Lecture 7
Convolutional Neural Networks

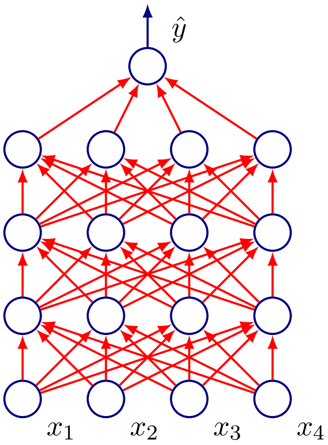
CMSC 35246: Deep Learning

Shubhendu Trivedi
&
Risi Kondor

University of Chicago

April 17, 2017

We saw before:

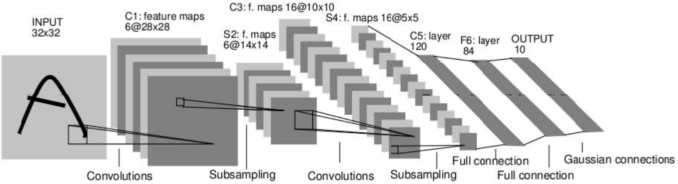


- A series of matrix multiplications:
- $\mathbf{x} \mapsto W_1^T\mathbf{x} \mapsto \mathbf{h}_1 = f(W_1^T\mathbf{x}) \mapsto W_2^T\mathbf{h}_1 \mapsto \mathbf{h}_2 = f(W_2^T\mathbf{h}_1) \mapsto W_3^T\mathbf{h}_2 \mapsto \mathbf{h}_3 = f(W_3^T\mathbf{h}_3) \mapsto W_4^T\mathbf{h}_3 = \hat{y}$

Convolutional Networks

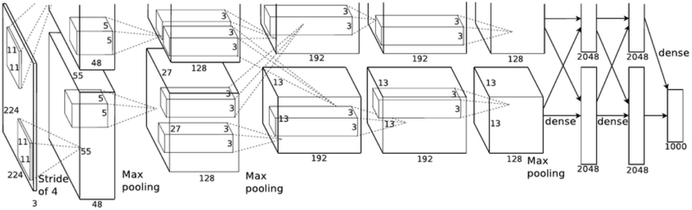
- Neural Networks that use convolution in place of general matrix multiplication in atleast one layer
- Next:
  - What is convolution?
  - What is pooling?
  - What is the motivation for such architectures (remember LeNet?)

LeNet-5 (LeCun, 1998)



- The original Convolutional Neural Network model goes back to 1989 (LeCun)

AlexNet (Krizhevsky, Sutskever, Hinton 2012)



- ImageNet 2012 15.4% error rate

Convolutional Neural Networks

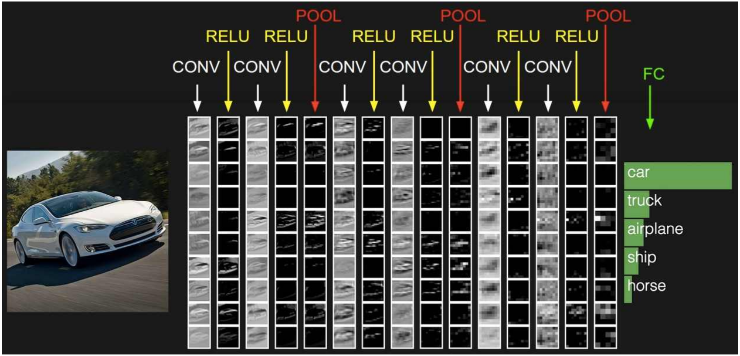
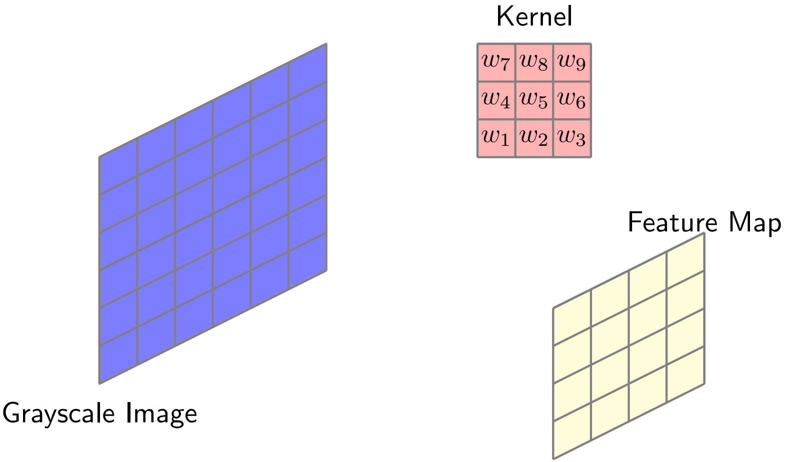


Figure: Andrej Karpathy

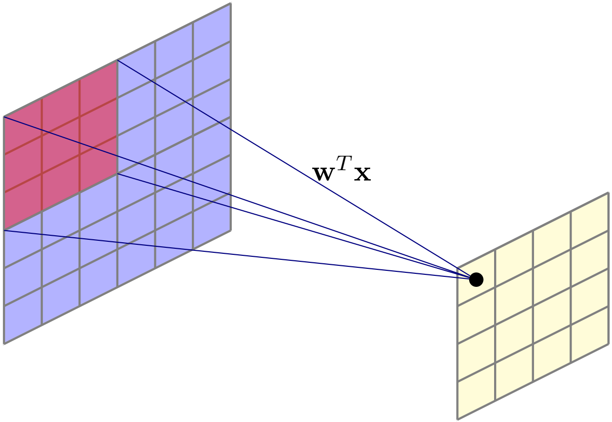
Now let's deconstruct them...

Convolution

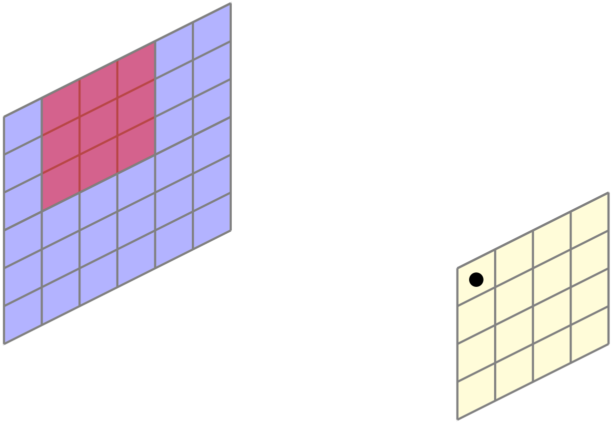


- Convolve image with kernel having weights $\mathbf{w}$ (learned by backpropagation)

