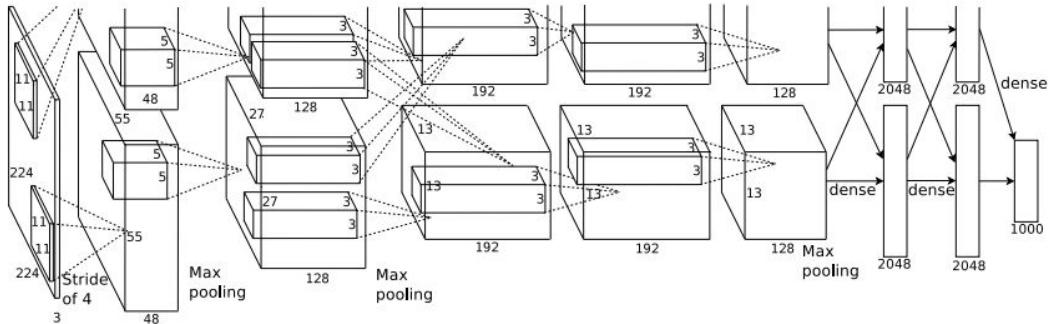
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1

[6x6x256] MAX POOL3: 3x3 filters at stride 2

[4096] FC6: 4096 neurons

[4096] FC7: 4096 neurons

[1000] FC8: 1000 neurons (class scores)



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

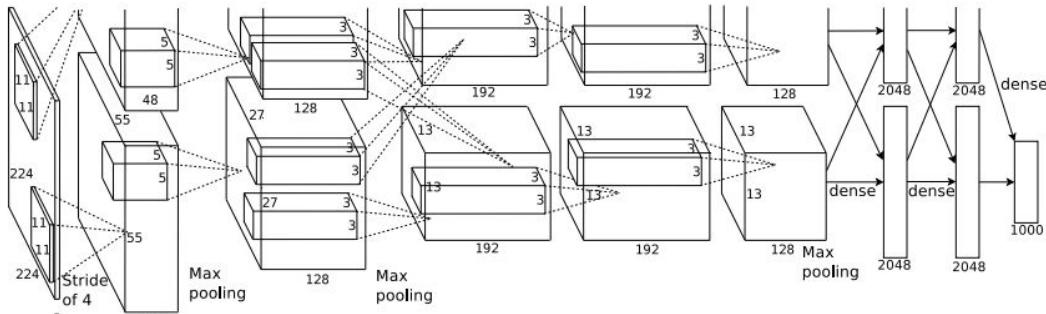
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1

[6x6x256] MAX POOL3: 3x3 filters at stride 2

[4096] FC6: 4096 neurons

[4096] FC7: 4096 neurons

[1000] FC8: 1000 neurons (class scores)



Compared to LeCun 1998:

1 DATA:

- More data: 10^6 vs. 10^3

2 COMPUTE:

- GPU ($\sim 20x$ speedup)

3 ALGORITHM:

- Deeper: More layers (8 weight layers)
- Fancy regularization (dropout)
- Fancy non-linearity (ReLU)

4 INFRASTRUCTURE:

- CUDA

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

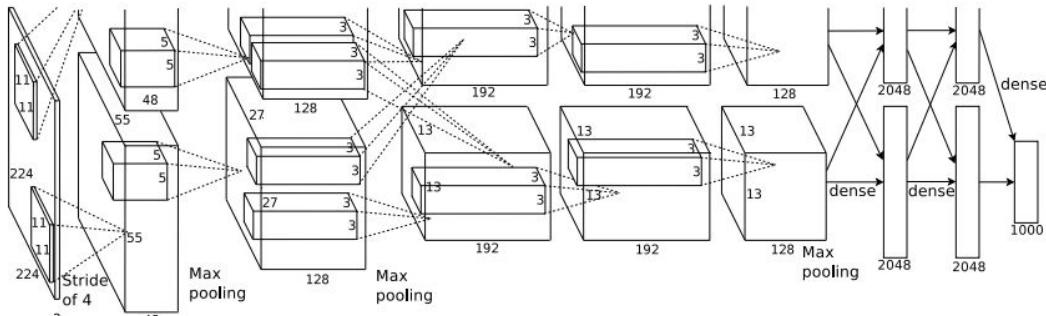
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1

[6x6x256] MAX POOL3: 3x3 filters at stride 2

[4096] FC6: 4096 neurons

[4096] FC7: 4096 neurons

[1000] FC8: 1000 neurons (class scores)

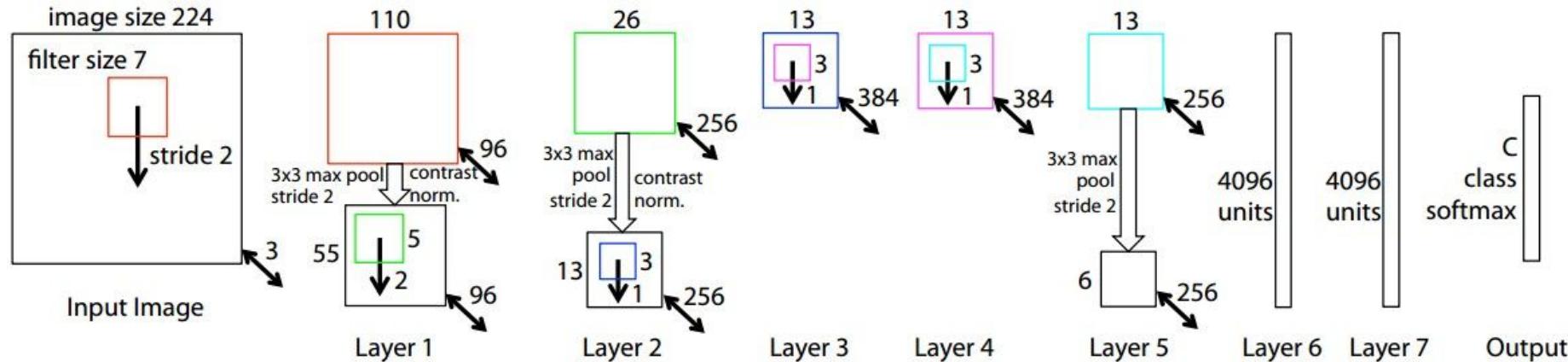


Details/Retrospectives:

- 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: ZFNet

[Zeiler and Fergus, 2013]



AlexNet but:

CONV1: change from (11x11 stride 4) to (7x7 stride 2)

CONV3,4,5: instead of 384, 384, 256 filters use 512, 1024, 512

ImageNet top 5 error: 15.4% -> 14.8%

Case Study: VGGNet

[Simonyan and Zisserman, 2014]

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

best model

11.2% top 5 error in ILSVRC 2013

->

7.3% top 5 error

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

Table 2: Number of parameters (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

INPUT: [224x224x3] memory: $224 \times 224 \times 3 = 150K$ params: 0 (not counting biases)

CONV3-64: [224x224x64] memory: $224 \times 224 \times 64 = 3.2M$ params: $(3 \times 3 \times 3) \times 64 = 1,728$

CONV3-64: [224x224x64] memory: $224 \times 224 \times 64 = 3.2M$ params: $(3 \times 3 \times 64) \times 64 = 36,864$

POOL2: [112x112x64] memory: $112 \times 112 \times 64 = 800K$ params: 0

CONV3-128: [112x112x128] memory: $112 \times 112 \times 128 = 1.6M$ params: $(3 \times 3 \times 64) \times 128 = 73,728$

CONV3-128: [112x112x128] memory: $112 \times 112 \times 128 = 1.6M$ params: $(3 \times 3 \times 128) \times 128 = 147,456$

POOL2: [56x56x128] memory: $56 \times 56 \times 128 = 400K$ params: 0

CONV3-256: [56x56x256] memory: $56 \times 56 \times 256 = 800K$ params: $(3 \times 3 \times 128) \times 256 = 294,912$

CONV3-256: [56x56x256] memory: $56 \times 56 \times 256 = 800K$ params: $(3 \times 3 \times 256) \times 256 = 589,824$

CONV3-256: [56x56x256] memory: $56 \times 56 \times 256 = 800K$ params: $(3 \times 3 \times 256) \times 256 = 589,824$

POOL2: [28x28x256] memory: $28 \times 28 \times 256 = 200K$ params: 0

CONV3-512: [28x28x512] memory: $28 \times 28 \times 512 = 400K$ params: $(3 \times 3 \times 256) \times 512 = 1,179,648$

CONV3-512: [28x28x512] memory: $28 \times 28 \times 512 = 400K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

CONV3-512: [28x28x512] memory: $28 \times 28 \times 512 = 400K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

POOL2: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: 0

CONV3-512: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

CONV3-512: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

CONV3-512: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

POOL2: [7x7x512] memory: $7 \times 7 \times 512 = 25K$ params: 0

FC: [1x1x4096] memory: 4096 params: $7 \times 7 \times 512 \times 4096 = 102,760,448$

FC: [1x1x4096] memory: 4096 params: $4096 \times 4096 = 16,777,216$

FC: [1x1x1000] memory: 1000 params: $4096 \times 1000 = 4,096,000$

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

INPUT: [224x224x3] memory: $224 \times 224 \times 3 = 150K$ params: 0 (not counting biases)

CONV3-64: [224x224x64] memory: $224 \times 224 \times 64 = 3.2M$ params: $(3 \times 3 \times 3) \times 64 = 1,728$

CONV3-64: [224x224x64] memory: $224 \times 224 \times 64 = 3.2M$ params: $(3 \times 3 \times 64) \times 64 = 36,864$

POOL2: [112x112x64] memory: $112 \times 112 \times 64 = 800K$ params: 0

CONV3-128: [112x112x128] memory: $112 \times 112 \times 128 = 1.6M$ params: $(3 \times 3 \times 64) \times 128 = 73,728$

CONV3-128: [112x112x128] memory: $112 \times 112 \times 128 = 1.6M$ params: $(3 \times 3 \times 128) \times 128 = 147,456$

POOL2: [56x56x128] memory: $56 \times 56 \times 128 = 400K$ params: 0

CONV3-256: [56x56x256] memory: $56 \times 56 \times 256 = 800K$ params: $(3 \times 3 \times 128) \times 256 = 294,912$

CONV3-256: [56x56x256] memory: $56 \times 56 \times 256 = 800K$ params: $(3 \times 3 \times 256) \times 256 = 589,824$

CONV3-256: [56x56x256] memory: $56 \times 56 \times 256 = 800K$ params: $(3 \times 3 \times 256) \times 256 = 589,824$

POOL2: [28x28x256] memory: $28 \times 28 \times 256 = 200K$ params: 0

CONV3-512: [28x28x512] memory: $28 \times 28 \times 512 = 400K$ params: $(3 \times 3 \times 256) \times 512 = 1,179,648$

CONV3-512: [28x28x512] memory: $28 \times 28 \times 512 = 400K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

