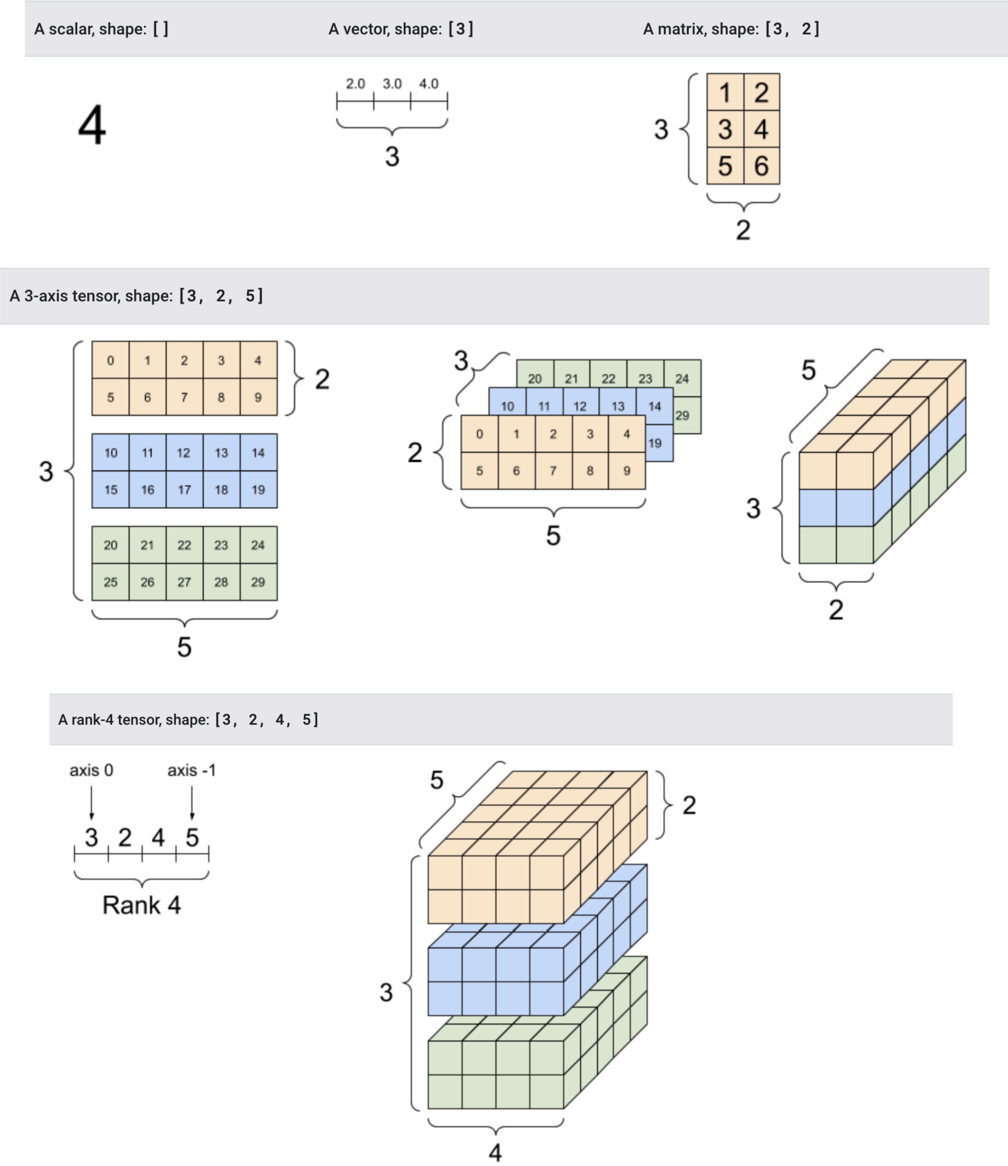
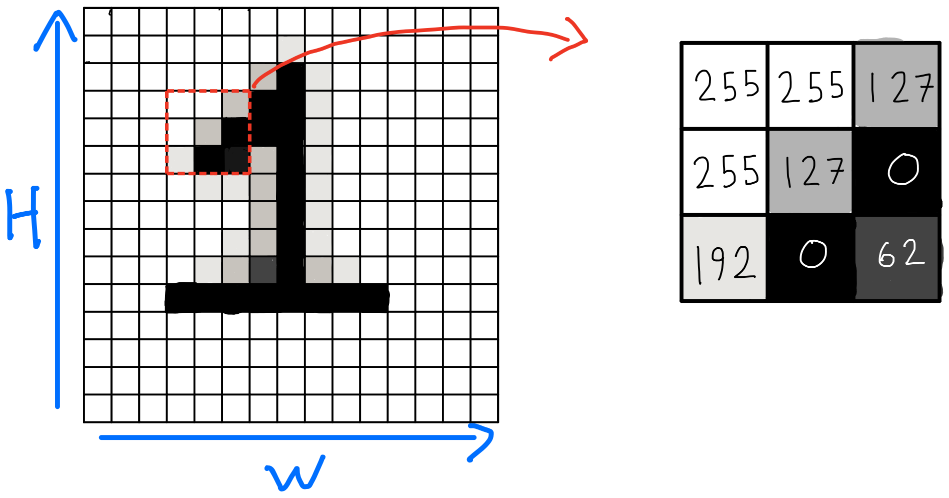
Reminder: Scalars, Vectors, matrices & Tensors

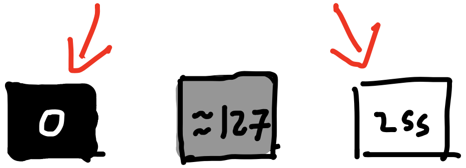


Representing Images



- Images composed of pixel features

- Typically 8-bit Values 0 – 255 (int)

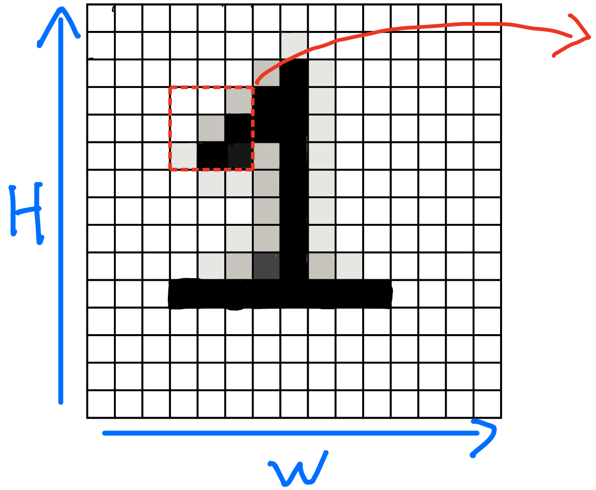


0 → Black 255 → White

Single obs x is a Matrix: $H \times W$

Batch X is a 3-D Tensor: $N \times H \times W$

Representing Images



255	255	127
255	127	0
192	0	62

0 to 1

good

1.	1.	0.5
1.	0.5	0.
0.7	0.	0.2

-1 to 1

better

1.	1.	0.
1.	0.	-1.
1.2	-1.	-.5

Norm

best!

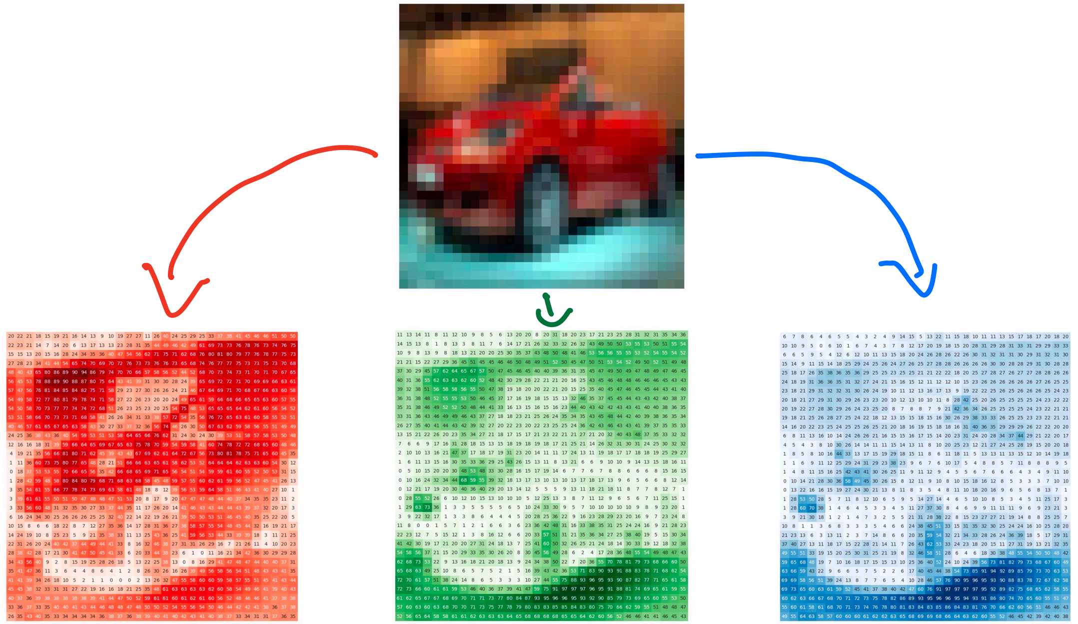
[But dataset specific :(]

3.1	3.1	0.1
3.1	0.1	-3.
1.4	-3.	-1.1

For Neural networks want to convert to float and rescale

RGB Images

Represent color pixels w/ 3-Channels
[3-Values per pixel]