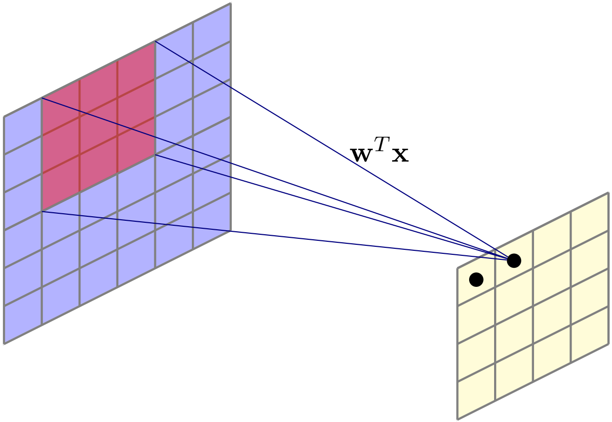
Convolution



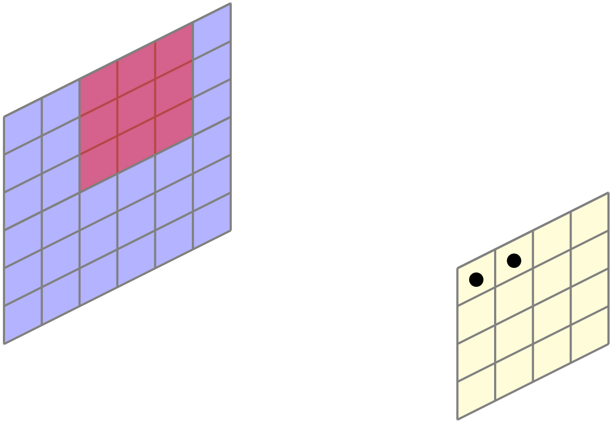
Convolution



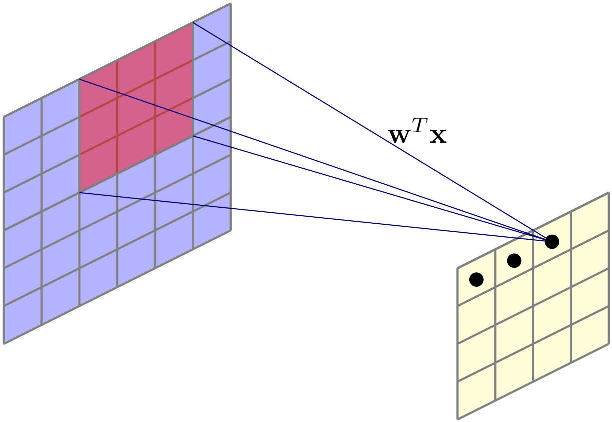
Convolution



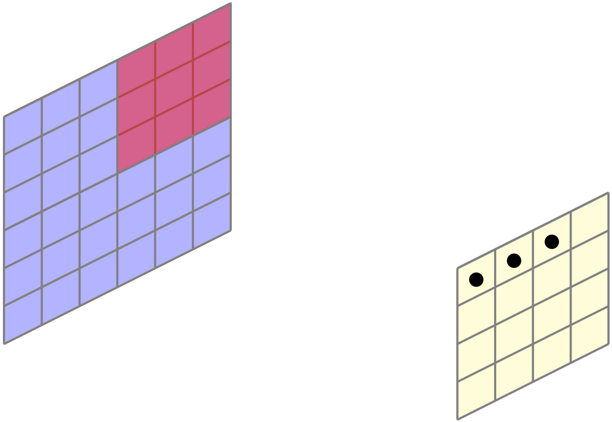
Convolution



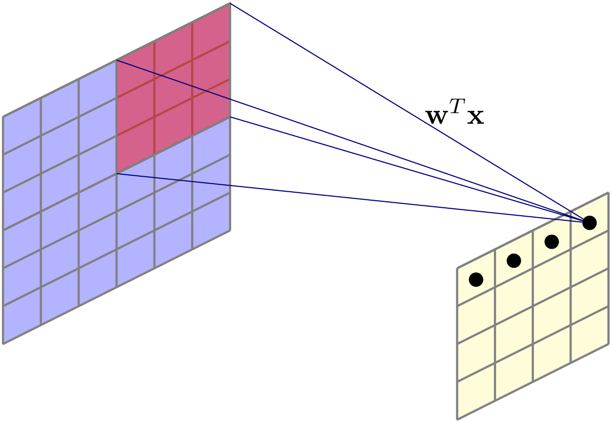
Convolution



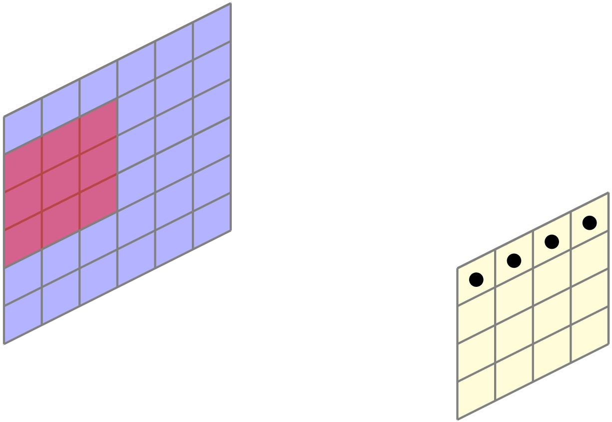
Convolution



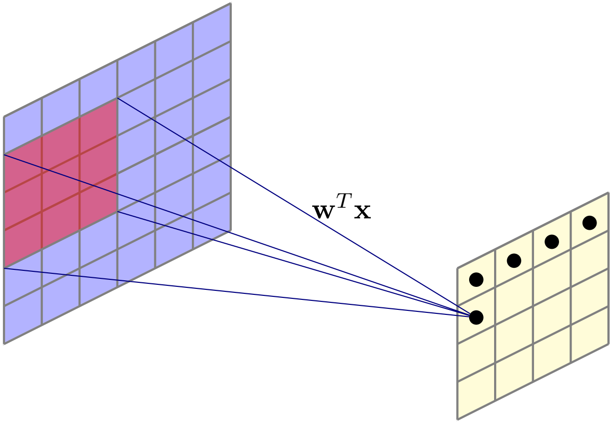
Convolution



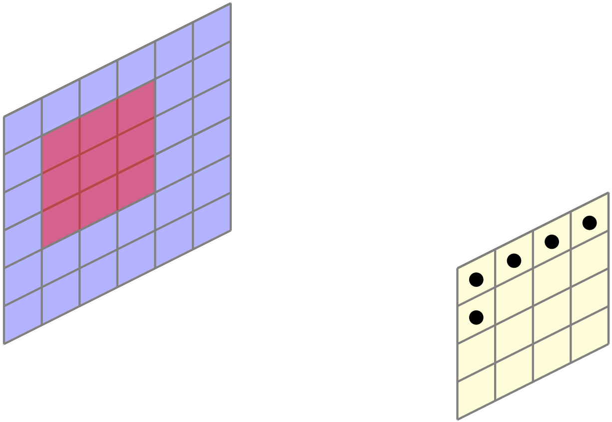
Convolution



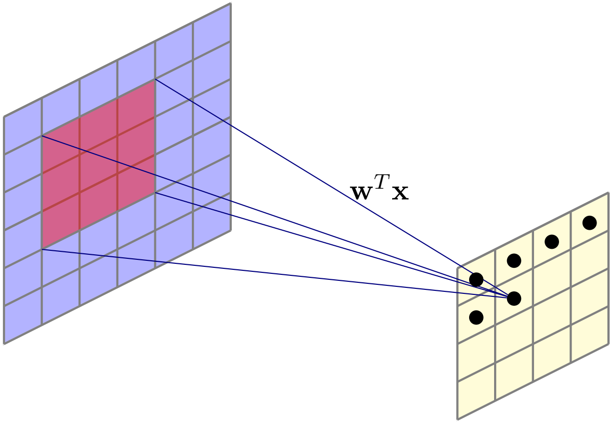
Convolution



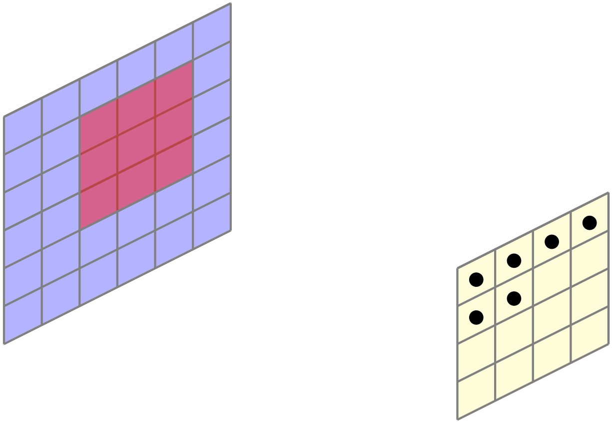
Convolution