CONV3-512: [28x28x512] memory: $28 \times 28 \times 512 = 400K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

POOL2: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: 0

CONV3-512: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

CONV3-512: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

CONV3-512: [14x14x512] memory: $14 \times 14 \times 512 = 100K$ params: $(3 \times 3 \times 512) \times 512 = 2,359,296$

POOL2: [7x7x512] memory: $7 \times 7 \times 512 = 25K$ params: 0

FC: [1x1x4096] memory: 4096 params: $7 \times 7 \times 512 \times 4096 = 102,760,448$

FC: [1x1x4096] memory: 4096 params: $4096 \times 4096 = 16,777,216$

FC: [1x1x1000] memory: 1000 params: $4096 \times 1000 = 4,096,000$

TOTAL memory: $24M \times 4 \text{ bytes} \approx 93\text{MB} / \text{image}$ (only forward! ~ 2 for bwd)

TOTAL params: 138M parameters

ConvNet Configuration			
B	C	D	19
13 weight layers	16 weight layers	16 weight layers	
put (224 × 224 RGB image)			
conv3-64	conv3-64	conv3-64	cc
conv3-64	conv3-64	conv3-64	cc
		maxpool	
conv3-128	conv3-128	conv3-128	co
conv3-128	conv3-128	conv3-128	co
		maxpool	
conv3-256	conv3-256	conv3-256	co
conv3-256	conv3-256	conv3-256	co
	conv1-256	conv3-256	co
		maxpool	
conv3-512	conv3-512	conv3-512	co
conv3-512	conv3-512	conv3-512	co
	conv1-512	conv3-512	co
		maxpool	
conv3-512	conv3-512	conv3-512	co
conv3-512	conv3-512	conv3-512	co
	conv1-512	conv3-512	co
		maxpool	
FC-4096			
FC-4096			
FC-1000			
soft-max			

INPUT: [224x224x3] memory: 224*224*3=150K params: 0 (not counting biases)

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

Note:

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

Most memory is in early CONV

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

Most params are in late FC

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

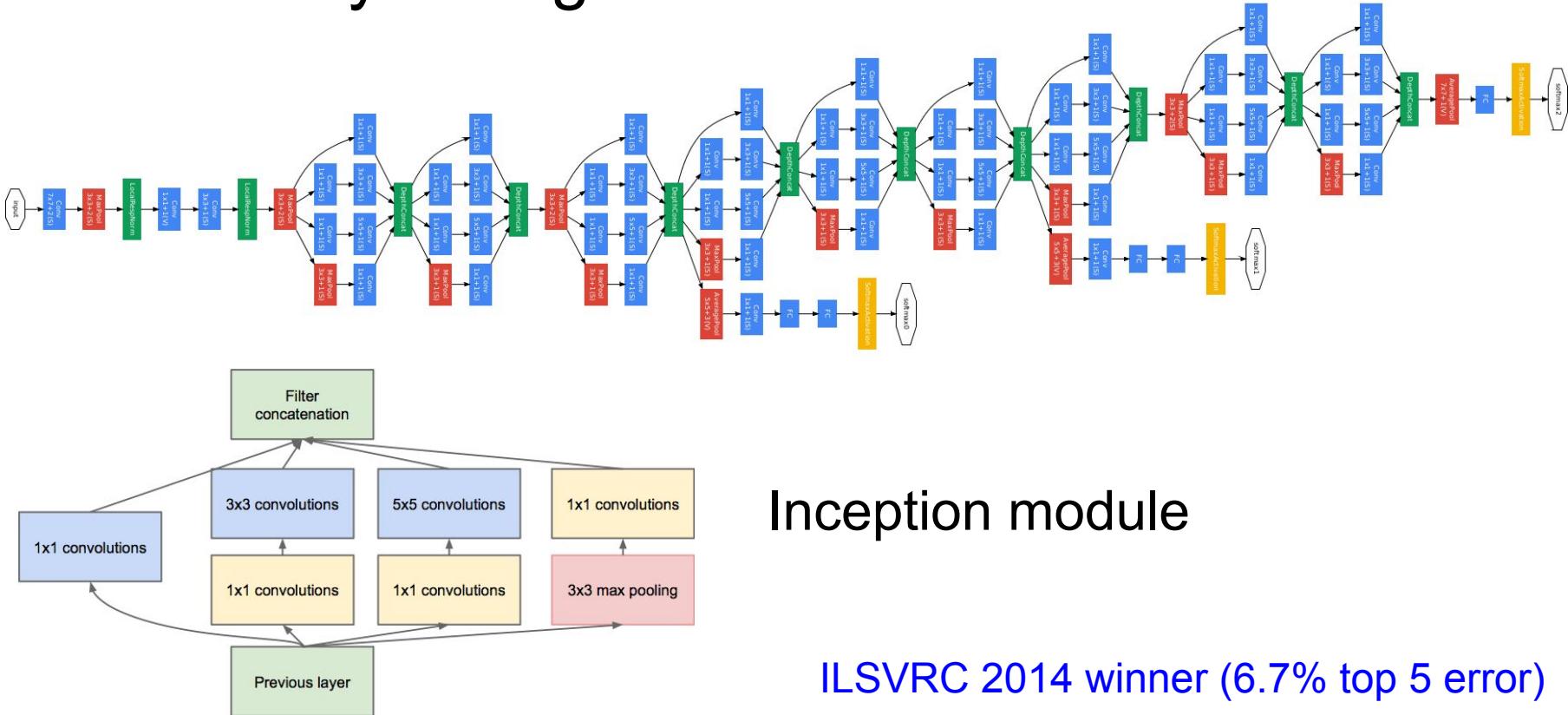
FC: [1x1x1000] memory: 1000 params: 4096*1000 = 4,096,000

TOTAL memory: 24M * 4 bytes ~= 93MB / image (only forward! ~*2 for bwd)

TOTAL params: 138M parameters

Case Study: GoogLeNet

[Szegedy et al., 2014]



Case Study: GoogLeNet

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Fun features:

- Only 5 million params!
(Removes FC layers completely)

Compared to AlexNet:

- 12X less params
- 2x more compute
- 6.67% (vs. 16.4%)

Case Study: ResNet [He et al., 2015]

ILSVRC 2015 winner (3.6% top 5 error)



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

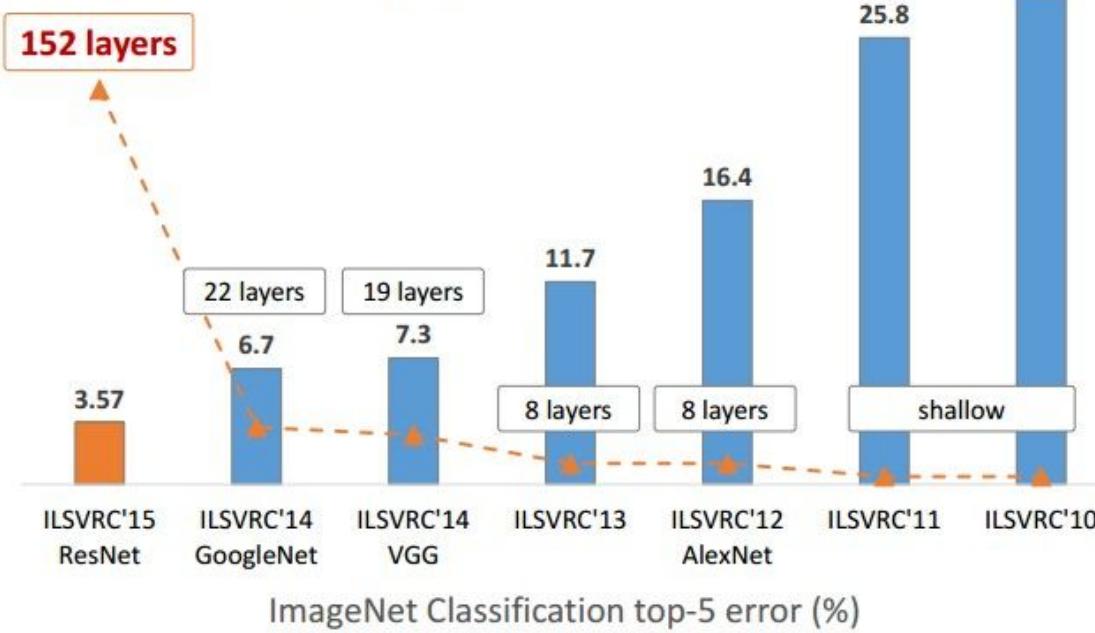
*improvements are relative numbers



Kaiming He, Xiangyu Zhang, Shaoqing Ren, & Jian Sun. “Deep Residual Learning for Image Recognition”. arXiv 2015.

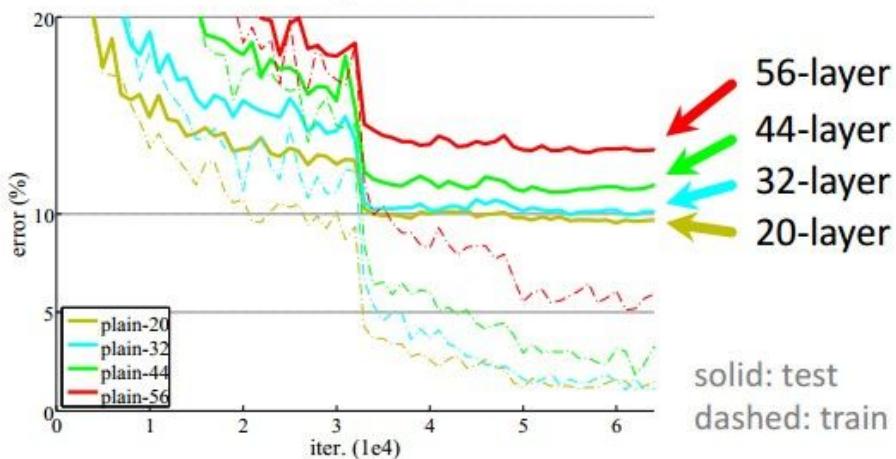
Slide from Kaiming He's recent presentation <https://www.youtube.com/watch?v=1PGLj-uKT1w>