Single obs. x is 3-D tensor: $H \times W \times 3$

Batch $\mathbb{X}$ is 4-D tensor: $N \times H \times W \times 3$

Purple
$\boxed{240}$ High red
$\boxed{10}$ Low green
$\boxed{200}$ High blue

White
$\boxed{255}$ Max red
$\boxed{255}$ Max green
$\boxed{255}$ Max blue

black
$\boxed{0}$ No red
$\boxed{0}$ No green
$\boxed{0}$ No blue

RGB Images - PyTorch

Normal

Single obs. x is 3-D tensor: $H \times W \times 3$

Batch X is 4-D tensor: $N \times H \times W \times 3$

PyTorch

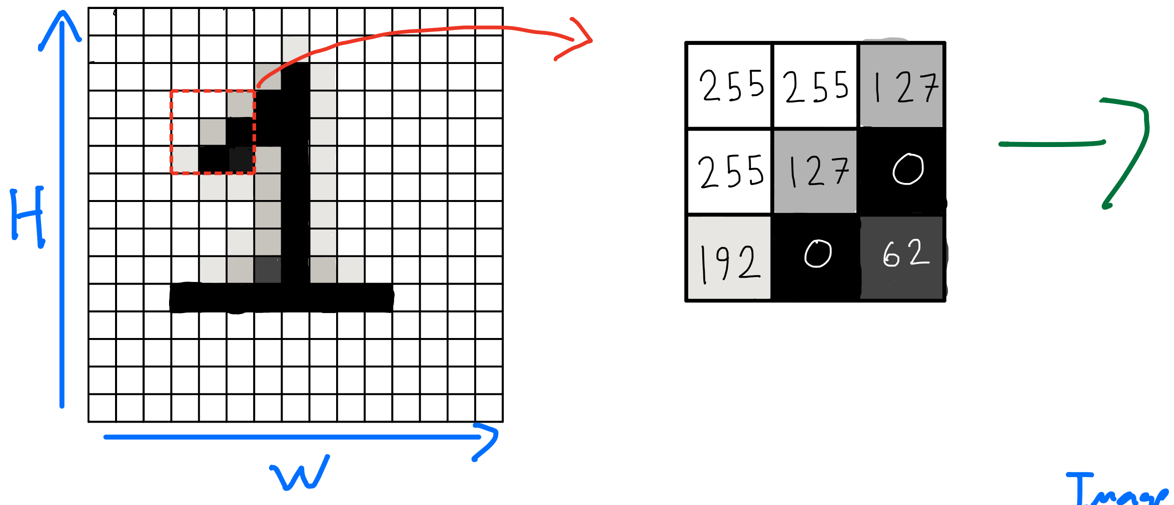
Single obs. x is 3-D tensor: $3 \times H \times W$

Batch X is 4-D tensor: $N \times 3 \times H \times W$

X.Permute(0, 3, 1, 2)

Representing Images (so far)

For standard neural network Reshape into vector



Single obs.

$x : \overbrace{HW}^{\text{Pixels}}$

255	255	127	255	127	0	192	0	64	...

Pixels

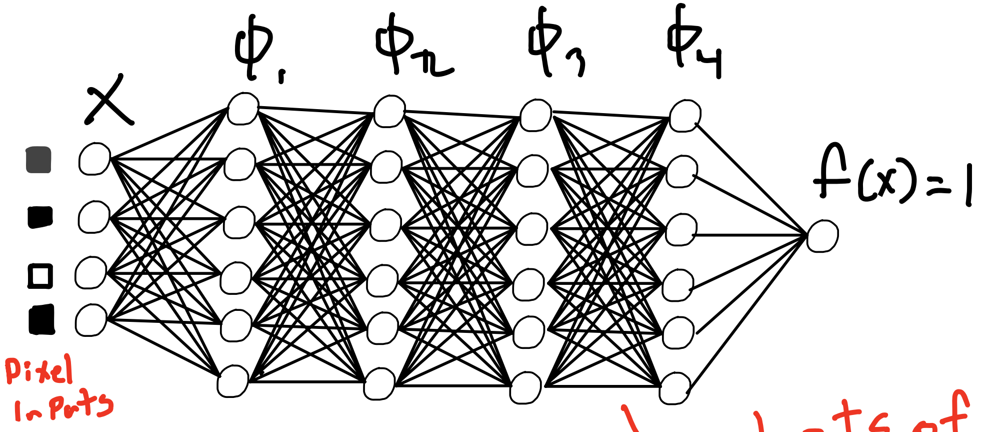
Batch

$X : N \times \overbrace{HW}^{\text{Pixels}}$

Image 1	255	255	127	255	127	0	192	0	64
Image 2	255	255	127	255	127	0	192	0	64
Image 3	255	255	127	255	127	0	192	0	64
⋮	255	255	127	255	127	0	192	0	64

N

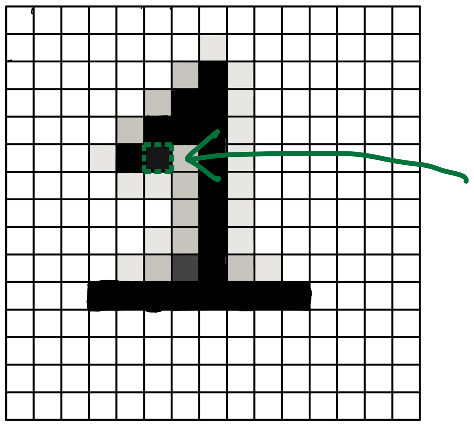
Pixels



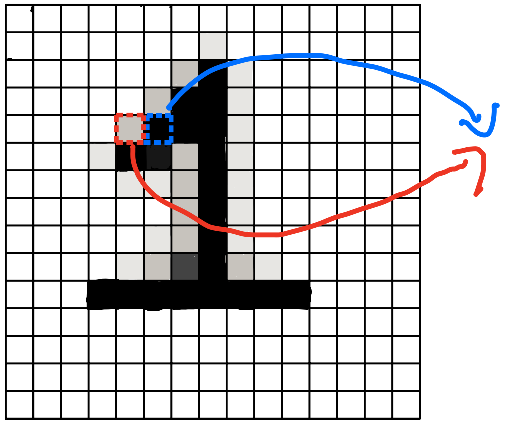
Lots of parameters!

Image Structure

- Class determined by relationships between features



- Relationships between nearby pixels are more important than far pixels



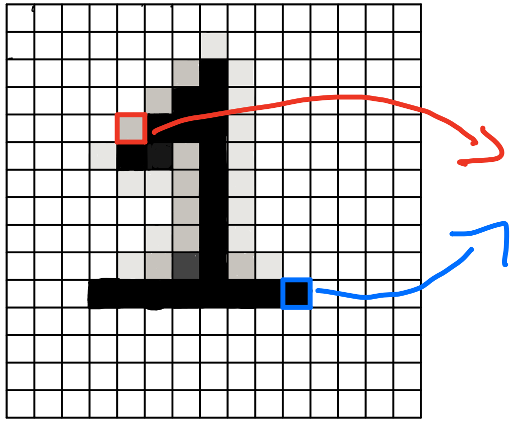
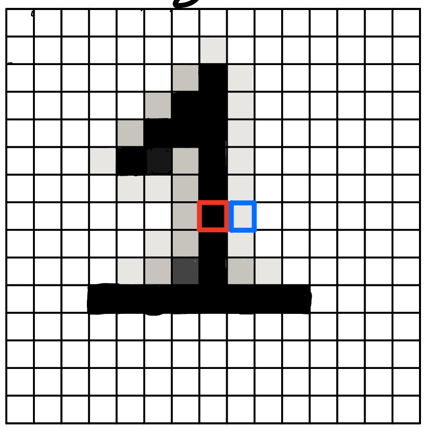