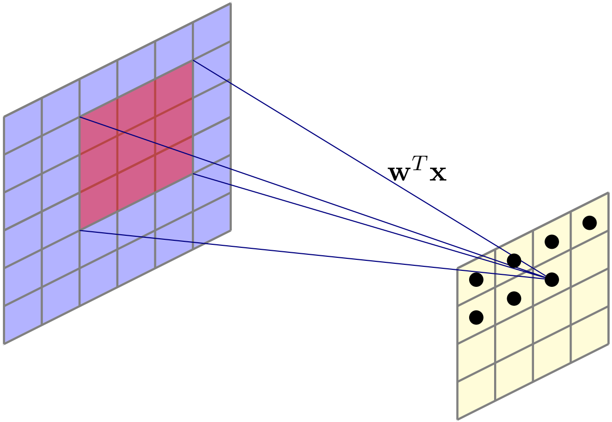
Convolution



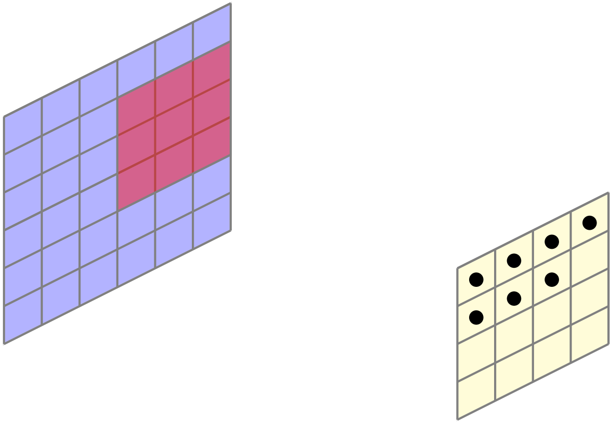
Convolution



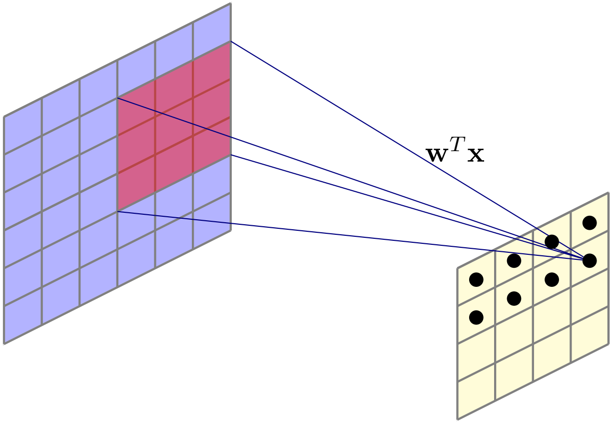
Convolution



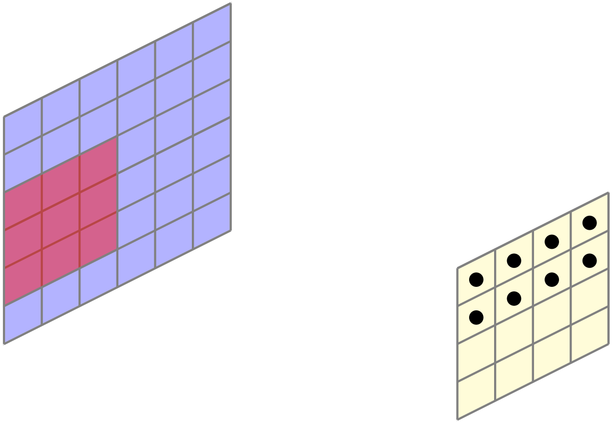
Convolution



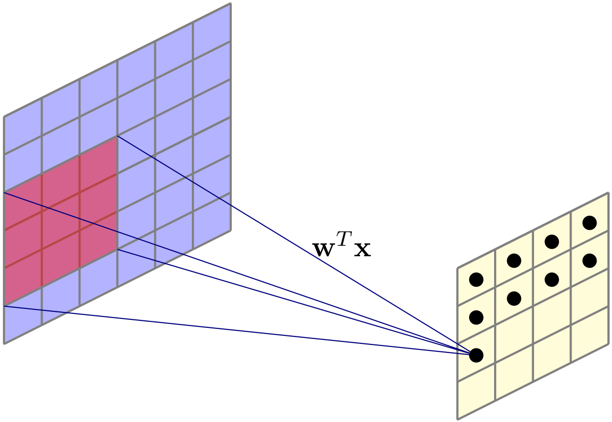
Convolution



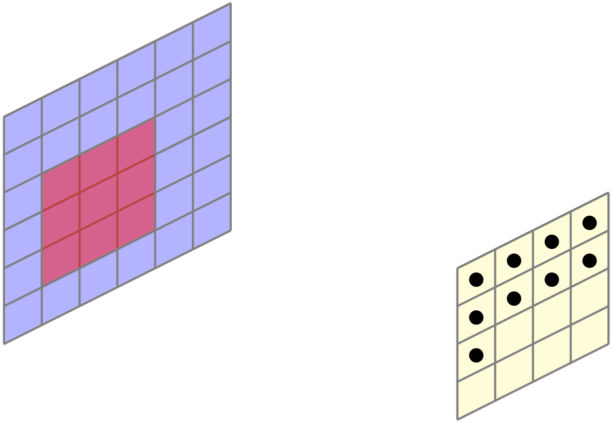
Convolution



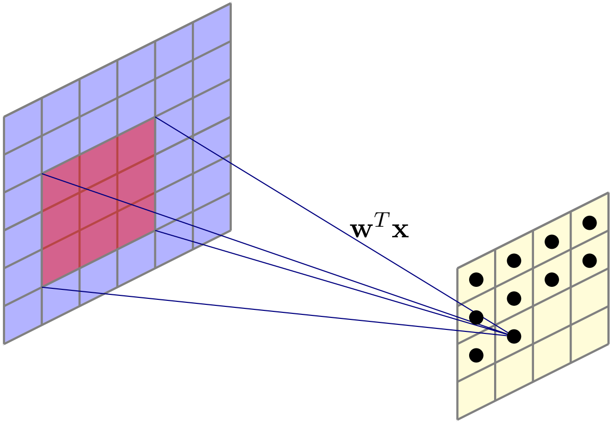
Convolution



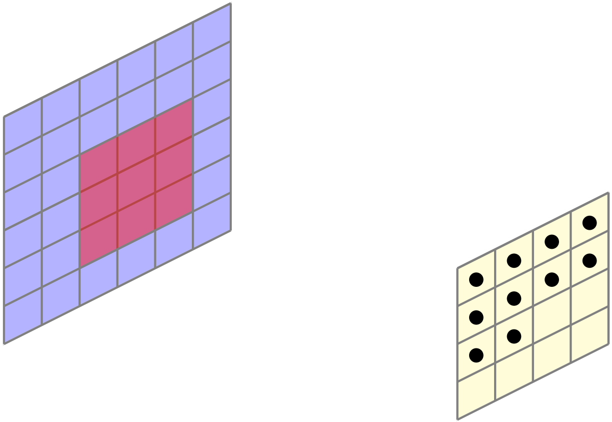
Convolution



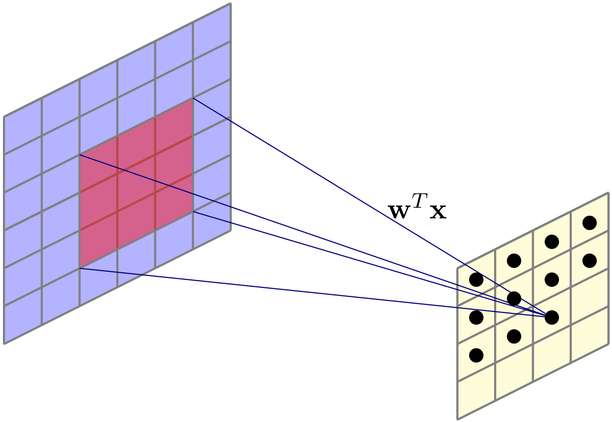
Convolution



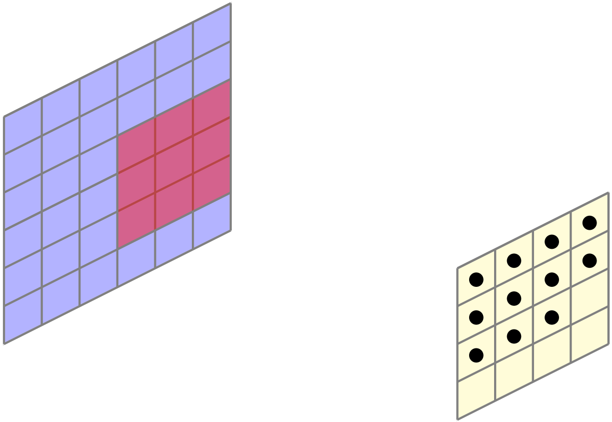
Convolution