Revolution of Depth

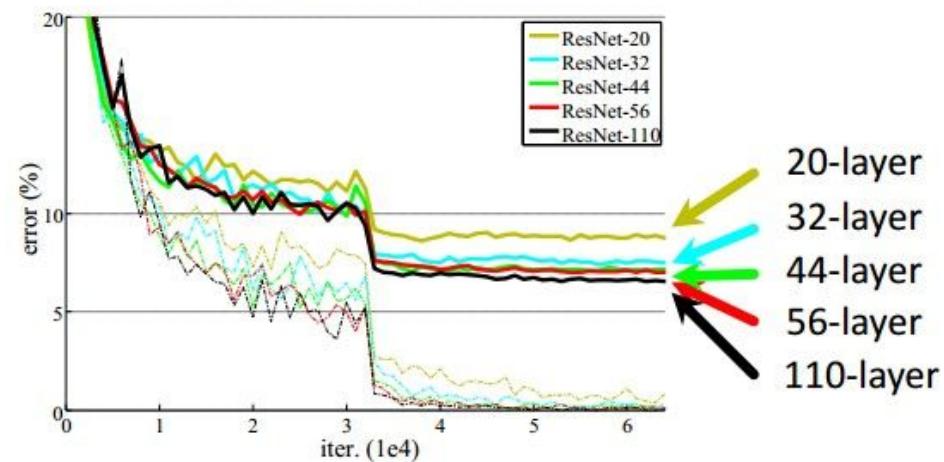


CIFAR-10 experiments

CIFAR-10 plain nets

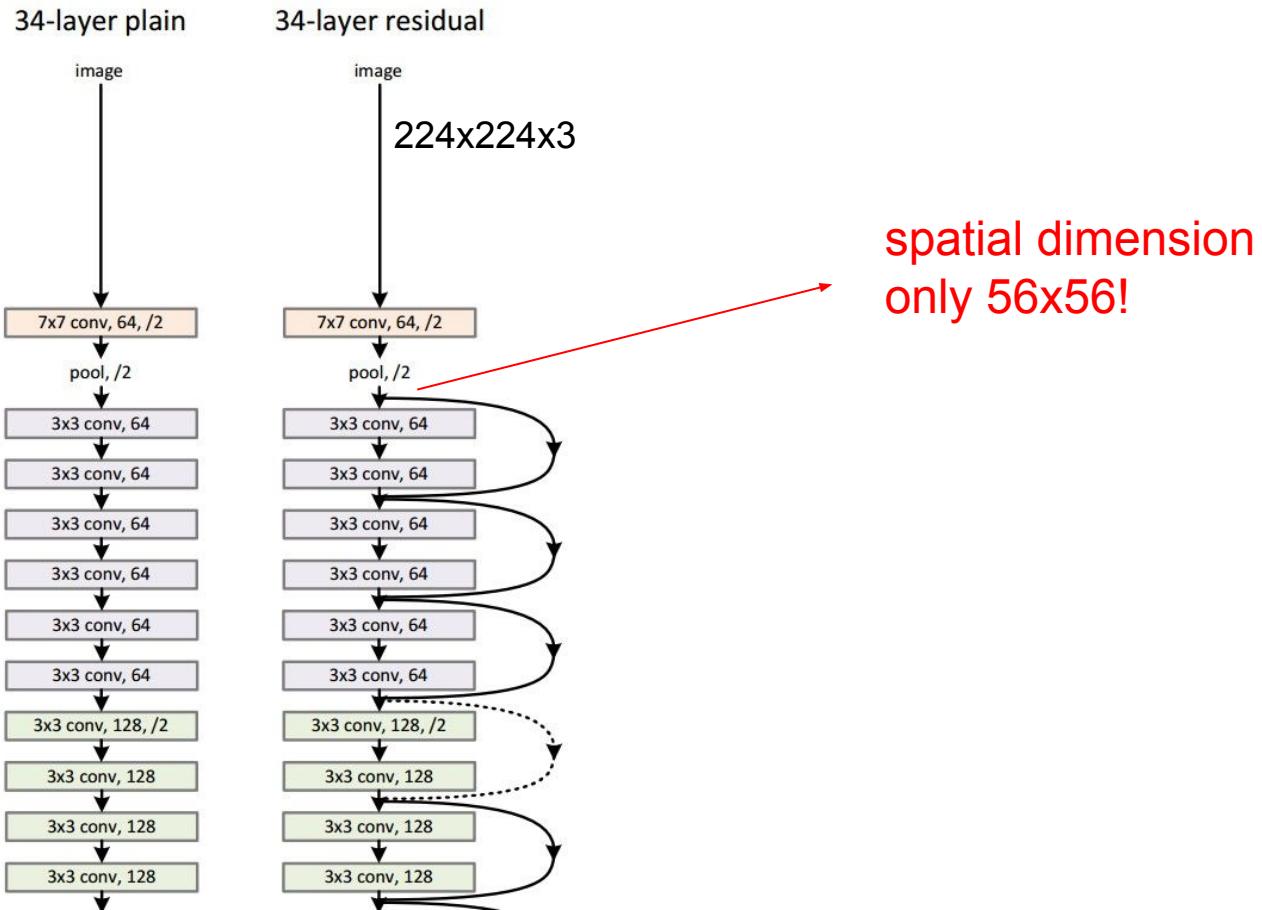


CIFAR-10 ResNets

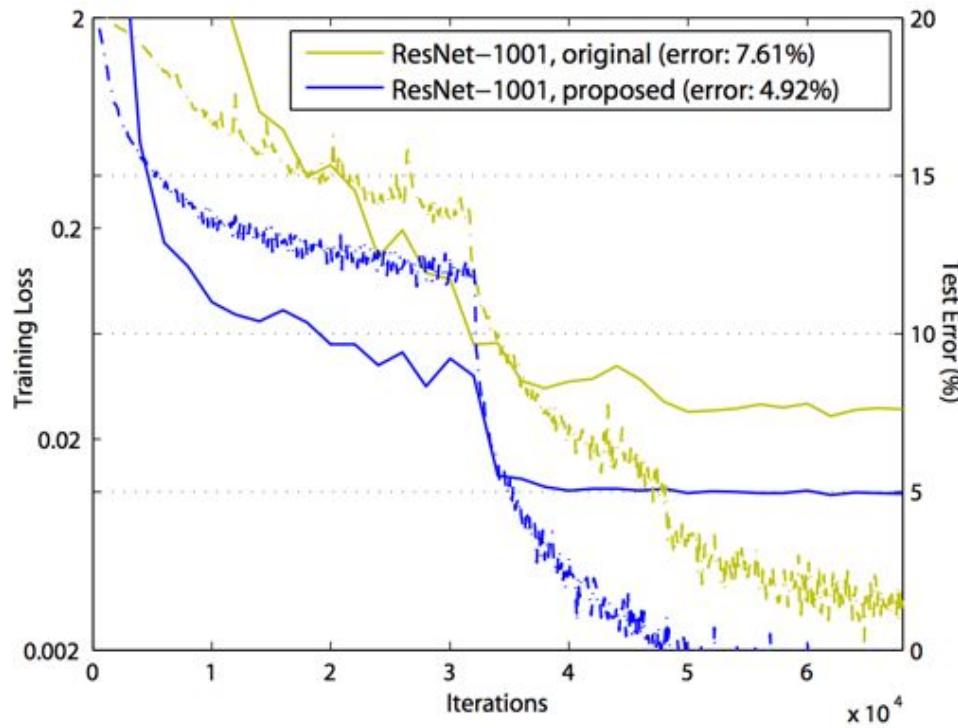
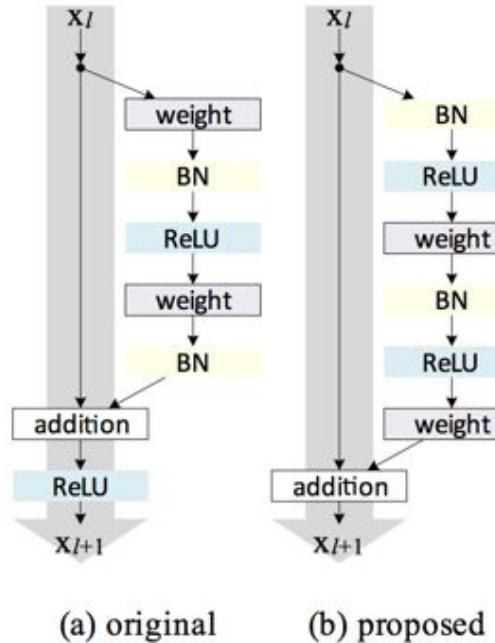


Case Study: ResNet

[He et al., 2015]

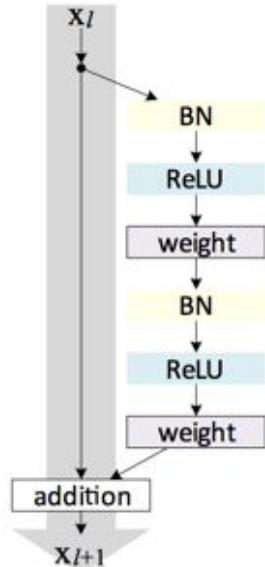


Identity Mappings in Deep Residual Networks, He et al. 2016

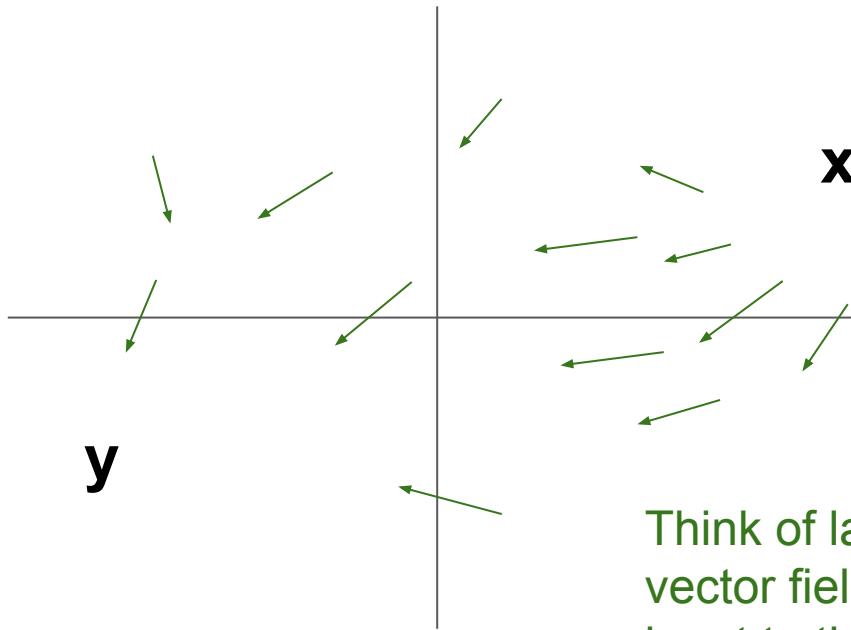


Deep Networks with Stochastic Depth, Huang et al., 2016

“We start with very deep networks but during training, for each mini-batch, randomly drop a subset of layers and bypass them with the identity function.”