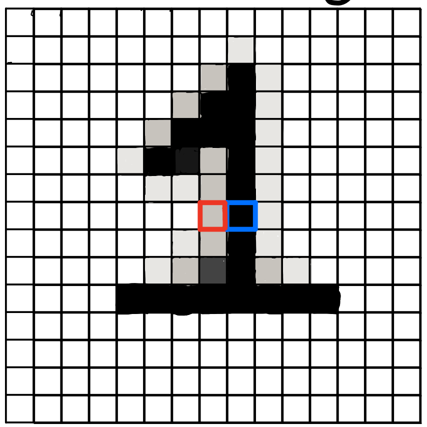
Image Structure

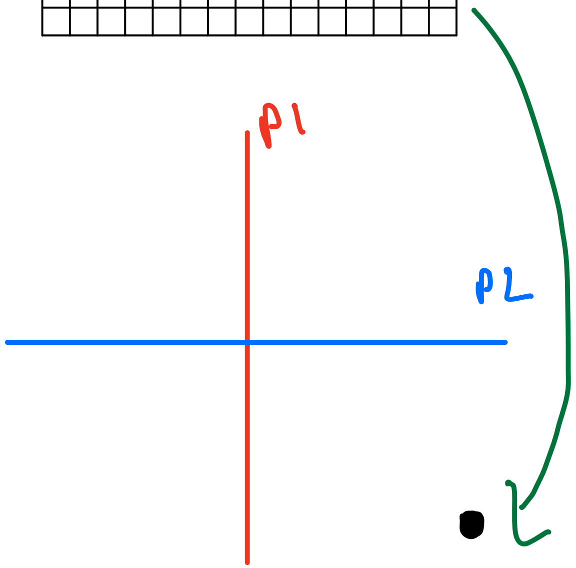
original

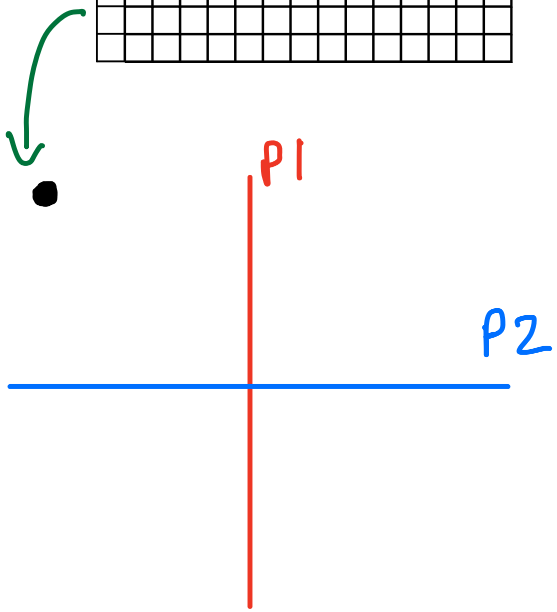


Translation shouldn't change prediction

shifted right

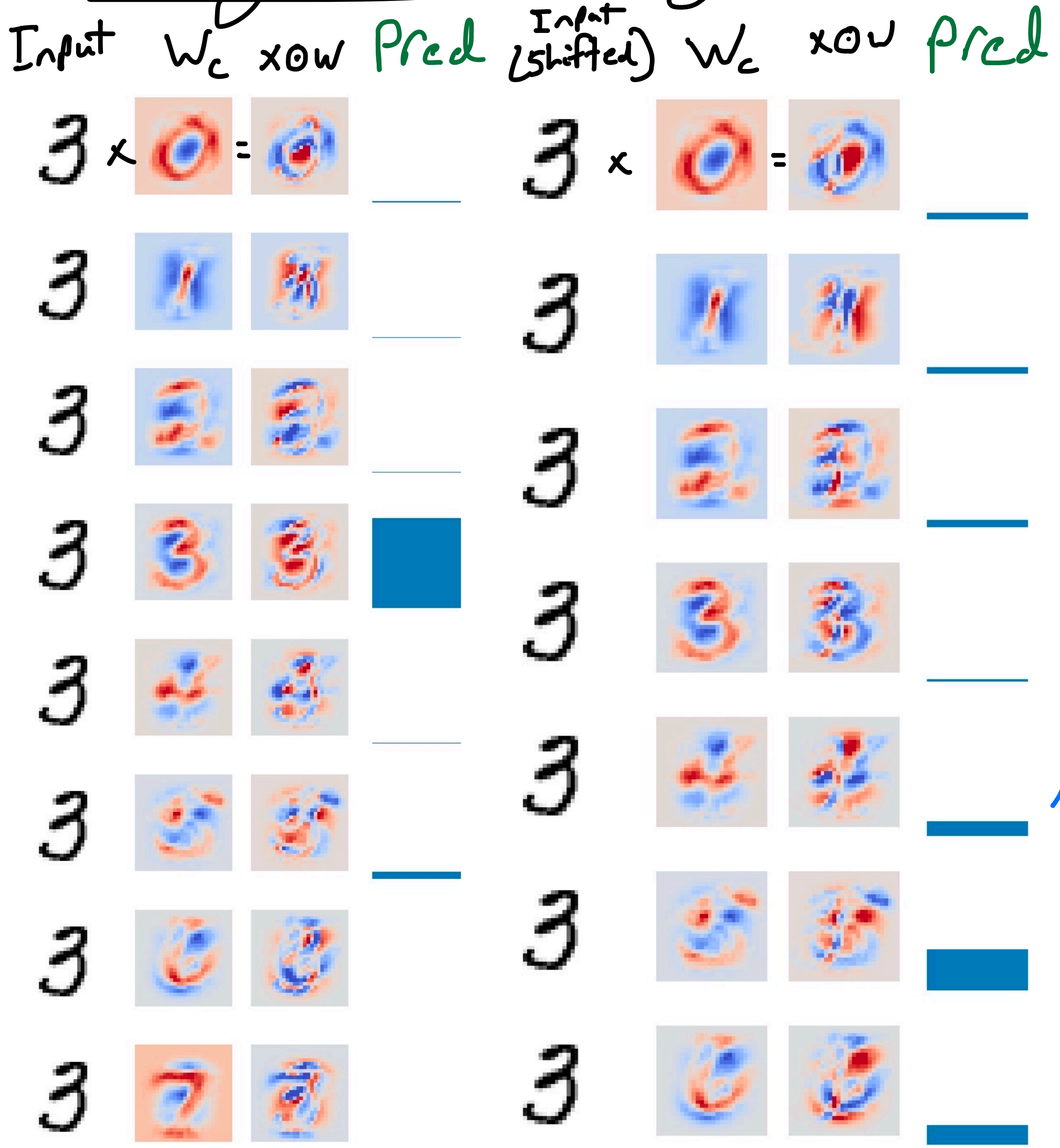






But features can change a lot!

Logistic Regression



- Regression weights from HW 3
- Shifting Image Changes predicted class!

Remember prediction function for multiclass Classification

→ Class w/ max output

Pred. $y = f(x) = \underset{c}{\operatorname{argmax}}\, W_c^T x$

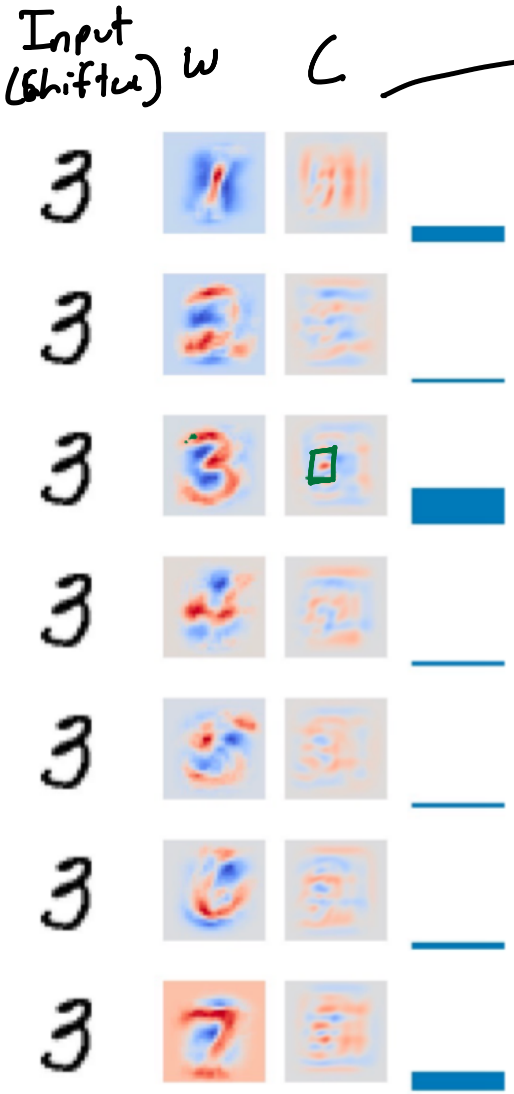
Convolutional Networks (CNNs)

1) Maintain Image Structure
   (Don't flatten)

2) Shift weights to find best alignment

3) Make network sparse
   (Remove weights)

Shifting Weights



$X^T W$ At <u>every</u> location

i.e. C_{ij} → Shift W to be centered at i, j and compute $X^T W$

Predict 3 again!

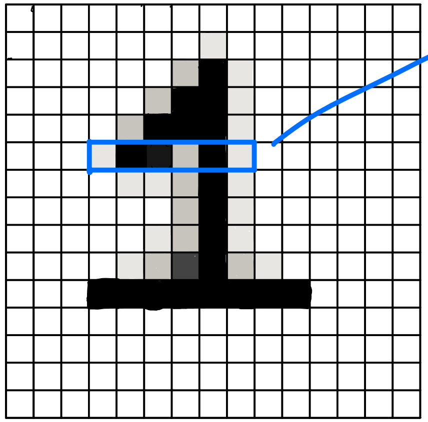
→ Predict with max(C)

→ Location where X and W most closely match

Convolution!

Convolution in 1-dimension

1-Row



Input

0.7 0.5 0.3 1.0 0.0 0.7

× × × × × ×

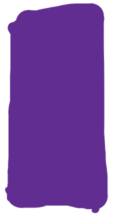
0.2 0.2 0.8 0.0 0.8 0.2

$\oplus \rightarrow$

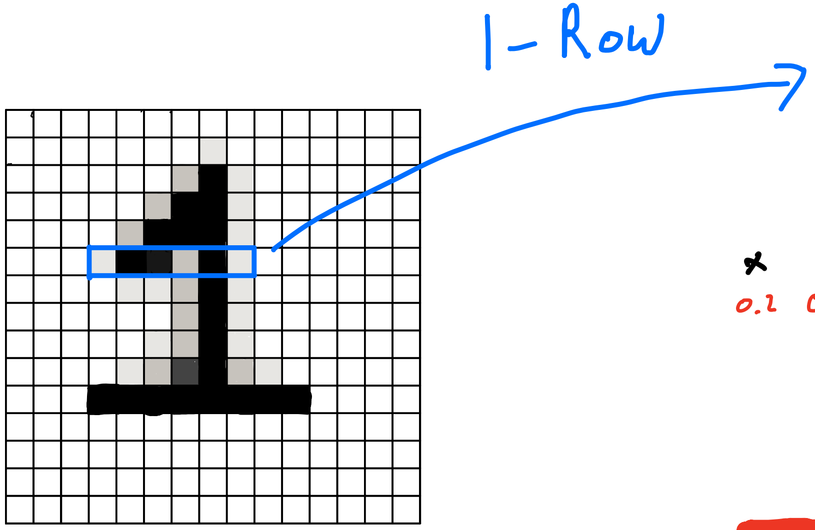
Weights

$$(0.7)(0.2)+(0.5)(0.2)+(0.3)(0.8)+(1.0)(0.0)+(0.0)(0.8)+(0.7)(0.2)$$

$$= 0.77 \{$$



Convolution in 1-dimension



Input

0.7 0.5 0.3 1.0 0.0 0.7

× × × × × × × ×

0.2 0.2 0.8 0.0 0.8 0.2 — —

⊕ →

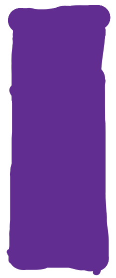
Assume 0

Weights

Try different alignment

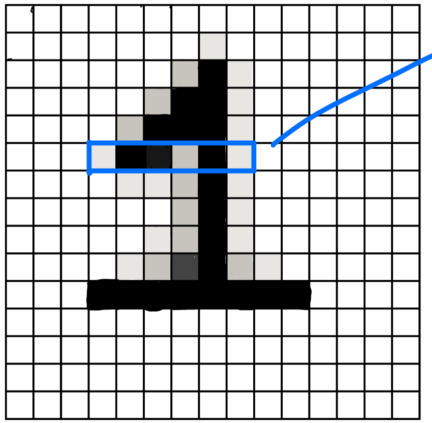
$$(0.7)(0.8)+(0.5)(0.0)+(0.3)(0.8)+(1.0)(0.2)$$

$$= 1.0$$



Better than before!