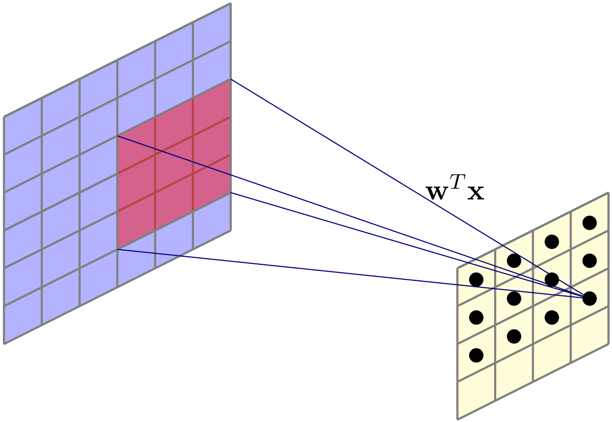
Convolution



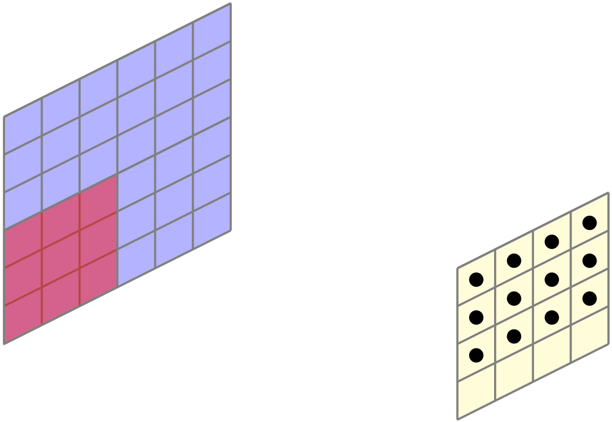
Convolution



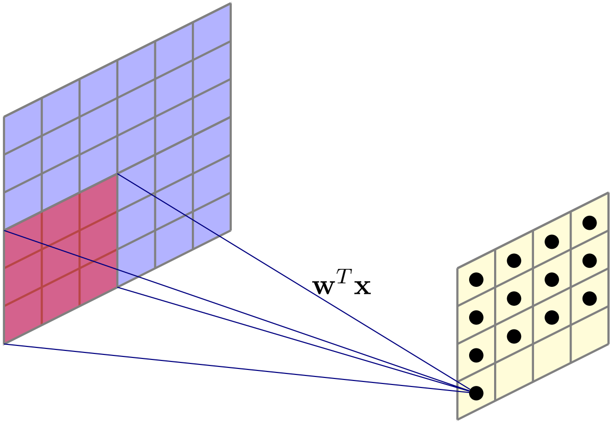
Convolution



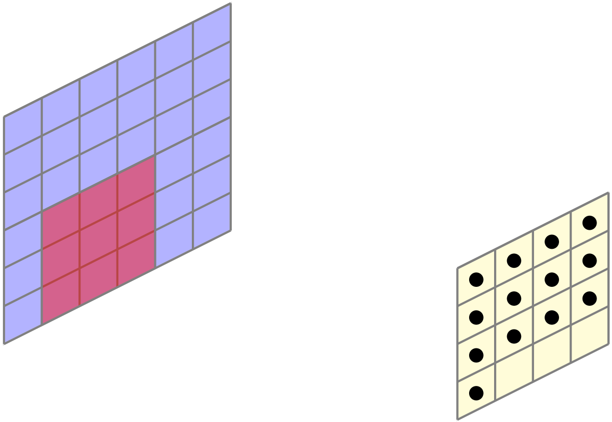
Convolution



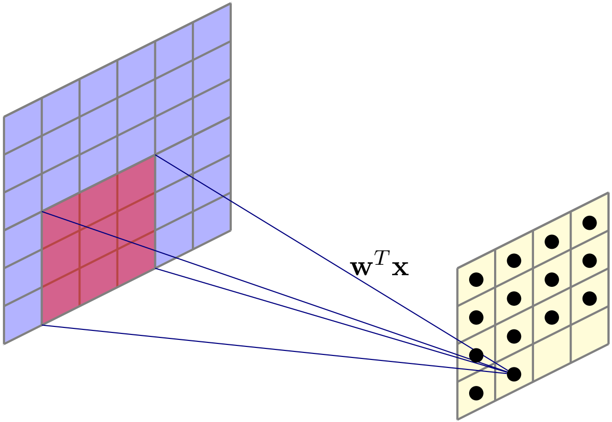
Convolution



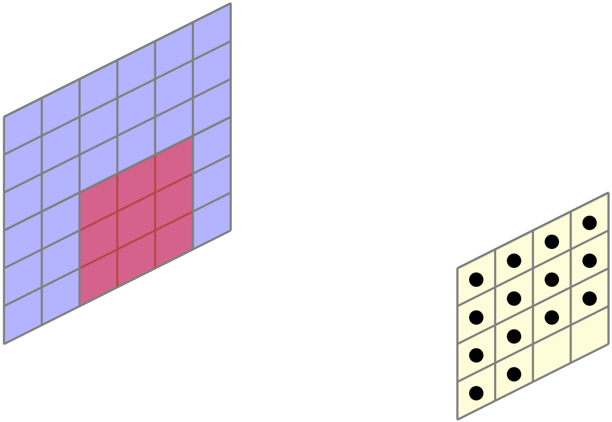
Convolution



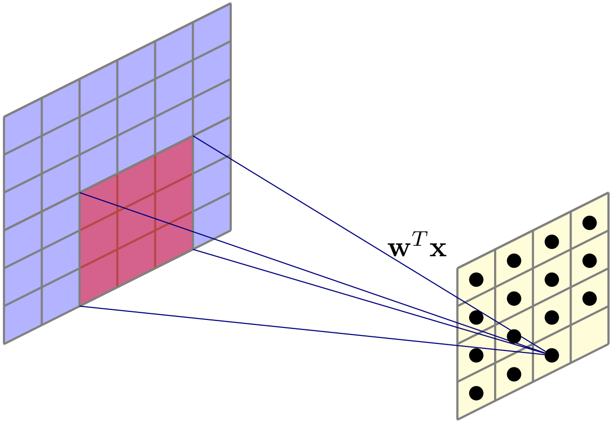
Convolution



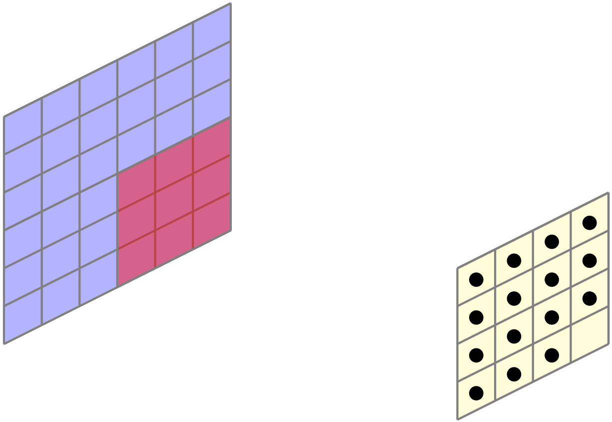
Convolution



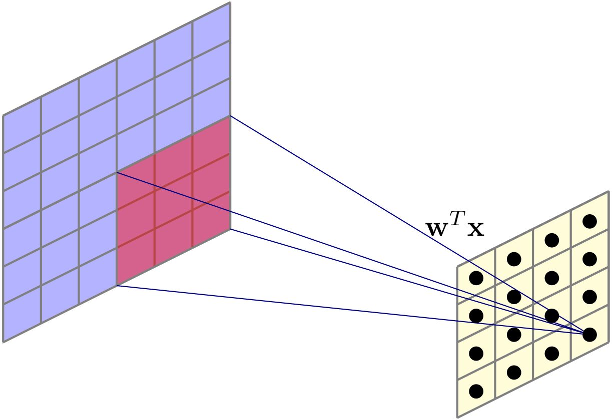
Convolution



Convolution



Convolution



- What is the number of parameters?