(b) proposed



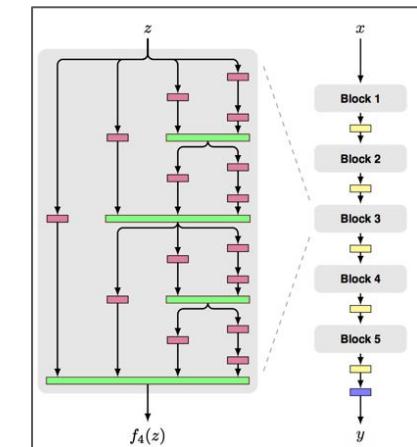
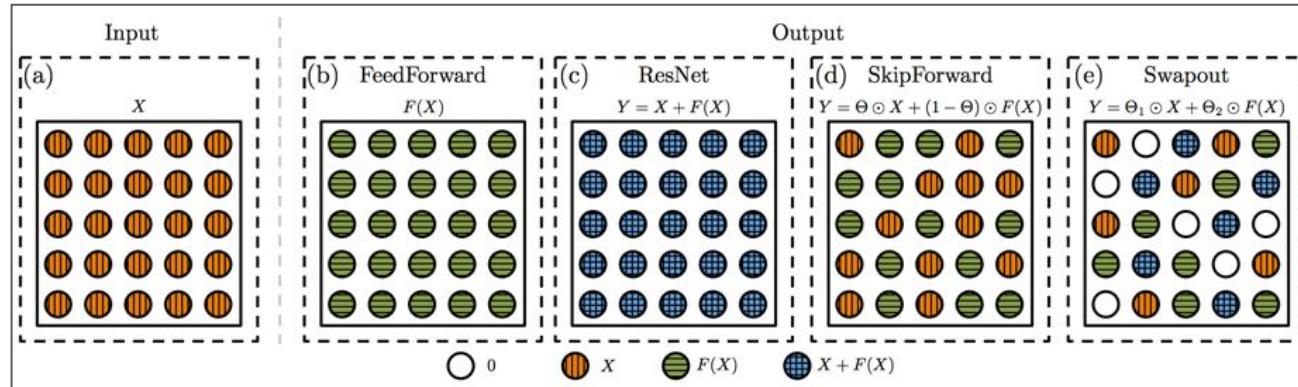
Think of layers more like vector fields, nudging the input to the label

Wide Residual Networks, Zagoruyko and Komodakis, 2016

- wide networks with only 16 layers can significantly outperform 1000-layer deep networks
- main power of residual networks is in residual blocks, and not in extreme depth
- wide residual networks are several times faster to train

Swapout: Learning an ensemble of deep architectures, Singh et al., 2016

- 32 layer wider model performs similar to a 1001 layer ResNet model



FractalNet: Ultra-Deep Neural Networks without Residuals, Larsson et al. 2016

Still an active area of research...

Densely Connected Convolutional Networks, Huang et al.

ResNet in ResNet, Targ et al.

Deeply-Fused Nets, Wang et al.

Weighted Residuals for Very Deep Networks, Shen et al.

Residual Networks of Residual Networks: Multilevel Residual Networks, Zhang et al.

...

In large part likely due to open source code available, e.g.:

The screenshot shows a GitHub repository page for 'facebook / fb.resnet.torch'. The page includes a header with navigation links for 'Pull requests', 'Issues', and 'Gist'. Below the header, there's a search bar and a user profile icon. The main content area displays the repository name 'facebook / fb.resnet.torch' and various metrics: 95 stars, 729 forks, and 262 issues. At the bottom, there are links for 'Code', 'Issues 12', 'Pull requests 3', 'Projects 0', 'Wiki', 'Pulse', and 'Graphs'. A red border highlights the entire repository card.

ASIDE: arxiv-sanity.com plug

Arxiv Sanity Preserver
Built by @karpathy to accelerate research.
Serving last 21816 papers from cs.[CV|CL|LG|NE|stat.ML]

andrej log out Fork me on GitHub

most recent top recent recommended library

Identity Mappings in Deep Residual Networks
Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun
7/25/2016 (v1: 3/16/2016) cs.CV | cs.LG
ECCV 2016 camera-ready

Deep residual networks have emerged as a family of extremely deep architectures showing compelling accuracy and nice convergence behaviors. In this paper, we analyze the propagation formulations behind the residual building blocks, which suggest that the forward and backward signals can be directly propagated from one block to any other block, when using identity mappings as the skip connections and after-addition activation. A series of ablation experiments sum up the importance of these identity mappings. This motivates us to propose a new residual unit, which makes training easier and improves generalization. We report improved results using a 1001-layer ResNet on CIFAR-10 (4.62% error) and CIFAR-100, and a 200-layer ResNet on ImageNet. Code is available at: <https://github.com/KaimingHe/resnet-1k-layers>

Most similar papers:

Resnet in Resnet: Generalizing Residual Architectures
Sasha Targ, Diogo Almeida, Kevin Lyman
3/25/2016 cs.LG | cs.CV | cs.NE | stat.ML

Residual networks (ResNets) have recently achieved state-of-the-art on challenging computer vision tasks. We introduce Resnet in Resnet (RIR): a deep dual-stream architecture that generalizes ResNets and standard CNNs and is easily implemented with no computational overhead. RIR consistently improves performance over ResNets, outperforms architectures with similar amounts of augmentation on CIFAR-10, and establishes a new state-of-the-art on CIFAR-100.

Convolutional Residual Memory Networks
Joel Moniz, Christopher Pal
7/14/2016 (v1: 6/16/2016) cs.CV

In this paper we address the question of how to render sequence-level networks better at handling structured input. We propose a machine reading simulator which processes text incrementally from left to right and performs shallow reasoning with memory and attention. The reader extends the Long Short-Term Memory architecture with a memory network in place of a single memory cell. This enables adaptive memory usage during recurrence with neural attention, offering a way to weakly induce relations among tokens. The system is initially designed to process a single sequence but we also demonstrate how to integrate it with an encoder-decoder architecture. Experiments on language modeling, sentiment analysis, and natural language inference show that our model matches or outperforms the state of the art.

Arxiv Sanity Preserver
Built by @karpathy to accelerate research.
Serving last 21816 papers from cs.[CV|CL|LG|NE|stat.ML]

andrej log out Fork me on GitHub

most recent top recent recommended library

Only show v1 | Last day Last 3 days Last week Last month Last year All time

Recommended papers: (based on SVM trained on tfidf of papers in your library, refreshed every day or so)

Neural Photo Editing with Introspective Adversarial Networks
Andrew Brock, Theodore Lim, J. M. Ritchie, Nick Weston
9/22/2016 cs.LG | cs.CV | cs.NE | stat.ML
10 pages, 6 figures

We present the Neural Photo Editor, an interface for exploring the latent space of generative image models and making large, semantically coherent changes to existing images. Our interface is powered by the Introspective Adversarial Network, a hybridization of the Generative Adversarial Network and the Variational Autoencoder designed for use in the editor. Our model makes use of a novel computational block based on dilated convolutions, and Orthogonal Regularization, a novel weight regularization method. We validate our model on CelebA, SVHN, and ImageNet, and produce samples and reconstructions with high visual fidelity.

Long Short-Term Memory-Networks for Machine Reading
Jianguo Cheng, Li Dong, Mirella Lapata
9/20/2016 (v1: 1/25/2016) cs.CL | cs.NE
Published as a conference paper at EMNLP 2016

In this paper we address the question of how to render sequence-level networks better at handling structured input. We propose a machine reading simulator which processes text incrementally from left to right and performs shallow reasoning with memory and attention. The reader extends the Long Short-Term Memory architecture with a memory network in place of a single memory cell. This enables adaptive memory usage during recurrence with neural attention, offering a way to weakly induce relations among tokens. The system is initially designed to process a single sequence but we also demonstrate how to integrate it with an encoder-decoder architecture. Experiments on language modeling, sentiment analysis, and natural language inference show that our model matches or outperforms the state of the art.

Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations
Ilay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, Yoshua Bengio
9/22/2016 cs.NE | cs.LG

arXiv admin note: text reworded with arXiv:1602.02830

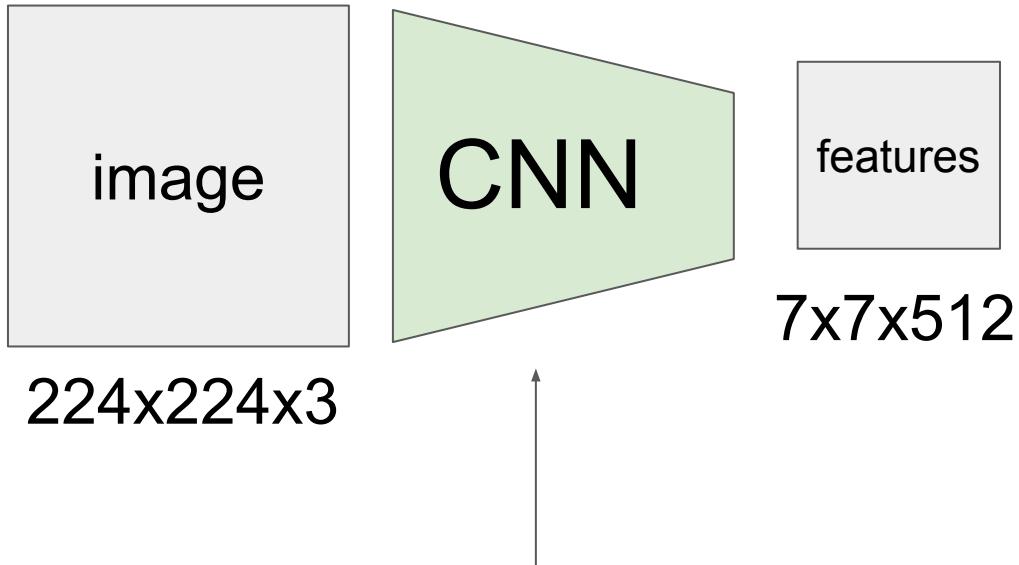
1609.07093v1 pdf show similar papers | review

1601.06733v7 pdf show similar papers | review

1609.07061v1 pdf show similar papers | review

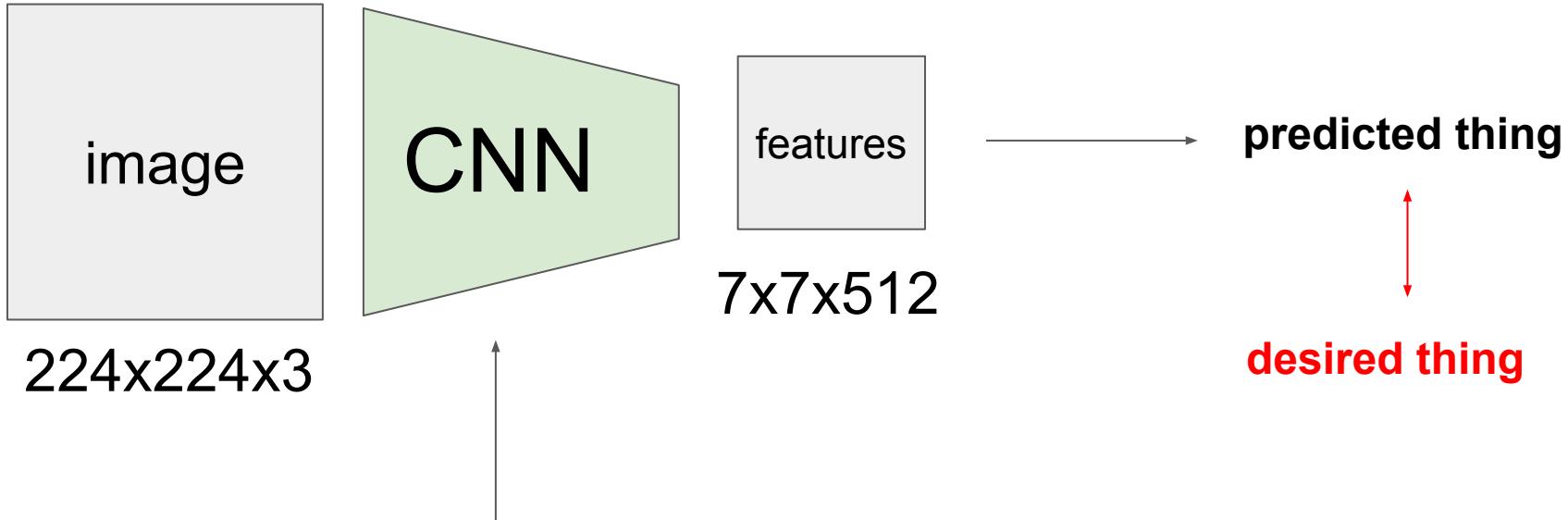
Addressing other tasks...