Convolution in 1-dimension



1-Row

Input

0.7 0.5 0.3 1.0 0.0 0.7

× × × × × × × ×

0.2 0.2 0.8 0.0 0.8 0.2 — —

⊕ →

└ Assume 0

Weights

$$(0.7)(0.8)+(0.5)(0.0)+(0.3)(0.8)+(1.0)(0.2)$$

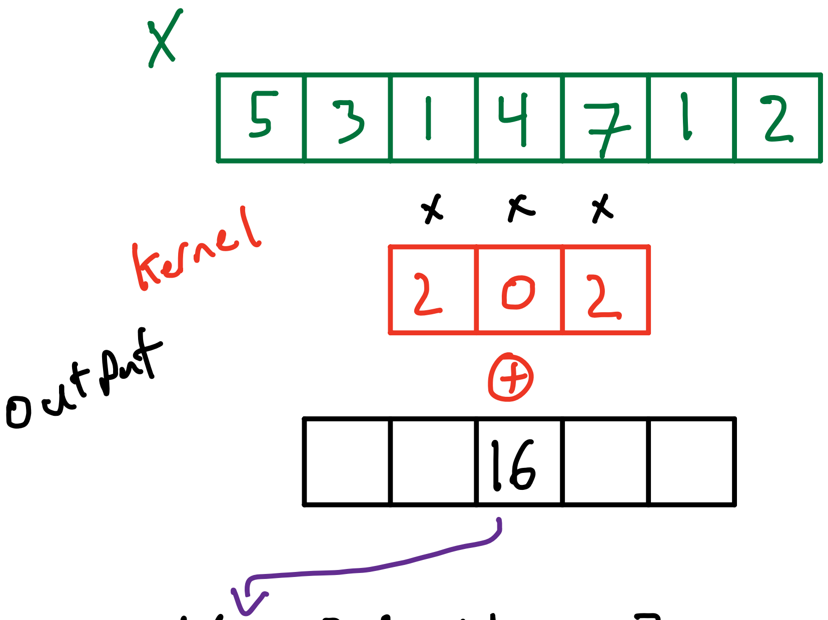
$$= 1.0$$

Try different alignment

Better than before!

Convolution operator (1-D)

Inputs: x : Array of length d
Kernel (weights) : Array of length s



$16 = 2\cdot 1 + 4\cdot 0 + 7\cdot 2$

Typically: $s < d$

Only compute alignments where kernel fully overlaps x

In torch: Padding='Valid'

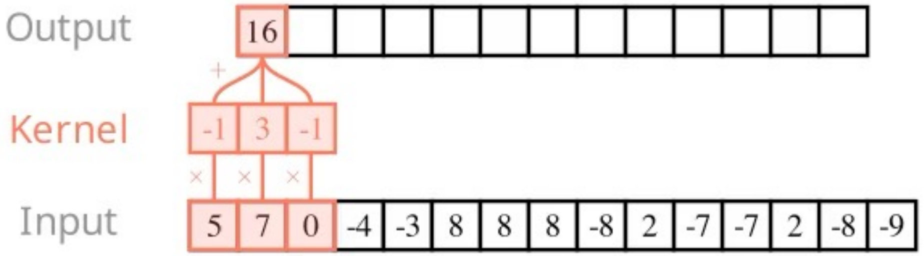
$$\mathrm{Conv}(x,k)_i = \sum_{j=1}^{s} x_{i+j} k_j$$

Ouput length $= d-(s-1)$

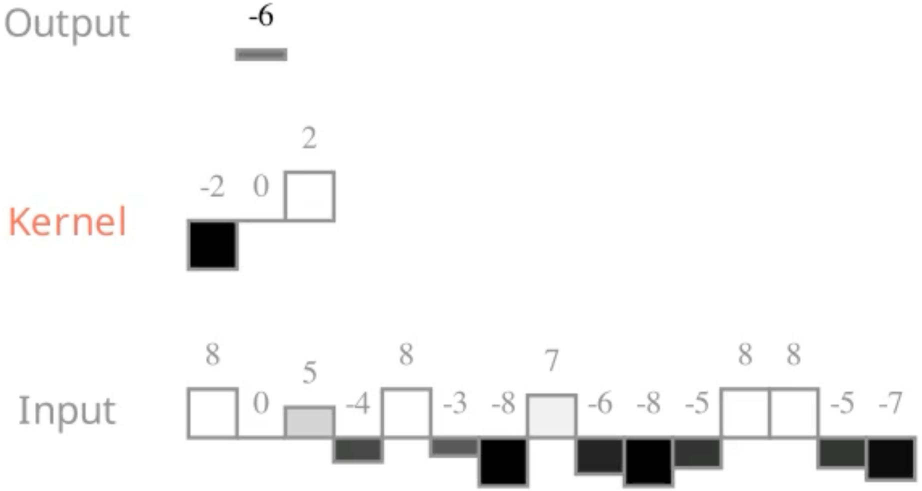
Convolution Animated!

Convolution (kernel size: 3)

Output[0]=(5)(-1)+(7)(3)+(0)(-1)= 16



Convolution (kernel size: 3)



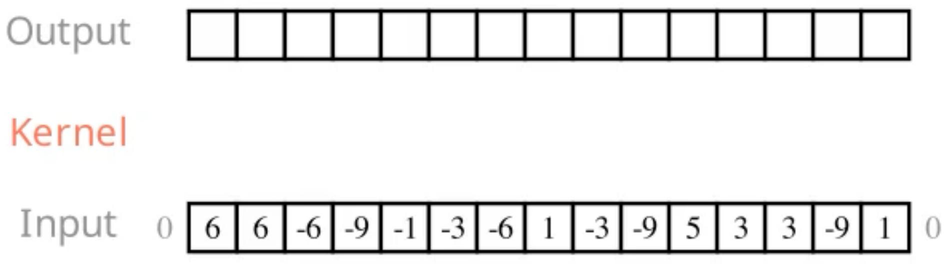
Padding

- If we want to try every possible alignment we need to pad the input w/ 0s
- In torch: padding='full'

$$\text{Output length} = d + 2(S-1)$$

Padding

Convolution (size: 3, padding: 1)

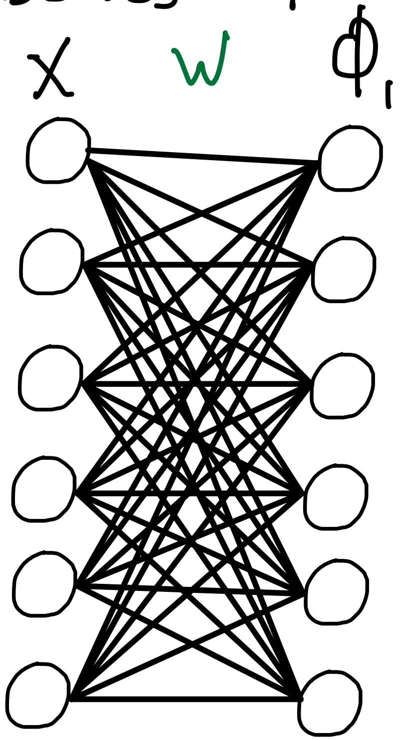


- Often want Padding in-between
- e.g. If we want an output size of d
- In torch: Padding='Same'

Output length = d

Convolution as a Layer

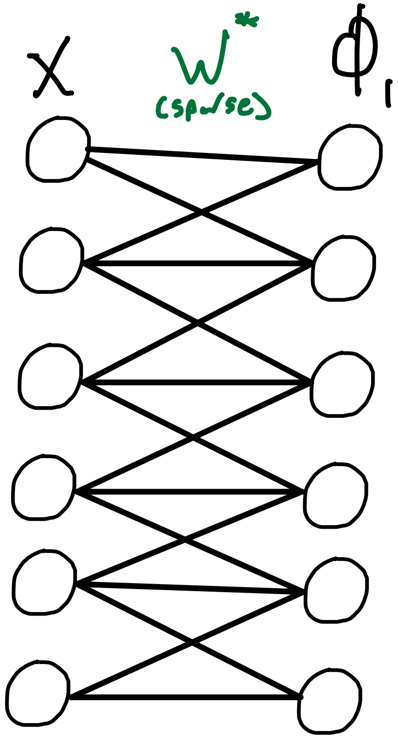
Standard ('Dense') Layer



$$\phi_1 = \sigma(X^T W + b)$$

Every output depends on <u>every</u> input

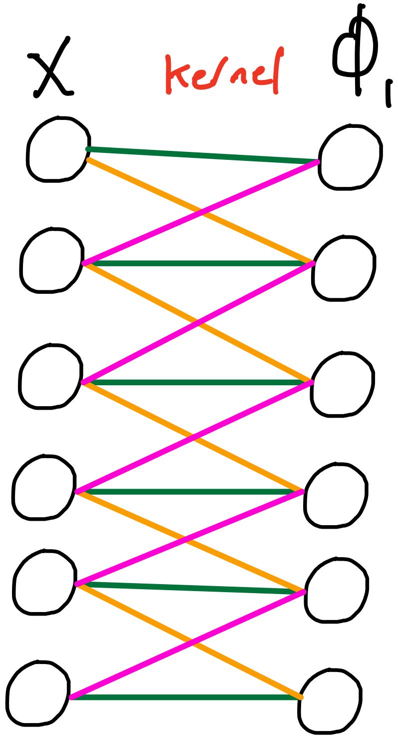
Locally-connected Layer



$$\phi_1 = \sigma(X^T W^* + b)$$

Every output depends only on <u>Local</u> inputs

Convolutional Layer



$$\phi_1 = \sigma(\text{conv}(X, k) + b)$$

<u>And</u> weights are <u>Shared</u> for each output!

Derivatives of convolutions

Ignoring activations!