Output Size

- We used stride of 1, kernel with receptive field of size 3 by 3
- Output size:

$$\frac{N-K}{S}+1$$

- In previous example: $N = 6, K = 3, S = 1$, Output size $= 4$
- For $N = 8, K = 3, S = 1$, output size is 6

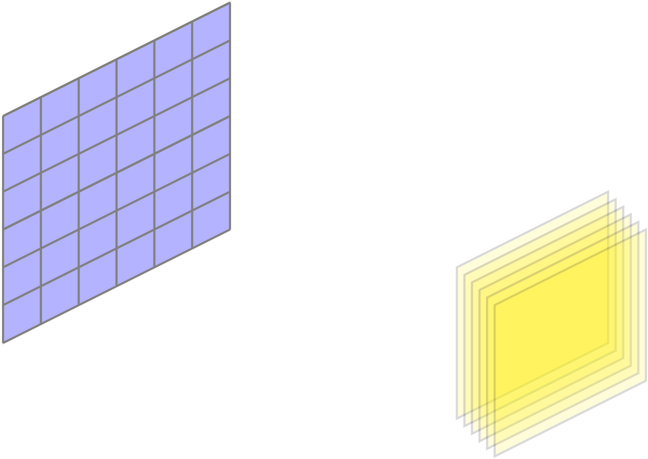
Zero Padding

- Often, we want the output of a convolution to have the same size as the input. Solution: Zero padding.
- In our previous example:

0	0	0	0	0	0	0	0
0							0
0							0
0							0
0							0
0							0
0							0
0	0	0	0	0	0	0	0

- Common to see convolution layers with stride of 1, filters of size K, and zero padding with $\frac{K-1}{2}$ to preserve size

Learn Multiple Filters



Learn Multiple Filters

- If we use 100 filters, we get 100 feature maps

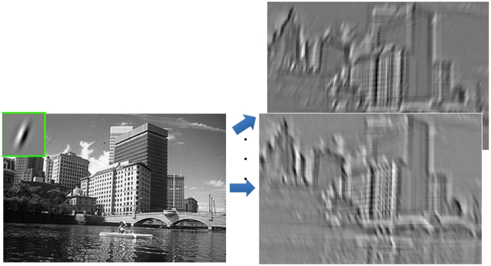


Figure: I. Kokkinos

In General

- We have only considered a 2-D image as a running example
- But we could operate on volumes (e.g. RGB Images would be depth 3 input, filter would have same depth)

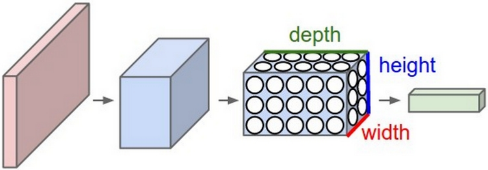


Image from Wikipedia

In General: Output Size

- For convolutional layer:
  - Suppose input is of size $W_1 \times H_1 \times D_1$
  - Filter size is K and stride S
  - We obtain another volume of dimensions $W_2 \times H_2 \times D_2$
  - As before:

$$W_2 = \frac{W_1 - K}{S} + 1 \text{ and } H_2 = \frac{H_1 - K}{S} + 1$$

  - Depths will be equal

Convolutional Layer Parameters

Example volume: $28 \times 28 \times 3$ (RGB Image)
100 3×3 filters, stride 1
What is the zero padding needed to preserve size?
Number of parameters in this layer?
For every filter: $3 \times 3 \times 3 + 1 = 28$ parameters
Total parameters: $100 \times 28 = 2800$

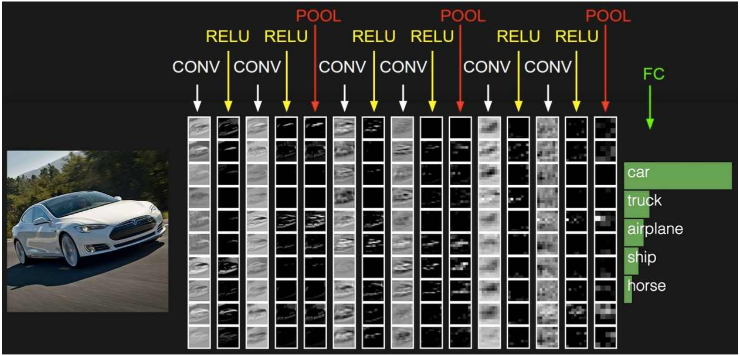
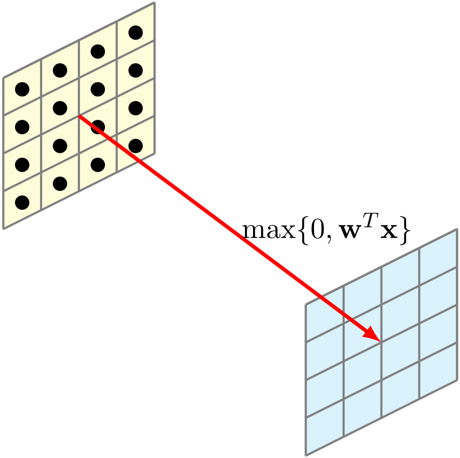


Figure: Andrej Karpathy

Non-Linearity



- After obtaining feature map, apply an elementwise non-linearity to obtain a transformed feature map (same size)

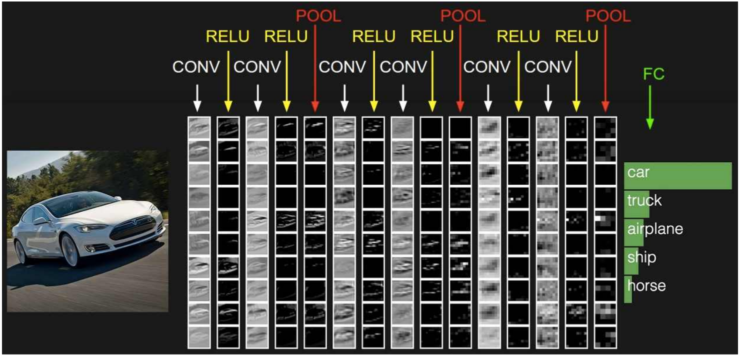
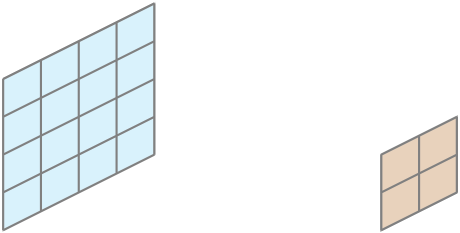
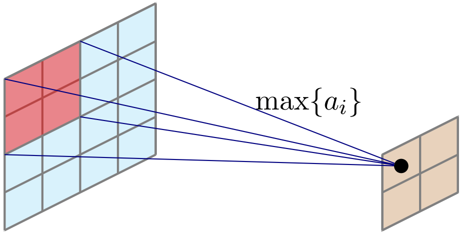


Figure: Andrej Karpathy

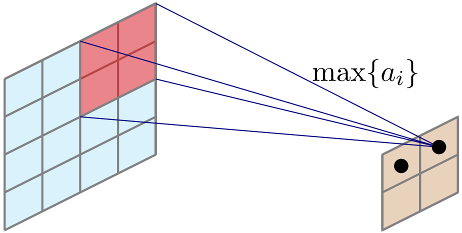
Pooling



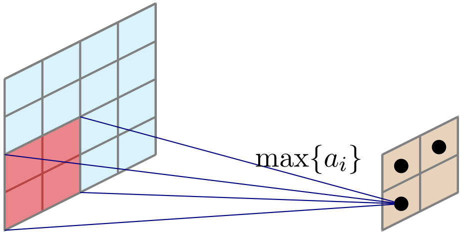
Pooling



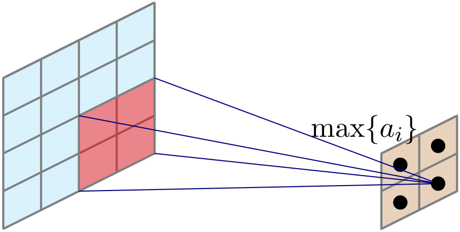
Pooling



Pooling

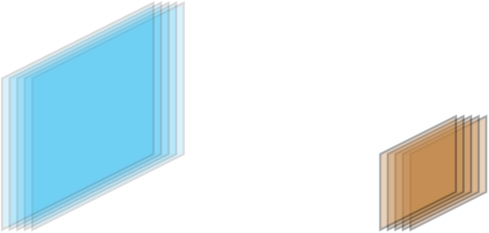


Pooling



- Other options: Average pooling, L2-norm pooling, random pooling

Pooling



- We have multiple feature maps, and get an equal number of subsampled maps
- This changes if cross channel pooling is done