Addressing other tasks...



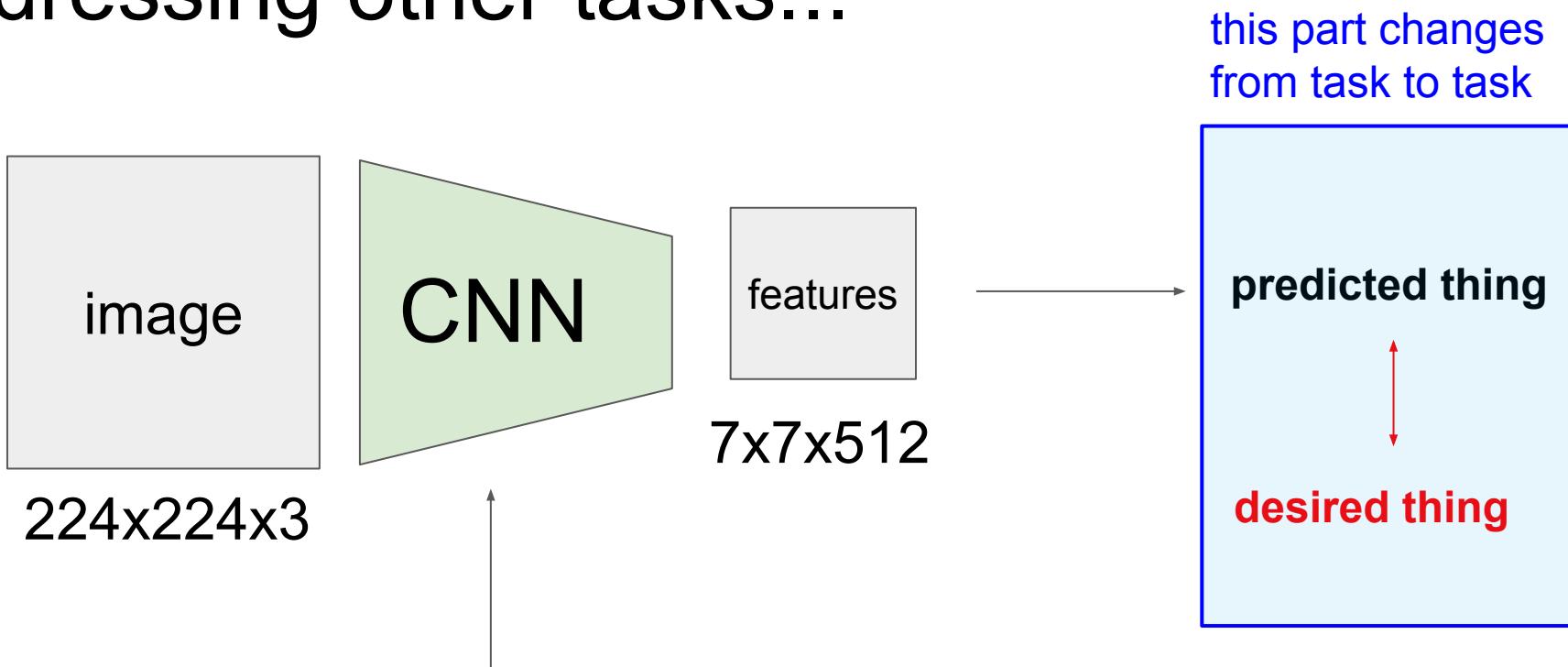
A block of compute with a few
million parameters.

Addressing other tasks...



A block of compute with a few
million parameters.

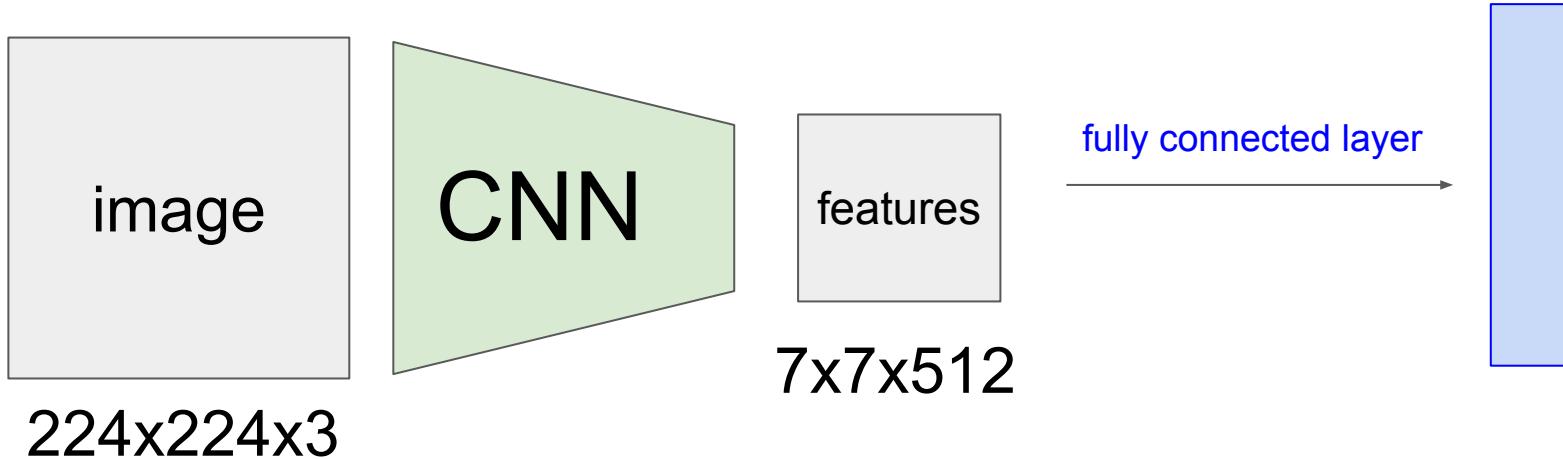
Addressing other tasks...



A block of compute with a few million parameters.

Image Classification

thing = a vector of probabilities for different classes

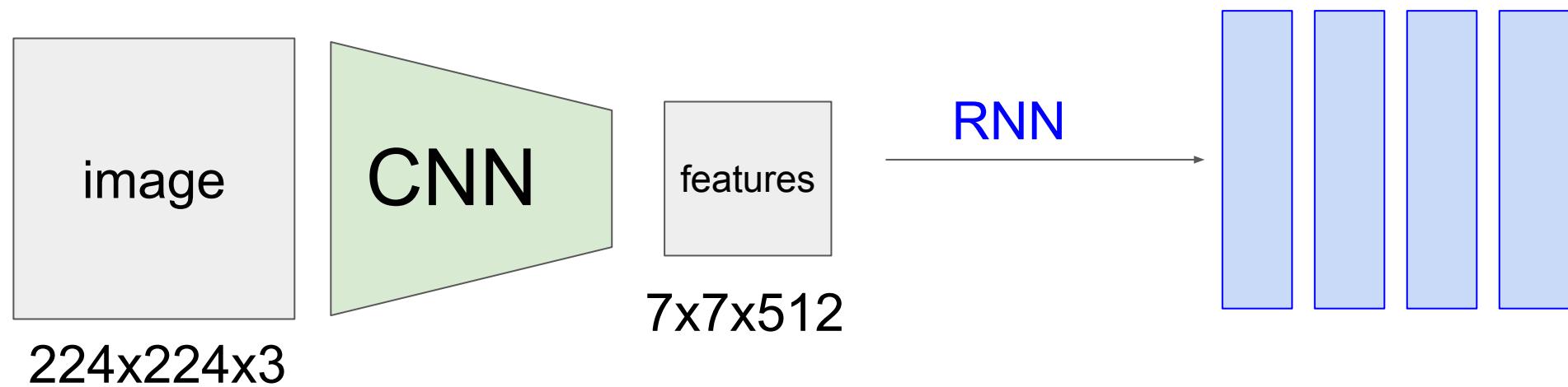


e.g. vector of 1000 numbers giving probabilities for different classes.

Image Captioning

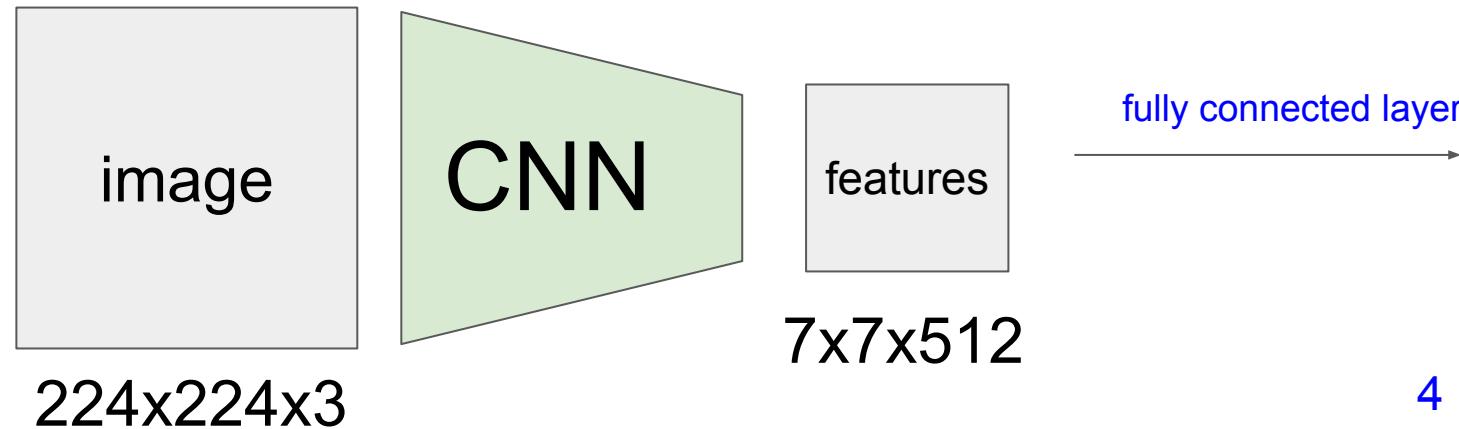
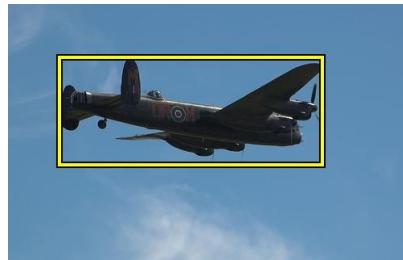


A person on a beach flying a kite.