In general:

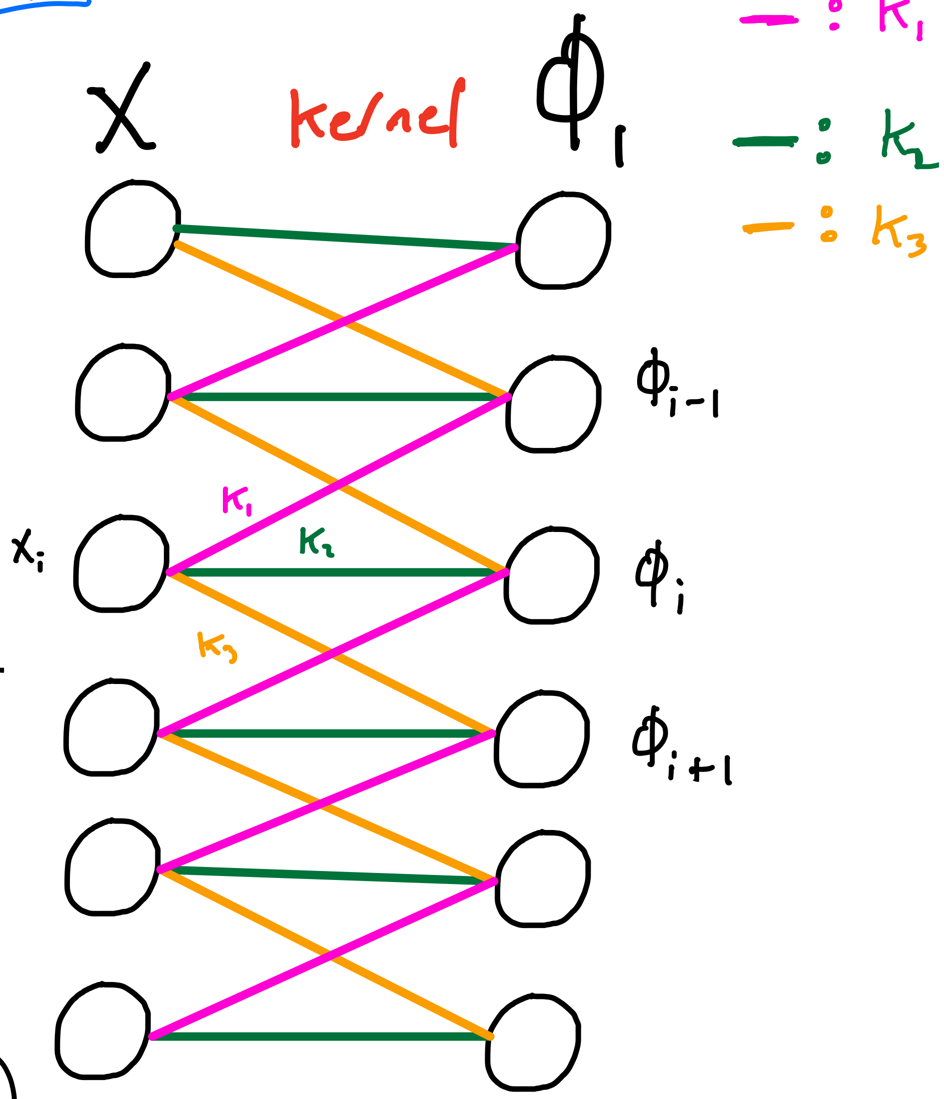
$$\frac{dL}{dx_i} = \sum_{j=1}^{d} \frac{dL}{d\phi_j} \cdot \frac{d\phi_j}{dx_i}$$

but only $\phi_{i-1}, \phi_i, \phi_{i+1}$ depend on x_i so:

$$\frac{dL}{dx_i} = \frac{dL}{d\phi_{i-1}} \underbrace{\frac{d\phi_{i-1}}{dx_i}}_{= k_1} + \frac{dL}{d\phi_i} \underbrace{\frac{d\phi_i}{dx_i}}_{= k_2} + \frac{dL}{d\phi_{i+1}} \underbrace{\frac{d\phi_{i+1}}{dx_i}}_{= k_3}$$

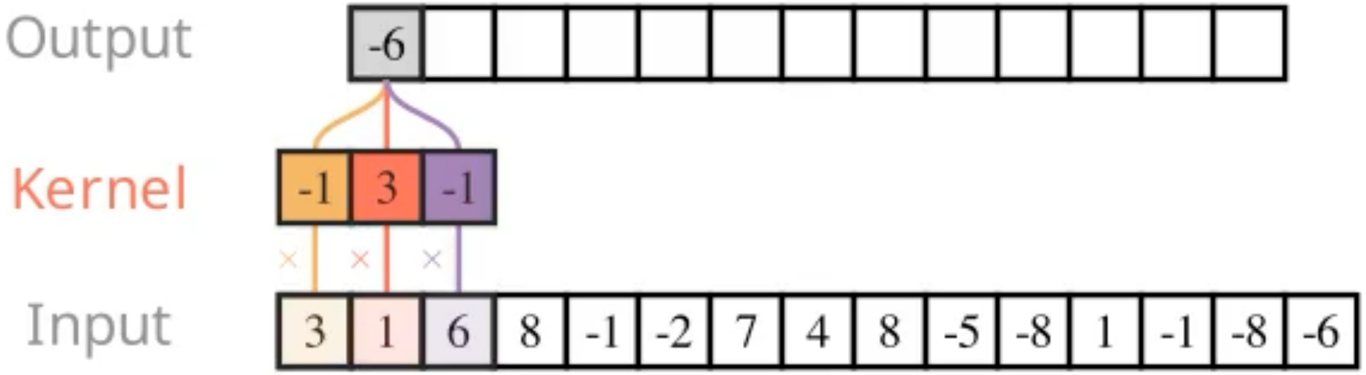
$$\phi = \mathrm{conv}(x, [k_1, k_2, k_3])$$

$$\frac{dL}{dx} = \mathrm{conv}\left(\frac{dL}{d\phi}, [k_3, k_2, k_1]\right)$$

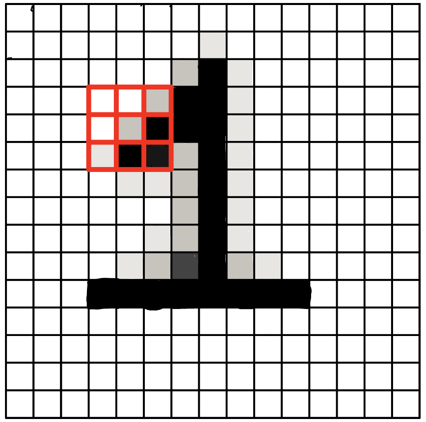


Derivatives Animated!

Convolution (kernel size: 3)



Convolutions in 2-D



Align kernel in every 2-d location

$$\mathrm{Conv2d}(x,k)_{ij} = \sum_{a=1}^{s}\sum_{b=1}^{s} x_{i+a,j+b}\cdot k_{a,b}$$

[For padding = 'Valid']

5	-1	3	8	4	6	-2
-3	4	9	1	1	7	-4
5	-6	3	2	1	-2	0
0	-8	5	5	4	-2	3
4	7	2	-7	3	1	4
4	-3	1	2	6	1	-1
0	0	3	-2	3	-1	0

-1	-2	-1
0	0	0
1	2	1

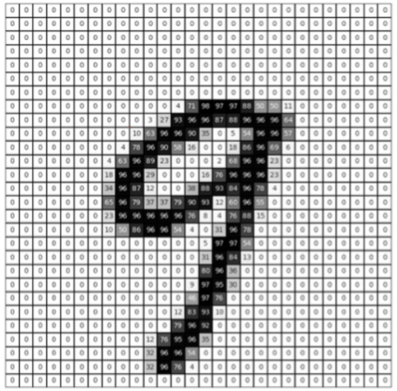
			1	

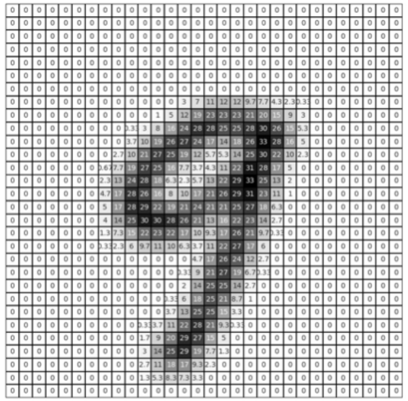
$$1 = 1\cdot(-1) + 1\cdot(-2) + 7(-1) + 2\cdot 0 + 1\cdot 0 + (-2)\cdot 0 + 5\cdot 1 + 4\cdot 2 + (-2)\cdot 1$$

Convolutions in 2-D (Blur)

Small Blur

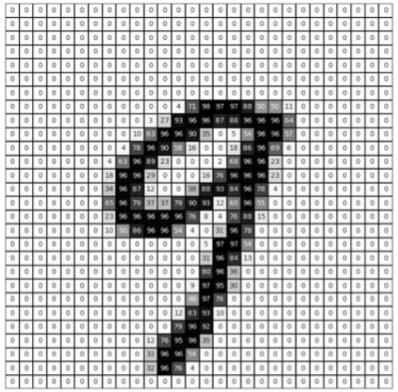
0.12	0.12	0.12
0.12	0.12	0.12
0.12	0.12	0.12

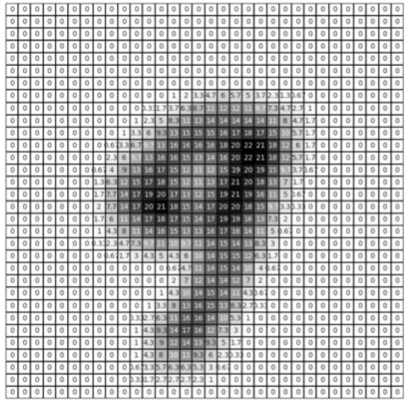




Large Blur

0.04	0.04	0.04	0.04	0.04
0.04	0.04	0.04	0.04	0.04
0.04	0.04	0.04	0.04	0.04
0.04	0.04	0.04	0.04	0.04
0.04	0.04	0.04	0.04	0.04





Convolutions in 2-D (Edge detect)

Vertical edges

pos.