So what's left: Fully Connected Layers

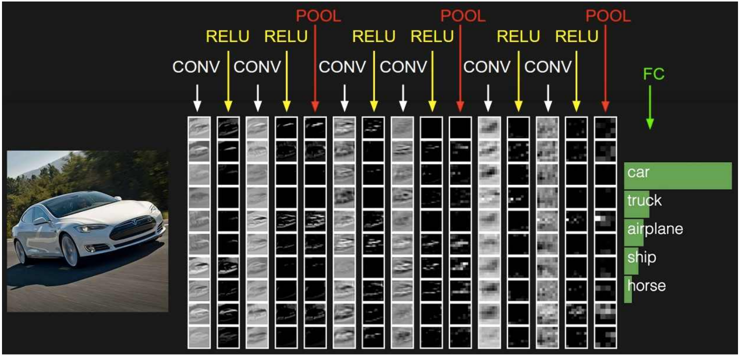
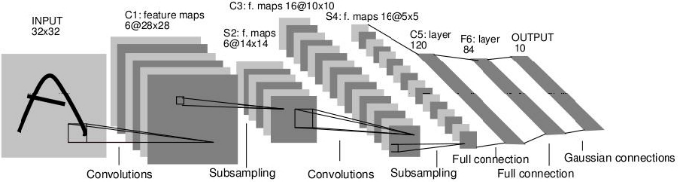


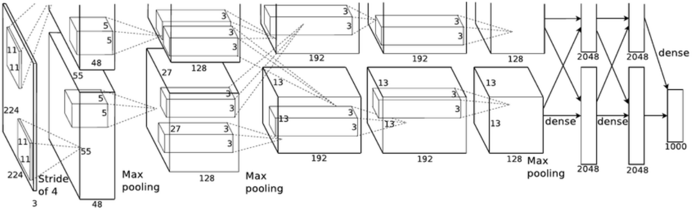
Figure: Andrej Karpathy

LeNet-5



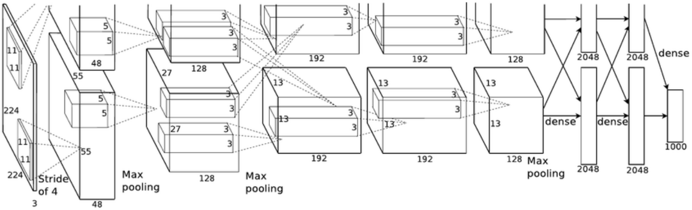
- Filters are of size 5×5, stride 1
- Pooling is 2×2, with stride 2
- How many parameters?

AlexNet



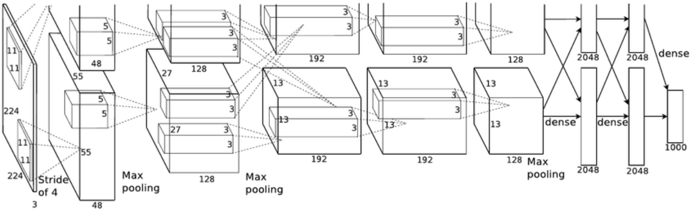
- Input image: 227 X 227 X 3
- First convolutional layer: 96 filters with K = 11 applied with stride = 4
- Width and height of output: $\frac{227-11}{4} + 1 = 55$

AlexNet



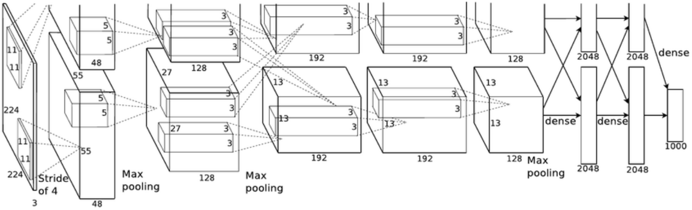
- Number of parameters in first layer?
- 11 X 11 X 3 X 96 = 34848

AlexNet



- Next layer: Pooling with 3 X 3 filters, stride of 2
- Size of output volume: 27
- Number of parameters?

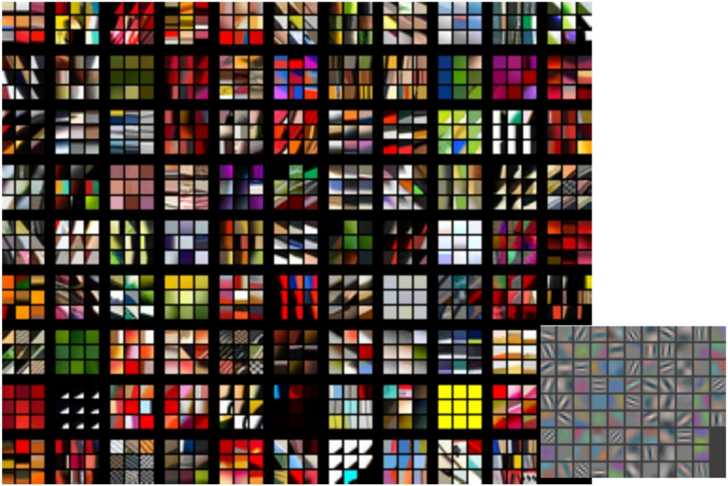
AlexNet



- Popularized the use of ReLUs
- Used heavy data augmentation (flipped images, random crops of size 227 by 227)
- Parameters: Dropout rate 0.5, Batch size = 128, Weight decay term: 0.0005 ,Momentum term $\alpha = 0.9$, learning rate η = 0.01, manually reduced by factor of ten on monitoring validation loss.

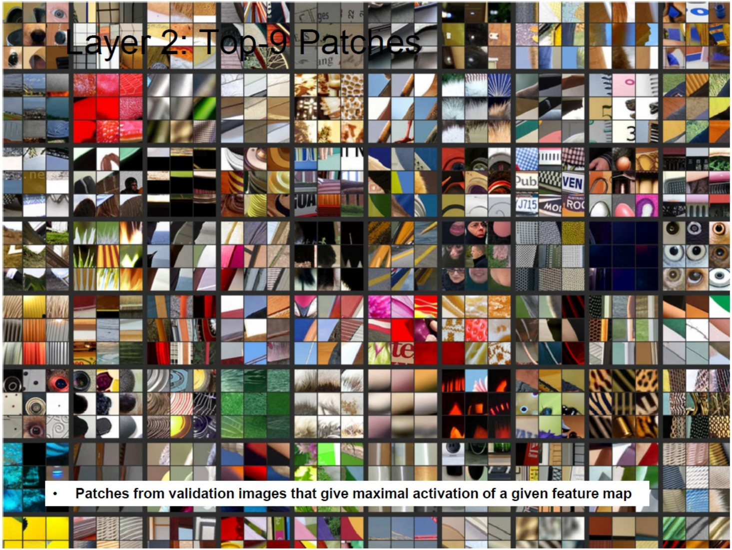
Short Digression: How do the features look like?