A sequence of 10,000-dimensional vectors giving probabilities of different words in the caption.

Localization

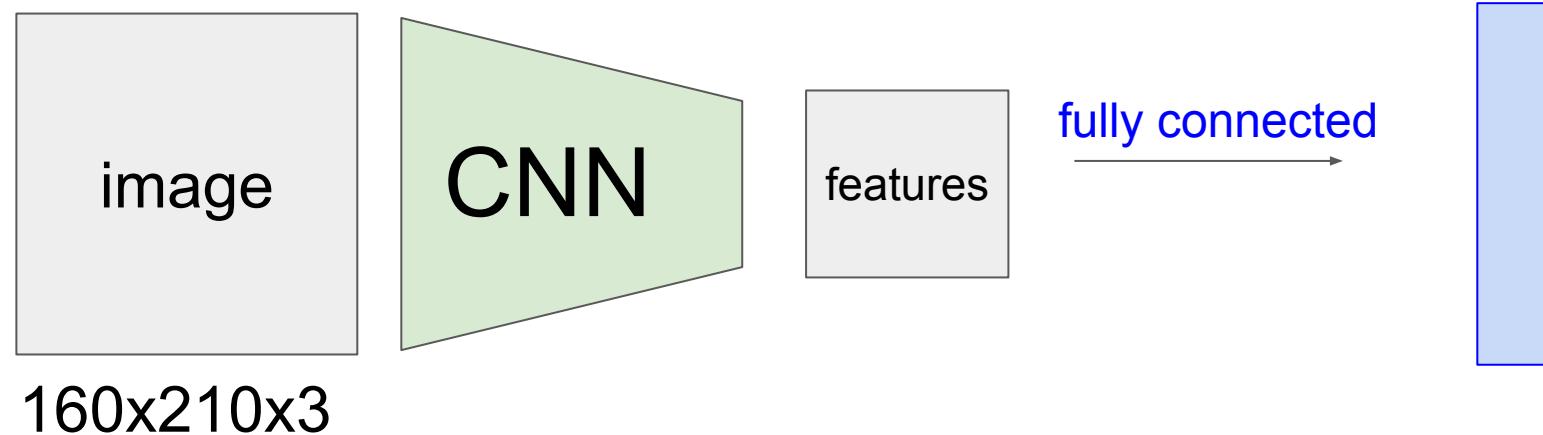


- X coord
- Y coord
- Width
- Height

Reinforcement Learning



Mnih et al. 2015



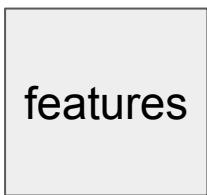
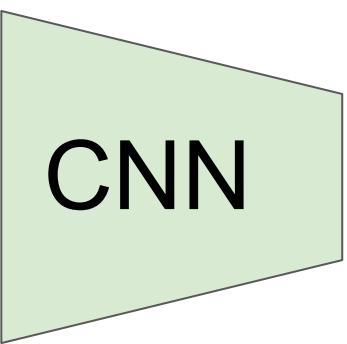
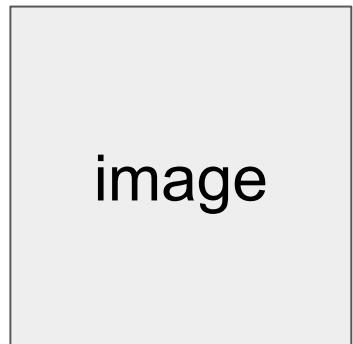
e.g. vector of 8 numbers giving probability of wanting to take any of the 8 possible ATARI actions.

Segmentation

image



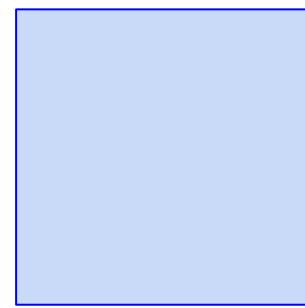
class “map”



$224 \times 224 \times 3$

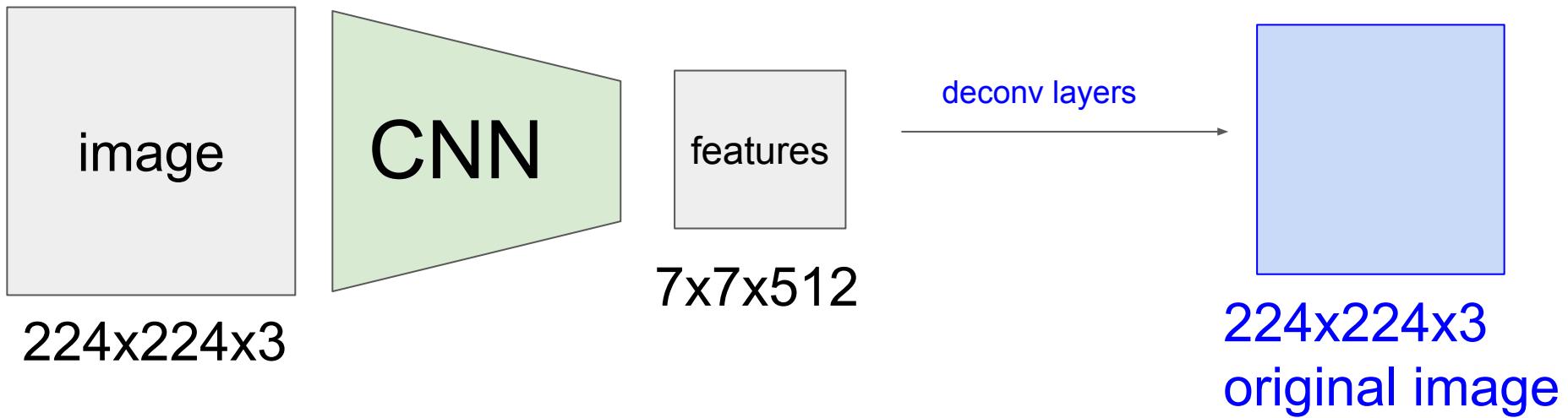
$7 \times 7 \times 512$

deconv layers

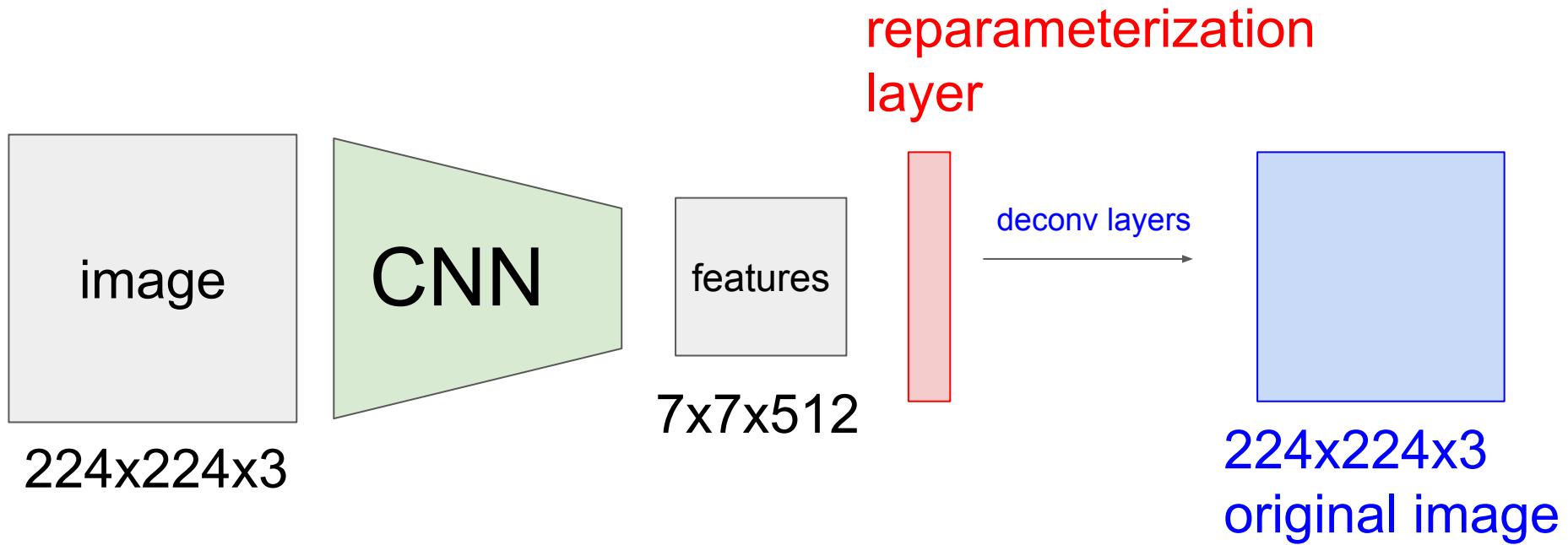


$224 \times 224 \times 20$
array of class
probabilities at
each pixel.