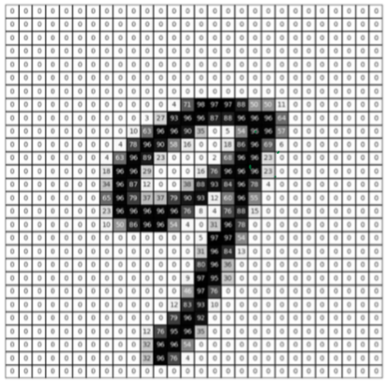
$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

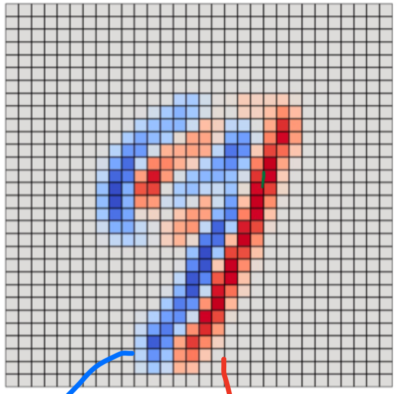
neg.

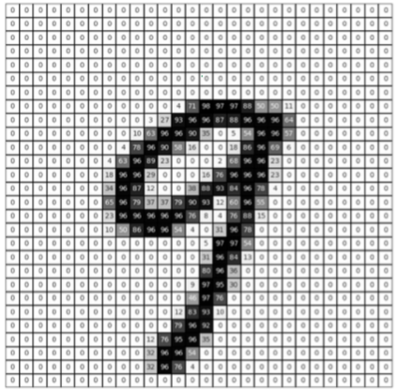
most outputs near 0

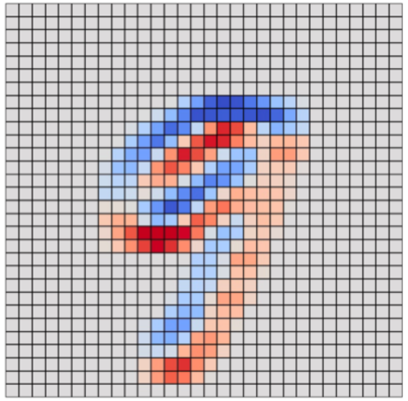
Horizontal edges

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$







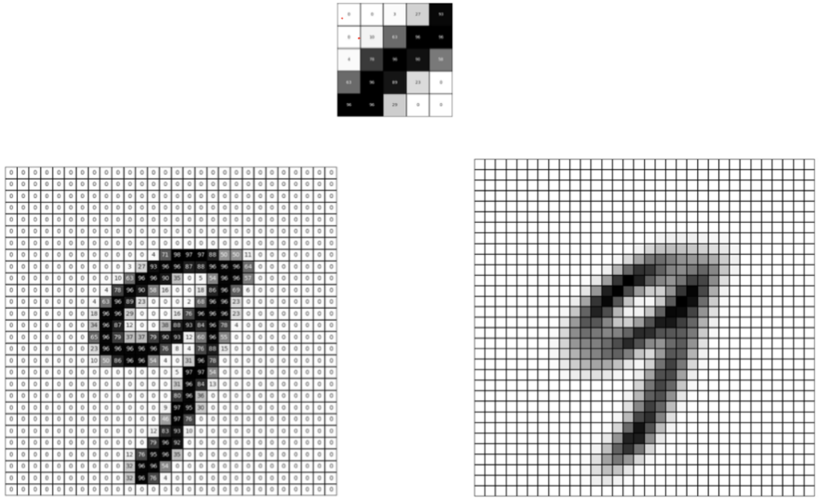


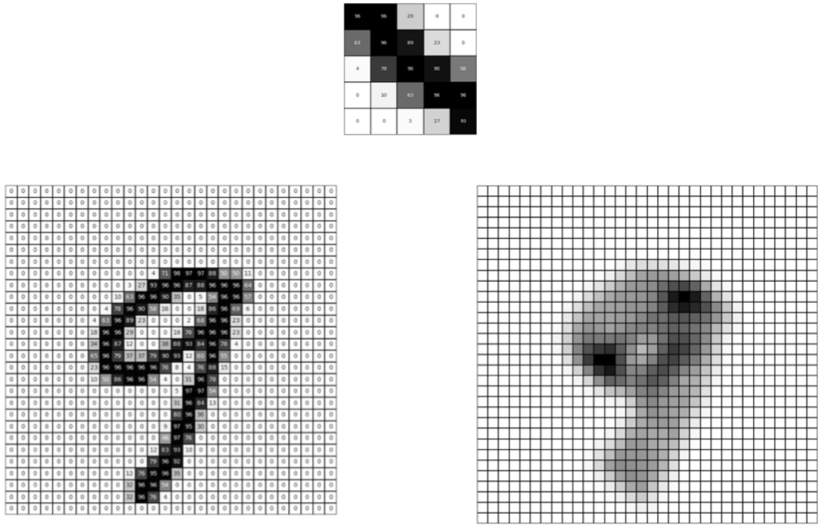
Low Values at neg. of kernel

High Values where pattern matches kernel

Convolutions in 2-D (Edge detect)

Diagonal edges

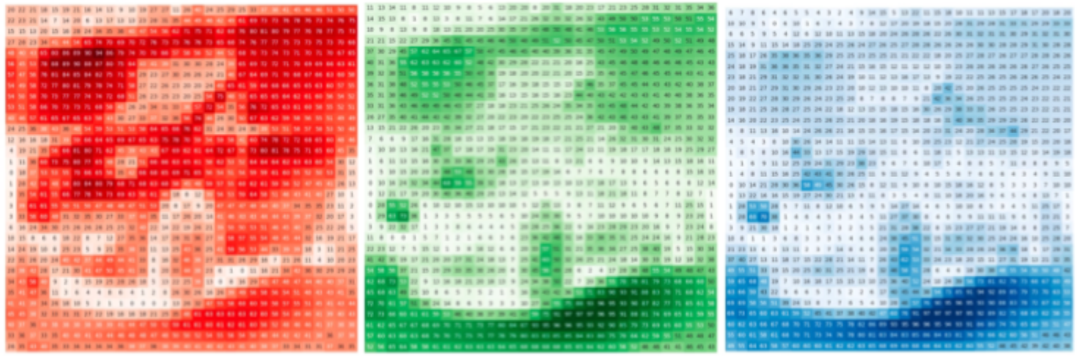




Convolutions for color images

Image (X)

Kernel (K)



Kernel has 3 channels like Image!

$$\underbrace{\mathrm{Conv}(\overset{H \times W \times 3}{X}, \overset{5 \times 5 \times 3}{K})}_{} =$$

$$\mathrm{Conv}(X_r, K_r) + \mathrm{Conv}(X_g, K_g) + \mathrm{Conv}(X_b, K_b)$$

Red

73	47	17
96	37	0
75	66	9

green

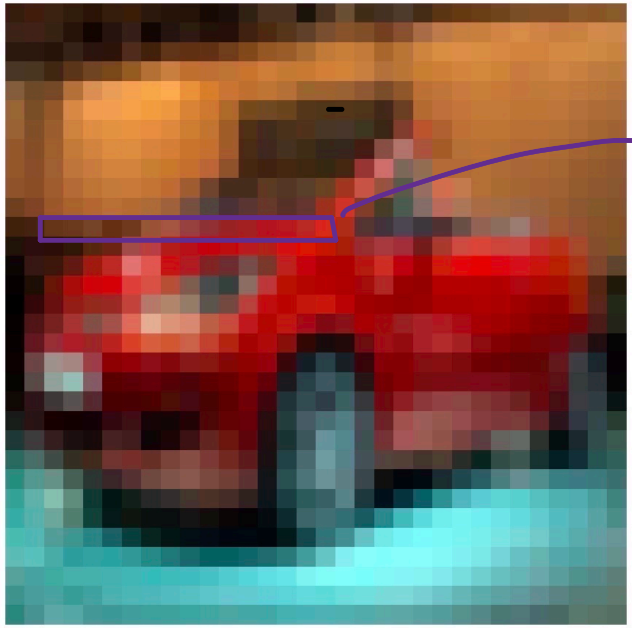
49	53	57
46	57	49
49	51	52

Blue

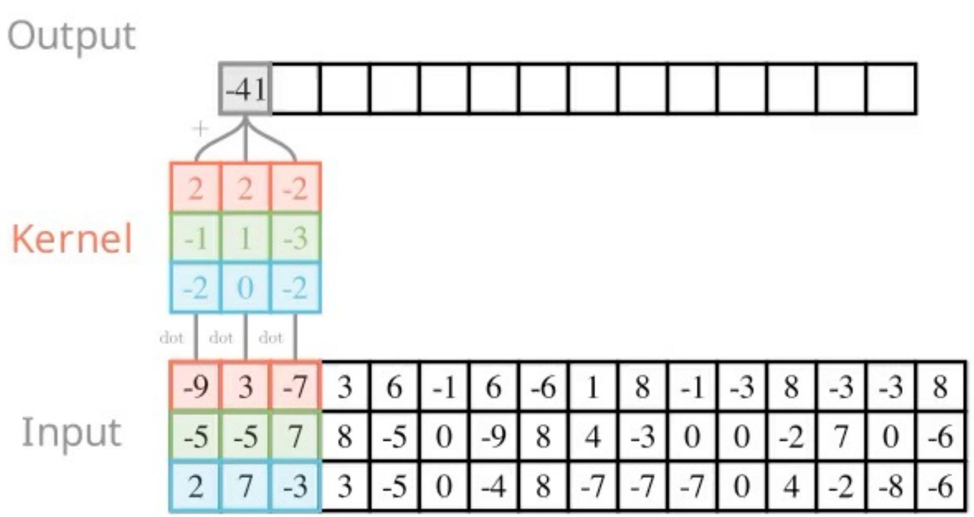
32	56	80
10	57	99
33	34	92

result

Multi-Channel Convolution (1-D)



Convolution (size: 3, channels: 3, output channels: 1)



Not limited to 3-channels!