Layer 1 filters

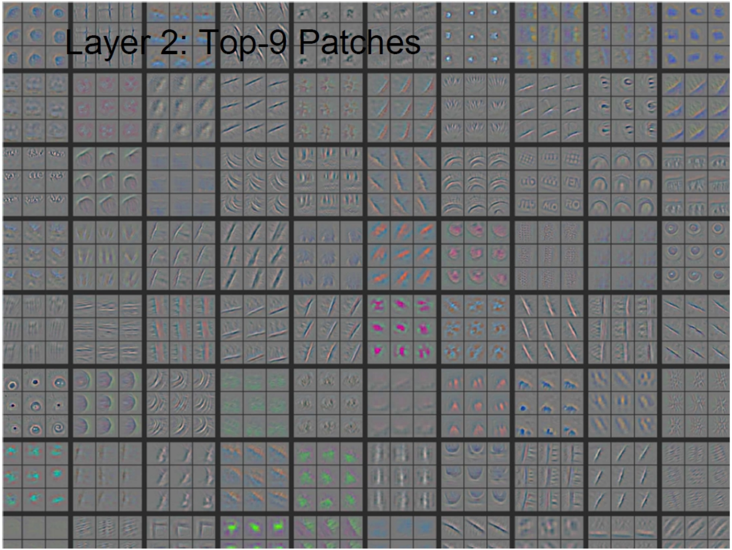


This and the next few illustrations are from Rob Fergus

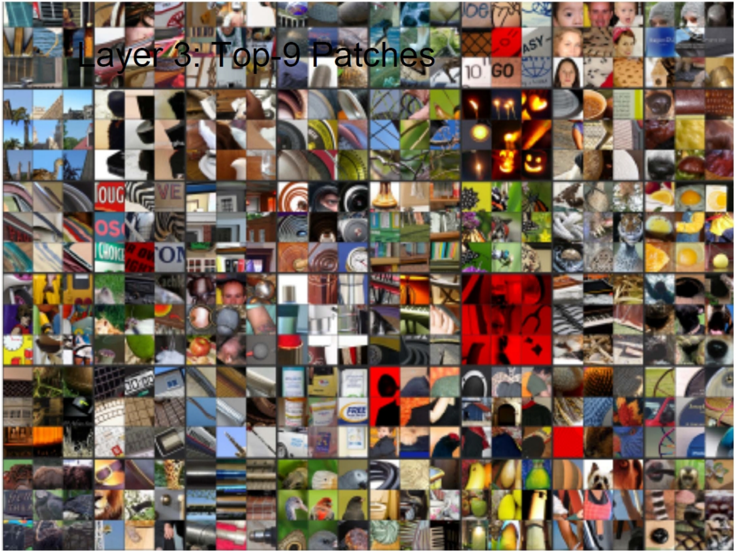
Layer 2 Patches



Layer 2 Patches

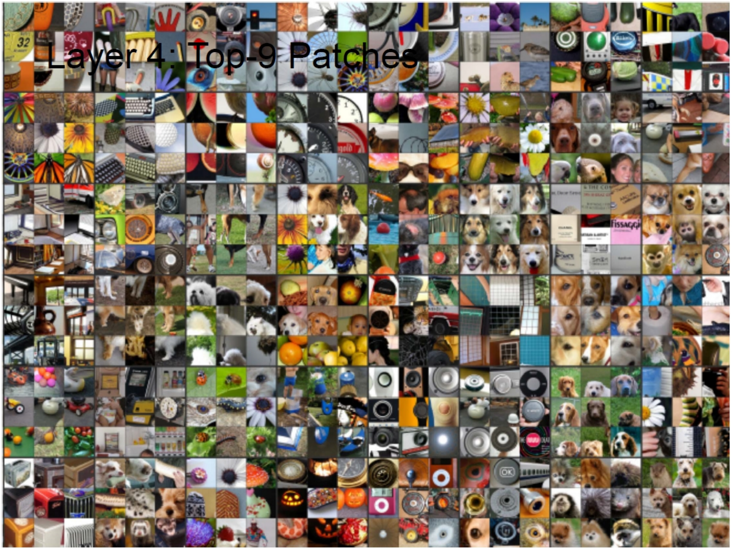


Layer 3 Patches



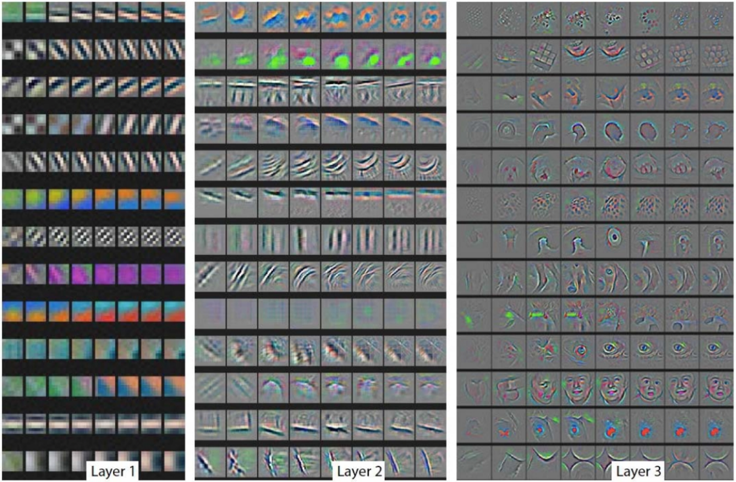
Layer 3 Patches

Layer 4 Patches

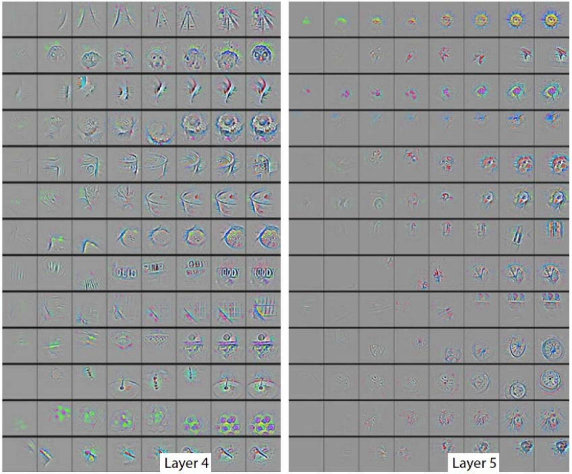


Layer 4 Patches

Evolution of Filters



Evolution of Filters



Caveat?

Back to Architectures

ImageNet 2013

- Was won by a network similar to AlexNet (Matthew Zeiler and Rob Fergus)
- Changed the first convolutional layer from 11 X 11 with stride of 4, to 7 X 7 with stride of 2
- AlexNet used 384, 384 and 256 layers in the next three convolutional layers, ZF used 512, 1024, 512
- ImageNet 2013: 14.8 % (reduced from 15.4 %) (top 5 errors)

VGGNet(Simonyan and Zisserman, 2014)

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

- Best model: Column D.
- Error: 7.3 % (top five error)

VGGNet(Simonyan and Zisserman, 2014)

- Total number of parameters: 138 Million (calculate!)
- Memory (Karpathy): 24 Million X 4 bytes $\approx$ 93 MB per image
- For backward pass the memory usage is doubled per image
- Observations:
  - Early convolutional layers take most memory
  - Most parameters are in the fully connected layers

Going Deeper

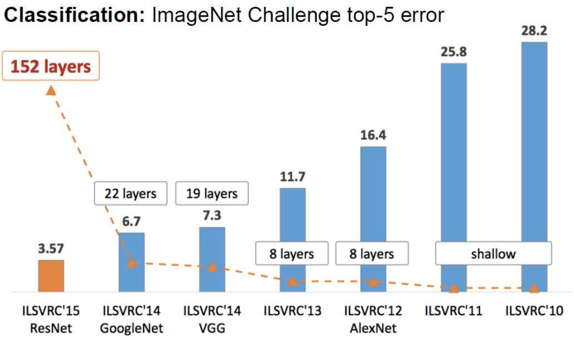
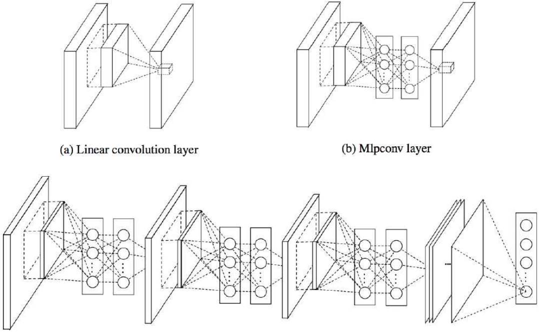


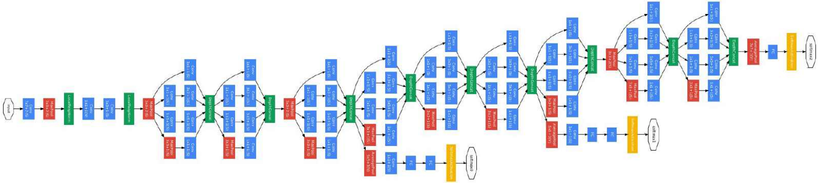
Figure: Kaiming He, MSR

Network in Network



M. Lin, Q. Chen, S. Yan, Network in Network, ICLR 2014

Google LeNet

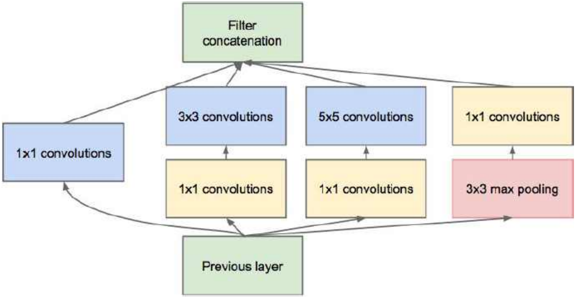


C.