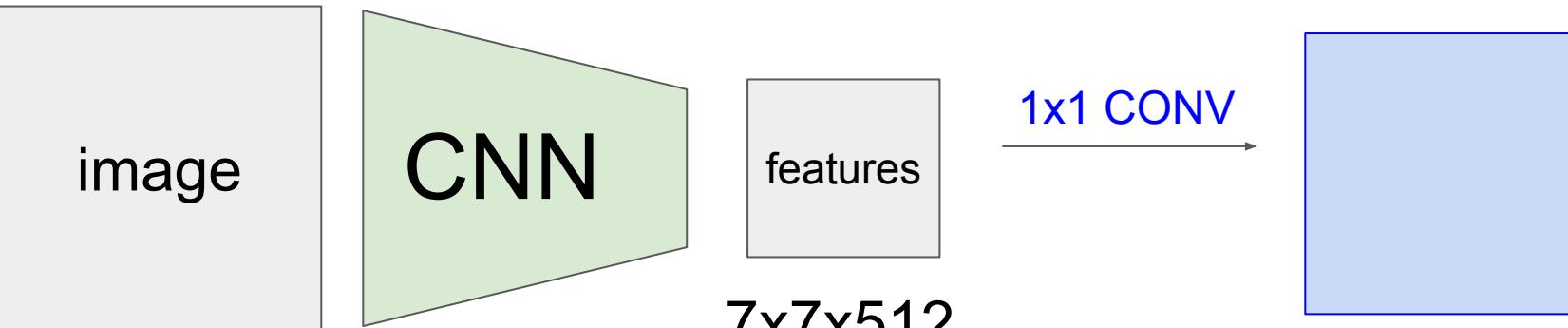
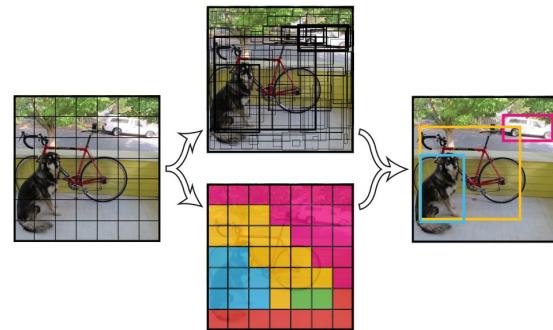
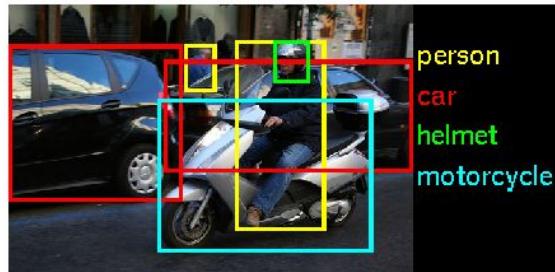
Autoencoders



Variational Autoencoders



Detection



224x224x3

7x7x512

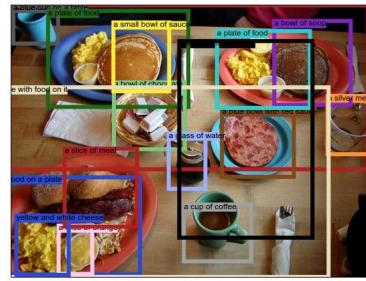
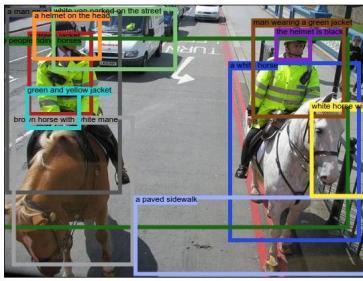
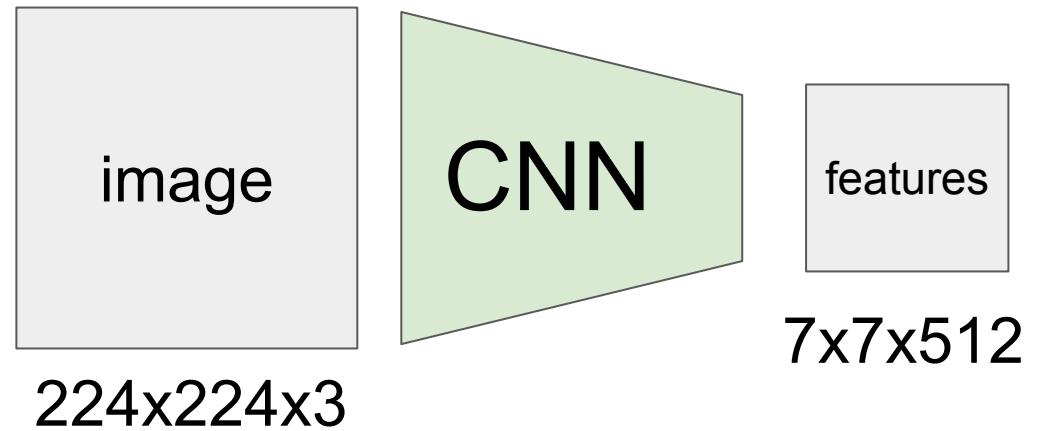
7x7x(5*B+C)

For each of 7x7 locations:

- [x,y,width,height,confidence]*B
- class

E.g. YOLO: You Only Look Once (Demo: <http://pjreddie.com/darknet/yolo/>)

Dense Image Captioning



For each of 7x7 locations:

- x,y,width,height,confidence
- sequence of words

Practical considerations when applying ConvNets

What hardware do I use?

Buy your own machine:

- NVIDIA DIGITS DevBox (TITAN X GPUs)
- NVIDIA DGX-1 (P100 GPUs)

Build your own machine:

<https://graphific.github.io/posts/building-a-deep-learning-dream-machine/>

GPUs in the cloud:

- Amazon AWS (GRID K520 :()
- Microsoft Azure (soon); 4x K80 GPUs
- Cirrascale (“rent-a-box”)

What framework do I use?

Caffe

No need to write code!

1. Convert data (run a script)
2. Define net (edit prototxt)
3. Define solver (edit prototxt)
4. Train (with pretrained weights)

TensorFlow

```
1 import tensorflow as tf
2 import numpy as np
3 
4 N, D, H, C = 64, 1000, 100, 10
5 
6 x = tf.placeholder(tf.float32, shape=[None, D])
7 y = tf.placeholder(tf.float32, shape=[None, C])
8 
9 w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
10 w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))
11 
12 a = tf.matmul(x, w1)
13 a_relu = tf.nn.relu(a)
14 scores = tf.matmul(a_relu, w2)
15 probs = tf.nn.softmax(scores)
16 loss = -tf.reduce_sum(y * tf.log(probs))
17 
18 learning_rate = 1e-2
19 train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
20 
21 xx = np.random.randint(N, D).astype(np.float32)
22 yy = np.zeros((N, C)).astype(np.float32)
23 yy[0].randint(C, size=N) = 1
24 
25 with tf.Session() as sess:
26     sess.run(tf.initialize_all_variables())
27 
28 for t in xrange(100):
29     _, loss_value = sess.run([train_step, loss],
30                            feed_dict={x: xx, y: yy})
31     print loss_value
```

Torch

```
1 require 'torch'
2 require 'nn'
3 require 'optim'
4 
5 -- Batch size, input dim, hidden dim, num classes
6 local N, D, H, C = 100, 1000, 100, 10
7 
8 -- Build a one-layer ReLU network
9 local net = nn.Sequential()
10 net:add(nn.Linear(D, H))
11 net:add(nn.ReLU())
12 net:add(nn.Linear(H, C))
13 
14 -- Collect all weights and gradients in a single Tensor
15 local weights, grad_weights = net:getParameters()
16 
17 -- Loss functions are called "criterions"
18 local crit = nn.CrossEntropyCriterion() -- Softmax loss
19 
20 -- Callback to interface with optim methods
21 local function f(w)
22     assert(w == weights)
23 
24     -- Generate some random input data
25     local x = torch.randn(N, D)
26     local y = torch.Tensor(N):random(C)
27 
28     -- Forward pass: Compute scores and loss
29     local scores = net:forward(x)
30     local loss = crit:forward(scores, y)
31 
32     -- Backward pass: Compute gradients
33     grad_weights:zero()
34     local dscores = crit:backward(scores, y)
35     local dx = net:backward(x, dscores)
36 
37     return loss, grad_weights
38 end
39 
40 -- Make a step using Adam
41 local state = {learningRate=1e-3}
42 optim.adam(f, weights, state)
```

Theano

```
import theano
import theano.tensor as T

# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')

relu = lasagne.nonlinearities.rectify
softmax = lasagne.nonlinearities.softmax
net = lasagne.layers.InputLayer(shape=(None, D), input_var=x)
net = lasagne.layers.DenseLayer(net, H, nonlinearity=relu)
net = lasagne.layers.DenseLayer(net, C, nonlinearity=softmax)

probs = lasagne.layers.get_output(net)
loss = lasagne.objectives.categorical_crossentropy(probs, y).mean()

params = lasagne.layers.get_all_params(net, trainable=True)
updates = lasagne.updates.nesterov_momentum(
    params, grads, learning_rate=1e-2, momentum=0.9)

train_fn = theano.function([x, y], [loss, updates])

xx = np.random.randint(N, D)
yy = np.random.randint(C, size=N).astype(np.int64)

for t in xrange(100):
    loss_val = train_fn(xx, yy)
    print loss_val
```

Lasagne

```
1 import numpy as np
2 import theano
3 import theano.tensor as T
4 import lasagne
5 
6 N, D, H, C = 64, 1000, 100, 10
7 
8 x = T.matrix('x')
9 y = T.vector('y', dtype='int64')
10 
11 relu = lasagne.nonlinearities.rectify
12 softmax = lasagne.nonlinearities.softmax
13 net = lasagne.layers.InputLayer(shape=(None, D), input_var=x)
14 net = lasagne.layers.DenseLayer(net, H, nonlinearity=relu)
15 net = lasagne.layers.DenseLayer(net, C, nonlinearity=softmax)
16 
17 probs = lasagne.layers.get_output(net)
18 loss = lasagne.objectives.categorical_crossentropy(probs, y).mean()
19 
20 params = lasagne.layers.get_all_params(net, trainable=True)
21 updates = lasagne.updates.nesterov_momentum(
    params, grads, learning_rate=1e-2, momentum=0.9)
22 
23 train_fn = theano.function([x, y], [loss, updates])
24 
25 xx = np.random.randint(N, D)
26 yy = np.random.randint(C, size=N).astype(np.int64)
27 
28 for t in xrange(100):
29     loss_val = train_fn(xx, yy)
30     print loss_val
```

Keras

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD
from keras.utils import np_utils

D, H, C = 1000, 100, 10

model = Sequential()
model.add(Dense(input_dim=D, output_dim=H))
model.add(Activation('relu'))
model.add(Dense(input_dim=H, output_dim=C))
model.add(Activation('softmax'))

sgd = SGD(lr=1e-3, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd)

N, N_batch = 1000, 32
X = np.random.randint(N, D)
y = np.random.randint(C, size=N)
y = np_utils.to_categorical(y)

model.fit(X, y, nb_epoch=5, batch_size=N_batch, verbose=2)
```

Mxnet chainer

Nervana's Neon Microsoft's CNTK Deeplearning4j

...

What framework do I use?

Caffe

No need to write code!

1. Convert data (run a script)
2. Define net (edit prototxt)
3. Define solver (edit prototxt)
4. Train (with pretrained weights)

TensorFlow

```
1 import tensorflow as tf
2 import numpy as np
3 
4 N, D, H, C = 64, 1000, 100, 10
5 
6 x = tf.placeholder(tf.float32, shape=[None, D])
7 y = tf.placeholder(tf.float32, shape=[None, C])
8 
9 w1 = tf.Variable(1e-3 * np.random.randn(D, H).astype(np.float32))
10 w2 = tf.Variable(1e-3 * np.random.randn(H, C).astype(np.float32))
11 
12 a = tf.matmul(x, w1)
13 a_relu = tf.nn.relu(a)
14 scores = tf.matmul(a_relu, w2)
15 probs = tf.nn.softmax(scores)
16 loss = -tf.reduce_sum(y * tf.log(probs))
17 
18 learning_rate = 1e-2
19 train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
20 
21 xx = np.random.rand(N, D).astype(np.float32)
22 yy = np.zeros((N, C)).astype(np.float32)
23 yy[np.arange(N), np.random.randint(C, size=N)] = 1
24 
25 with tf.Session() as sess:
26     sess.run(tf.initialize_all_variables())
27 
28 for t in xrange(100):
29     _, loss_value = sess.run([train_step, loss],
30                            feed_dict={x: xx, y: yy})
31     print loss_value
```

Torch