Sum over channels!

$$\underbrace{\mathrm{Conv}(X, K)}_{} = \sum_{c=1}^{C} \mathrm{Conv}(X_c, K_c)$$

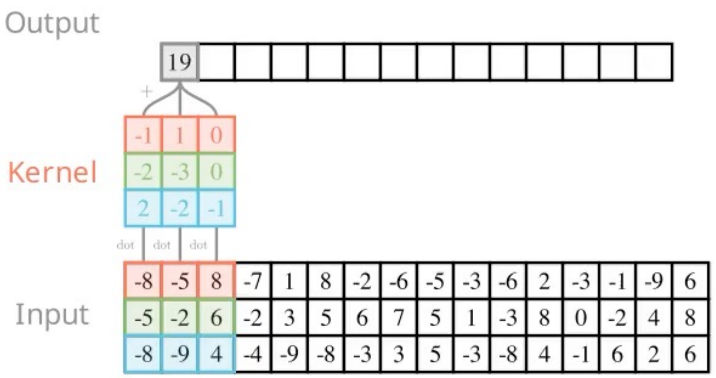
X: $H \times W \times C$ $\quad K$: $S \times S \times C$ $\quad X_c$: $H \times W$ $\quad K_c$: $S \times S$

Multiple Output Channels

Kernel can also produce multiple channels (>1 value at each location)

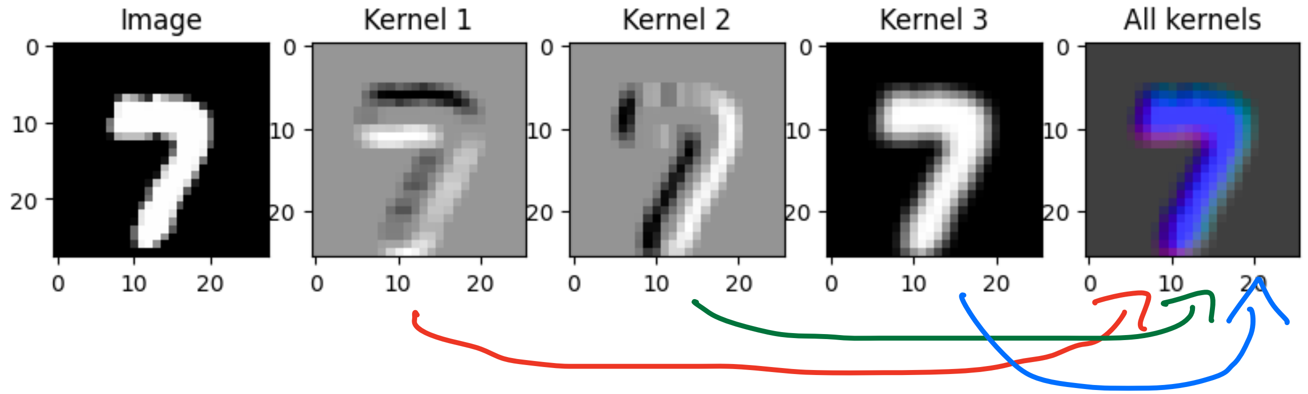
1-D

Convolution (size: 3, channels: 3, output channels: 3)



In 2-D

Combine 3 filters into a multi-channel Image



General 2-D Convolutional Layer

Input ($\underline{X}$): $N \times H \times W \times C$

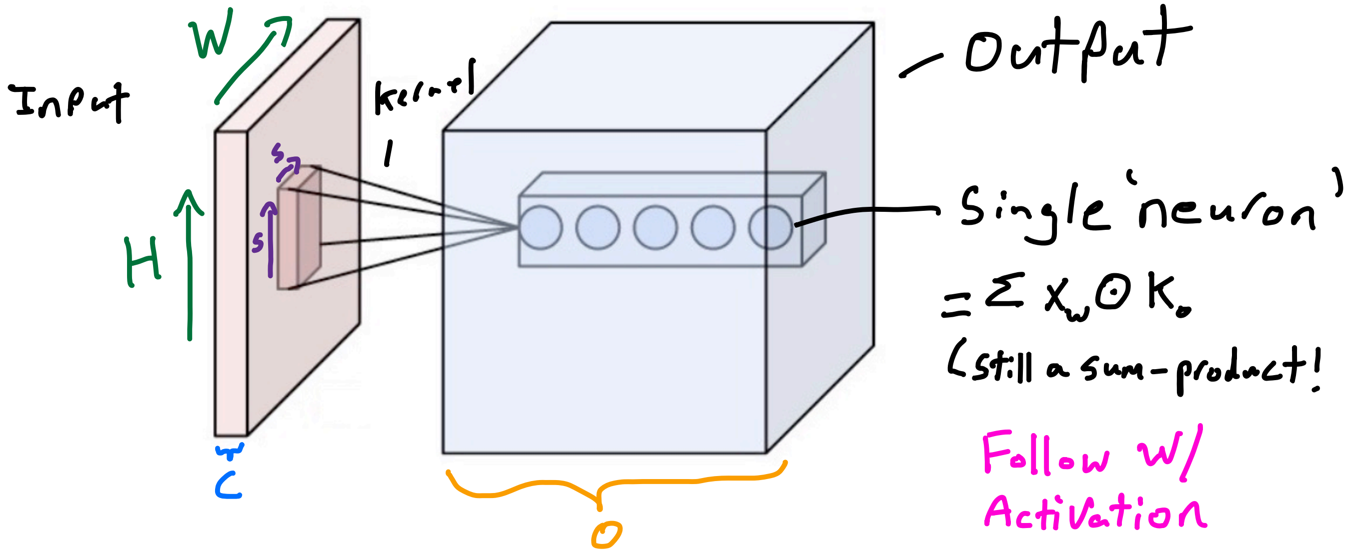
N: # Images (Batch) · H: Height · W: Width · C: # Channels

Kernel ($\underline{K}$): $S \times S \times C \times O$

$S \times S$: Kernel size · C: # channels · O: # output channels

Output (ϕ): $N \times \underbrace{H \times W}_{\text{May change for padding} \neq \text{'same'}} \times O$

Diagram:



PyTorch 2-D Convolutional Layer

Input (X): $N \times C \times H \times W$

\# Images (Batch) | # Channels | Height | Width

Channels before Image size

Kernel (K): $C \times O \times S \times S$

\#channels | # output channels | kernel size

Output (ϕ): $N \times O \times H \times S$

CONV2D

CLASS torch.nn.Conv2d(*in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None*) [SOURCE]

Applies a 2D convolution over an input signal composed of several input planes.

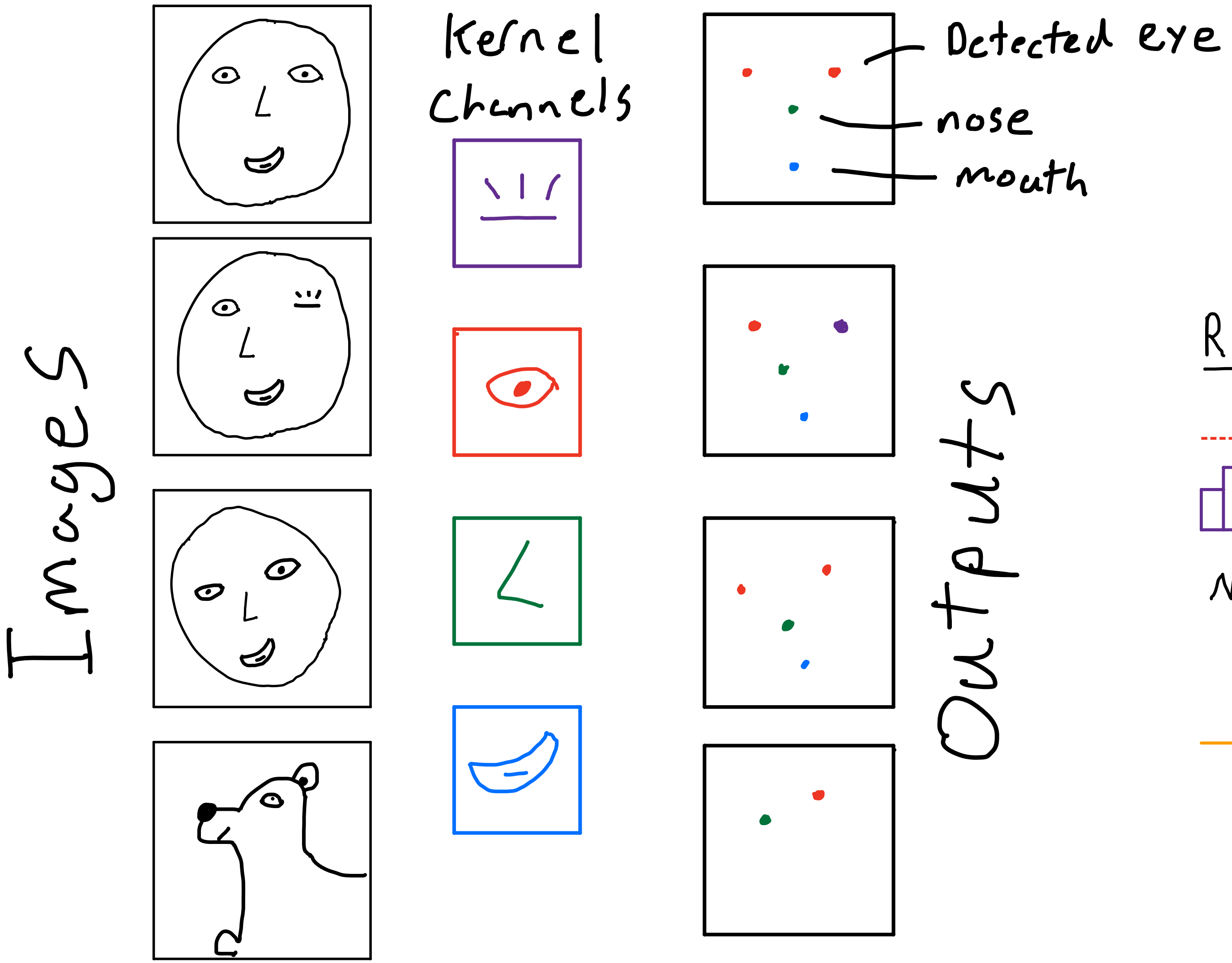
In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

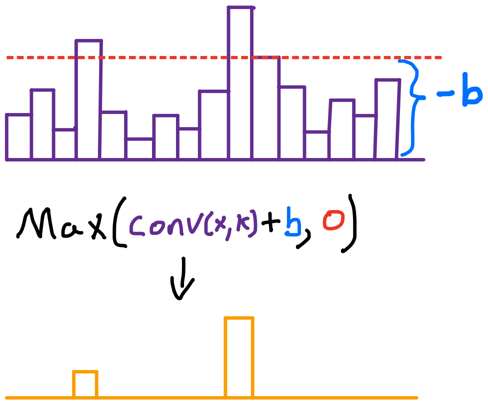
where $\star$ is the valid 2D cross-correlation operator, N is a batch size, C denotes a number of channels, H is a height of input planes in pixels, and W is width in pixels.