Szegedy et al, Going Deeper With Convolutions, CVPR 2015

- Error: 6.7 % (top five error)

The Inception Module



- Parallel paths with different receptive field sizes - capture sparse patterns of correlation in stack of feature maps
- Also include auxiliary classifiers for ease of training
- Also note 1 by 1 convolutions

Google LeNet

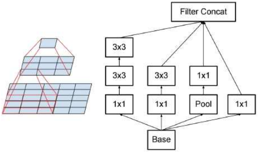
type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	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								

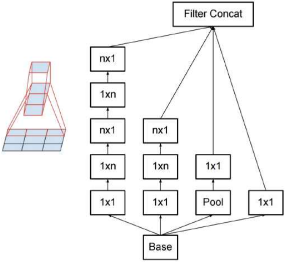
C. Szegedy et al, Going Deeper With Convolutions, CVPR 2015

Google LeNet

- Has 5 Million or 12X fewer parameters than AlexNet
- Gets rid of fully connected layers

Inception v2, v3





C. Szegedy et al, Rethinking the Inception Architecture for Computer Vision, CVPR 2016

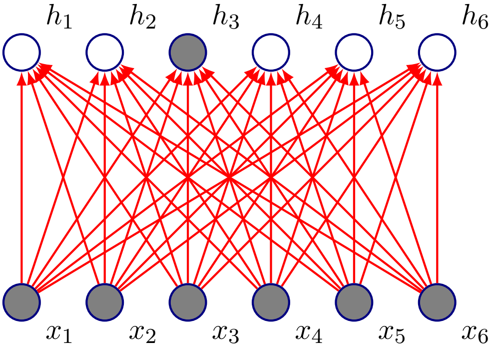
- Use Batch Normalization during training to reduce dependence on auxiliary classifiers
- More aggressive factorization of filters

Why do CNNs make sense? (Brain Stuff next time)

Convolutions: Motivation

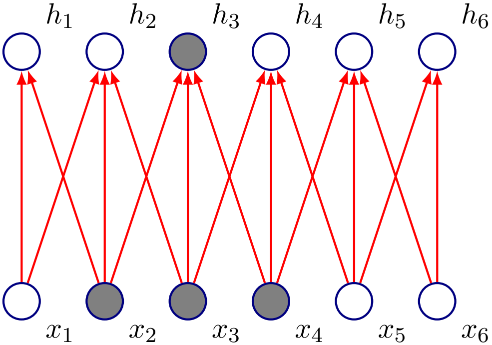
- Convolution leverages four ideas that can help ML systems:
  - Sparse interactions
  - Parameter sharing
  - Equivariant representations
  - Ability to work with inputs of variable size
- **Sparse Interactions**
  - Plain Vanilla NN ($y \in \mathbb{R}^n, x \in \mathbb{R}^m$): Need matrix multiplication $y = \mathbf{W}x$ to compute activations for each layer (every output interacts with every input)
  - Convolutional networks have *sparse interactions* by making kernel smaller than input
  - $\implies$ need to store fewer parameters, computing output needs fewer operations ($O(m \times n)$ versus $O(k \times n)$)

Motivation: Sparse Connectivity



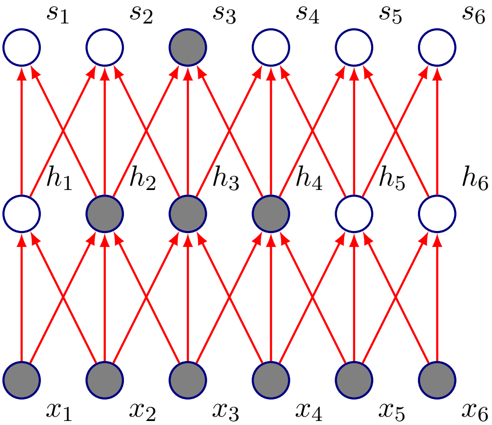
- Fully connected network: h_3 is computed by full matrix multiplication with no sparse connectivity

Motivation: Sparse Connectivity



- Kernel of size 3, moved with stride of 1
- h_3 only depends on x_2, x_3, x_4

Motivation: Sparse Connectivity



- Connections in CNNs are sparse, but units in deeper layers are connected to all of the input (larger receptive field sizes)

Motivation: Parameter Sharing

- Plain vanilla NN: Each element of $\mathbf{W}$ is used exactly once to compute output of a layer
- In convolutional networks, parameters are *tied*: weight applied to one input is tied to value of a weight applied elsewhere
- Same kernel is used throughout the image, so instead learning a parameter for each location, only a set of parameters is learnt
- Forward propagation remains unchanged $O(k \times n)$
- Storage improves dramatically as $k \ll m, n$

Motivation: Equivariance

- Let's first formally define convolution:

$$s(t) = (x * w)(t) = \int x(a)w(t-a)da$$

- In Convolutional Network terminology x is referred to as **input**, w as the **kernel** and s as the **feature map**
- Discrete Convolution:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

- Convolution is commutative, thus:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

Aside

- The latter is usually more straightforward to implement in ML libraries (less variation in range of valid values of m and n)
- Neither are usually used in practice in Neural Networks
- Libraries implement *Cross Correlation*, same as convolution, but without flipping the kernel

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n)$$

Motivation: Equivalence

- Equivariance: f is equivariant to g if $f(g(\mathbf{x})) = g(f(\mathbf{x}))$
- The form of parameter sharing used by CNNs causes each layer to be equivariant to translation
- That is, if g is any function that translates the input, the convolution function is equivariant to g

Motivation: Equivariance

- Implication: While processing time series data, convolution produces a timeline that shows when different features appeared (if an event is shifted in time in the input, the same representation will appear in the output)
- Images: If we move an object in the image, its representation will move the same amount in the output
- This property is useful when we know some local function is useful everywhere (e.g. edge detectors)
- Convolution is not equivariant to other operations such as change in scale or rotation

Pooling: Motivation

- Pooling helps the representation become slightly *invariant* to small translations of the input
- Reminder: Invariance: $f(g(\mathbf{x})) = f(\mathbf{x})$
- If input is translated by small amount: values of most pooled outputs don't change

Pooling: Invariance

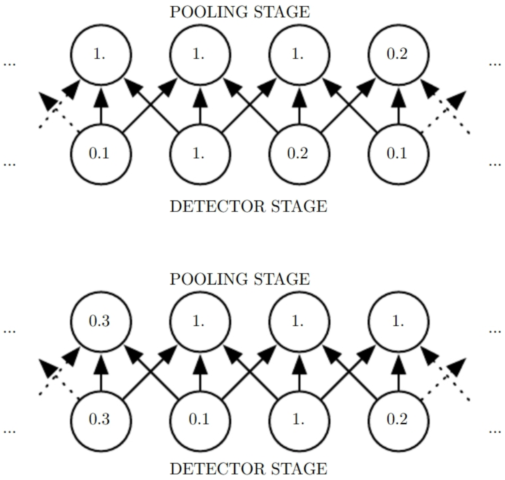


Figure: Goodfellow *et al.*

Pooling

- Invariance to local translation can be useful if we care more about whether a certain feature is present rather than exactly where it is
- Pooling over spatial regions produces invariance to translation, what if we pool over separately parameterized convolutions?
- Features can learn which transformations to become invariant to (Example: Maxout Networks, Goodfellow *et al* 2013)
- One more advantage: Since pooling is used for downsampling, it can be used to handle inputs of varying sizes

Next time

- More Architectures
- Variants on the CNN idea
- More motivation
- Group Equivariance
- Equivariance to Rotation

Quiz!