```
1 require 'torch'
2 require 'nn'
3 require 'optim'
4 
5 -- Batch size, input dim, hidden dim, num classes
6 local N, D, H, C = 100, 1000, 100, 10
7 
8 -- Build a one-layer ReLU network
9 local net = nn.Sequential()
10 net:add(nn.Linear(D, H))
11 net:add(nn.ReLU())
12 net:add(nn.Linear(H, C))
13 
14 -- Collect all weights and gradients in a single Tensor
15 local weights, grad_weights = net:getParameters()
16 local crit = nn.CrossEntropyCriterion()
17 
18 -- Loss functions are called "criterions"
19 local crit = nn.CrossEntropyCriterion() -- Softmax loss
20 
21 -- Callback to interface with optim methods
22 local function f(w)
23     assert(w == weights)
24 
25     -- Generate some random input data
26     local x = torch.randn(N, D)
27     local y = torch.Tensor(N):random(C)
28 
29     -- Forward pass: Compute scores and loss
30     local scores = net:forward(x)
31     local loss = crit:forward(scores, y)
32 
33     -- Backward pass: Compute gradients
34     grad_weights:zero()
35     local dscores = crit:backward(scores, y)
36     local dx = net:backward(x, dscores)
37 
38     return loss, grad_weights
39 end
40 
41 -- Make a step using Adam
42 local state = {learningRate=1e-3}
43 optim.adam(f, weights, state)
44
```

2,3

Theano

```
import theano
import theano.tensor as T
# Batch size, input dim, hidden dim, num classes
N, D, H, C = 64, 1000, 100, 10

x = T.matrix('x')
y = T.vector('y', dtype='int64')
w1 = T.matrix('w1')
w2 = T.matrix('w2')

# Forward pass: Compute scores
a = x.dot(w1)
a_relu = T.nnet.relu(a)
scores = a_relu.dot(w2)

# Forward pass: compute softmax loss
probs = T.nnet.softmax(scores)
loss = T.nnet.categorical_crossentropy(probs, y).mean()

# Backward pass: compute gradients
dw1, dw2 = T.grad(loss, [w1, w2])

f = theano.function(
    inputs=[x, y, w1, w2],
    outputs=[loss, scores, dw1, dw2],
)
```

Lasagne

```
1 import numpy as np
2 import theano
3 import theano.tensor as T
4 import lasagne
5 
6 N, D, H, C = 64, 1000, 100, 10
7 
8 x = T.matrix('x')
9 y = T.vector('y', dtype='int64')
10 
11 relu = lasagne.nonlinearities.rectify
12 softmax = lasagne.nonlinearities.softmax
13 net = lasagne.layers.InputLayer(shape=(None, D), input_var=x)
14 net = lasagne.layers.DenseLayer(net, H, nonlinearity=relu)
15 net = lasagne.layers.DenseLayer(net, C, nonlinearity=softmax)
16 
17 probs = lasagne.layers.get_output(net)
18 loss = lasagne.objectives.categorical_crossentropy(probs, y).mean()
19 
20 params = lasagne.layers.get_all_params(net, trainable=True)
21 updates = lasagne.updates.nesterov_momentum(params,
22                                              learning_rate=1e-2, momentum=0.9)
23 
24 train_fn = theano.function([x, y], loss, updates=updates)
25 
26 xx = np.random.randn(N, D)
27 yy = np.random.randint(C, size=N).astype(np.int64)
28 
29 for t in xrange(100):
30     loss_val = train_fn(xx, yy)
31     print loss_val
```

Keras

```
from keras.models import Sequential
from keras.layers.core import Dense, Activation
from keras.optimizers import SGD
from keras.utils import np_utils

D, H, C = 1000, 100, 10

model = Sequential()
model.add(Dense(input_dim=D, output_dim=H))
model.add(Activation('relu'))
model.add(Dense(input_dim=H, output_dim=C))
model.add(Activation('softmax'))

sgd = SGD(lr=1e-3, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd)

N, N_batch = 1000, 32
X = np.random.randn(N, D)
y = np.random.randint(C, size=N)
y = np_utils.to_categorical(y)

model.fit(X, y, nb_epoch=5, batch_size=N_batch, verbose=2)
```

1

Mxnet
chainer
Nervana's Neon
Microsoft's CNTK
Deeplearning4j

...

Q: How do I know what architecture to use?

Q: How do I know what architecture to use?

A: don't be a hero.

1. Take whatever works best on ILSVRC (latest ResNet)
2. Download a pretrained model
3. Potentially add/delete some parts of it
4. Finetune it on your application.

Q: How do I know what hyperparameters to use?

Q: How do I know what hyperparameters to use?

A: don't be a hero.

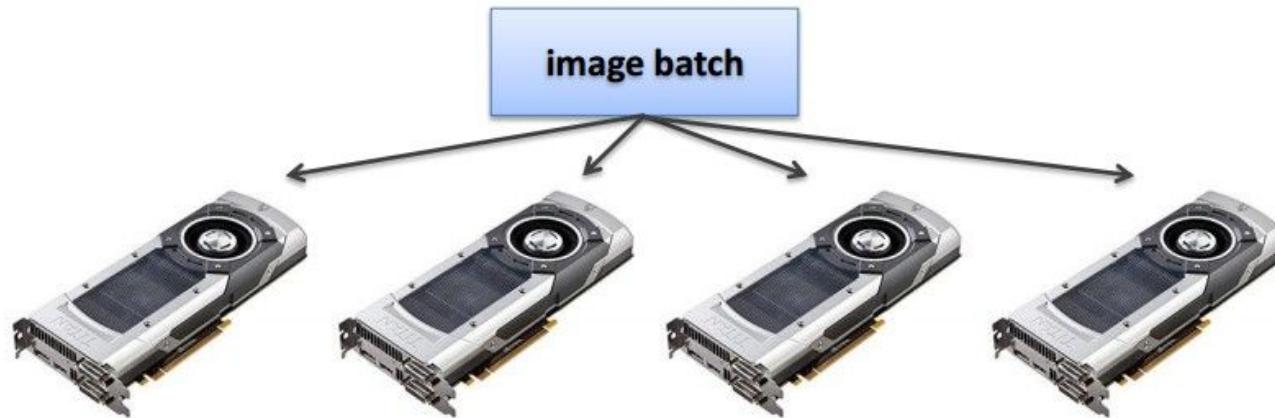
- Use whatever is reported to work best on ILSVRC.
- Play with the regularization strength (dropout rates)

ConvNets in practice: Distributed training

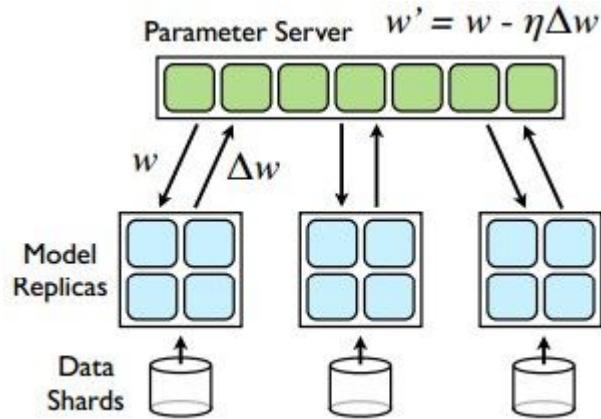
VGG: ~2-3 weeks training with 4 GPUs

ResNet 101: 2-3 weeks with 4 GPUs

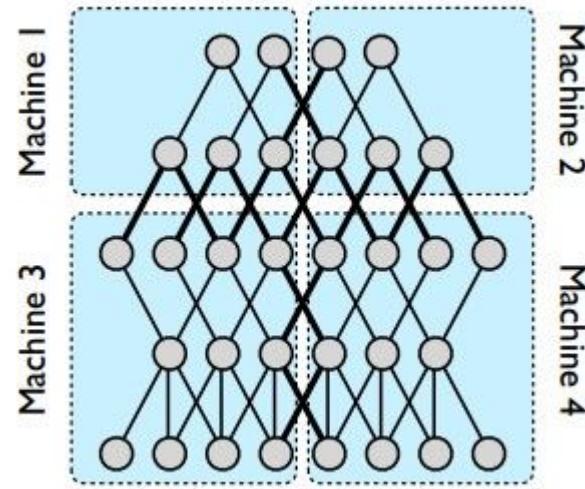
~\$1K each



ConvNets in practice: Distributed training



Data parallelism



Model parallelism

[*Large Scale Distributed Deep Networks, Jeff Dean et al., 2013*]

ConvNets in practice: pre-fetching threads

Moving parts lol

CPU-disk bottleneck

Hard disk is slow to read from
=> Pre-processed images
stored contiguously in files, read as
raw byte stream from SSD disk



CPU-GPU bottleneck

CPU data prefetch+augment thread running
while
GPU performs forward/backward pass



Learn more! CS231n

- lecture videos on YouTube
- slides
- notes
- assignments

cs231n.stanford.edu

CS231n: Convolutional Neural Networks for Visual Recognition

*This network is running live in your browser

Course Description

Computer Vision has become ubiquitous in our society, with applications in search, image understanding, apps, mapping, medicine, drones, and self-driving cars. Core to many of these applications are visual recognition tasks such as image classification, localization and detection. Recent developments in neural network (aka "deep learning") approaches have greatly advanced the performance of these state-of-the-art visual recognition systems. This course is a deep dive into details of the deep learning architectures with a focus on learning end-to-end models for these tasks, particularly image classification. During the 10-week course, students will learn to implement, train and debug their own neural networks and gain a detailed understanding of cutting-edge research in computer vision. The final assignment will involve training a multi-million parameter convolutional neural network and applying it on the largest image classification dataset (ImageNet). We will focus on teaching how to set up the problem of image recognition, the learning algorithms (e.g. backpropagation), practical engineering tricks for training and fine-tuning the networks and guide the students through hands-on assignments and a final course project. Much of the background and materials of this course will be drawn from the [ImageNet Challenge](#).

Course Instructors

Fei-Fei Li

Andrej Karpathy

Justin Johnson

Teaching Assistants

Serena Yeung

Subhasis Das

Song Han

Albert Haque

Bharath Ramsundar

Hieu Pham

Irwan Bello

Namrata Anand

Lane McIntosh

Catherine Dong

Kyle Griswold

Course Notes

Detailed Syllabus

Thank you!