Technically what we call Convolution is cross correlation

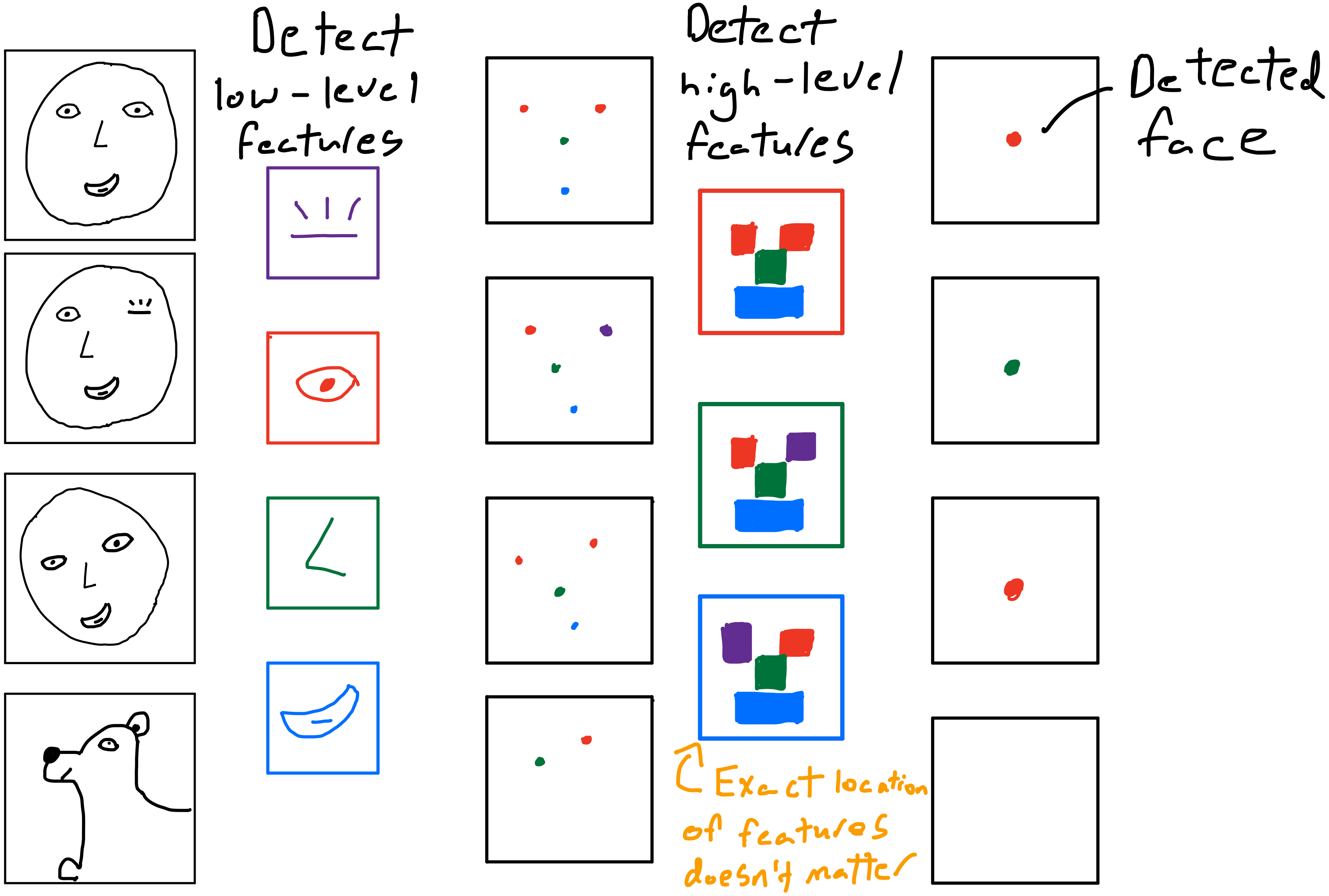
Convolutions as feature detectors



ReLU as threshold

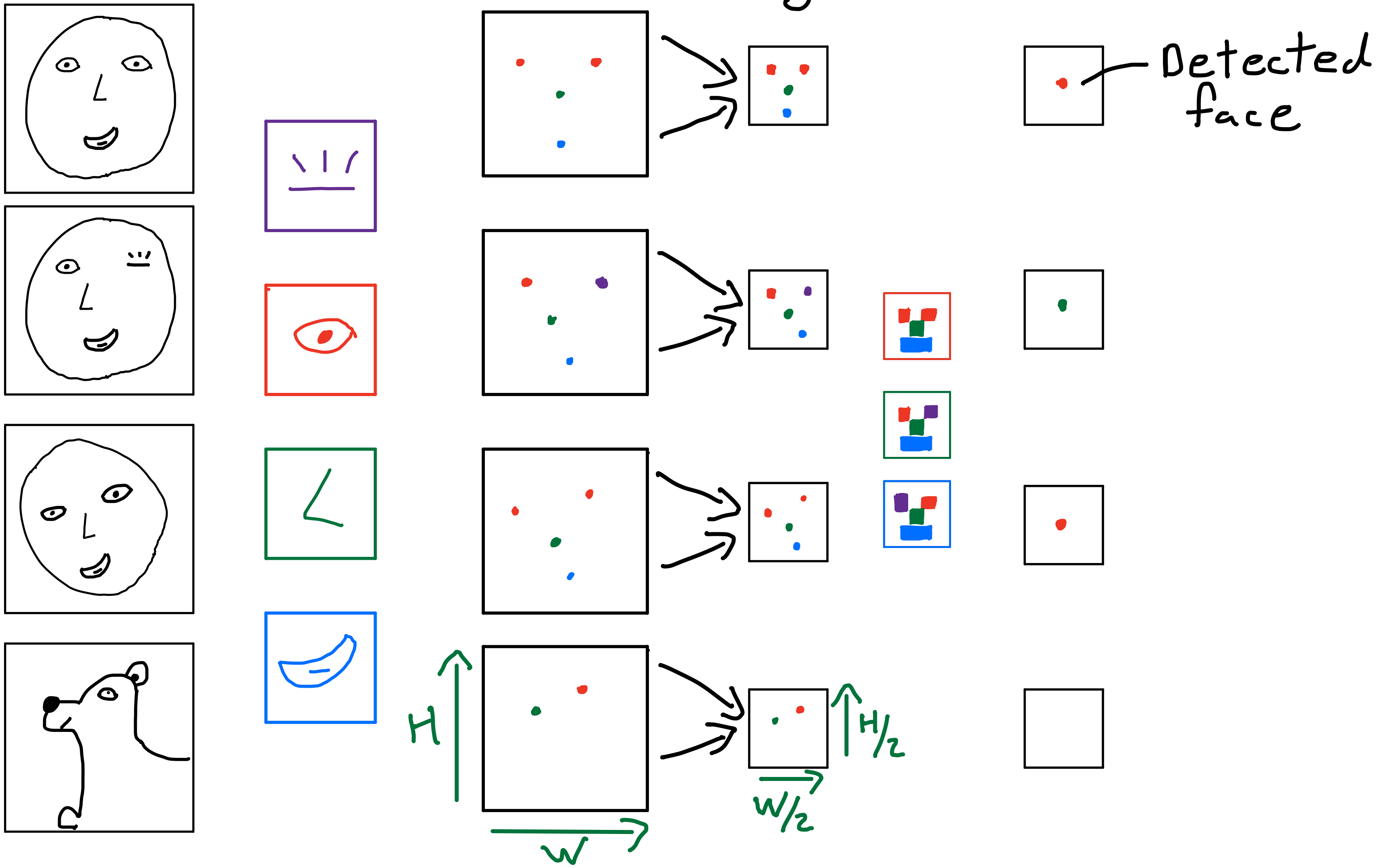


Multi-Layer CNNs



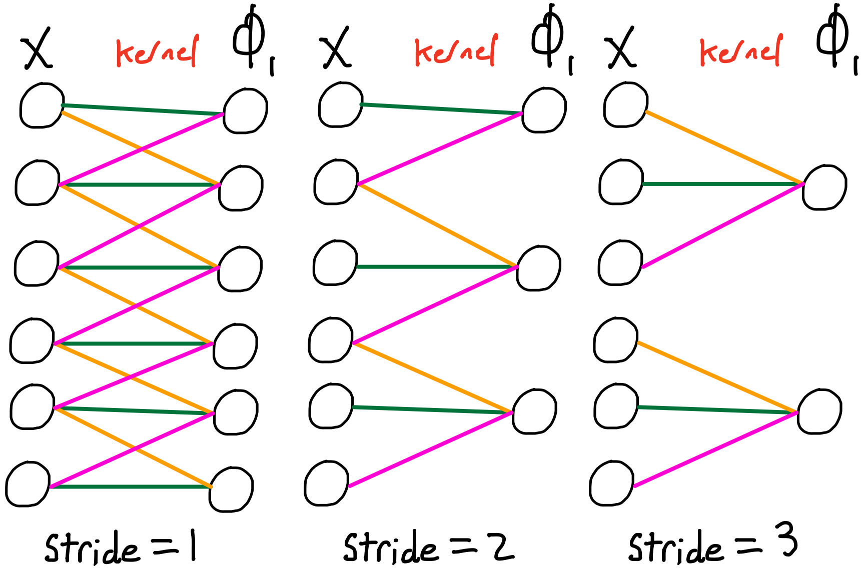
Down Sampling

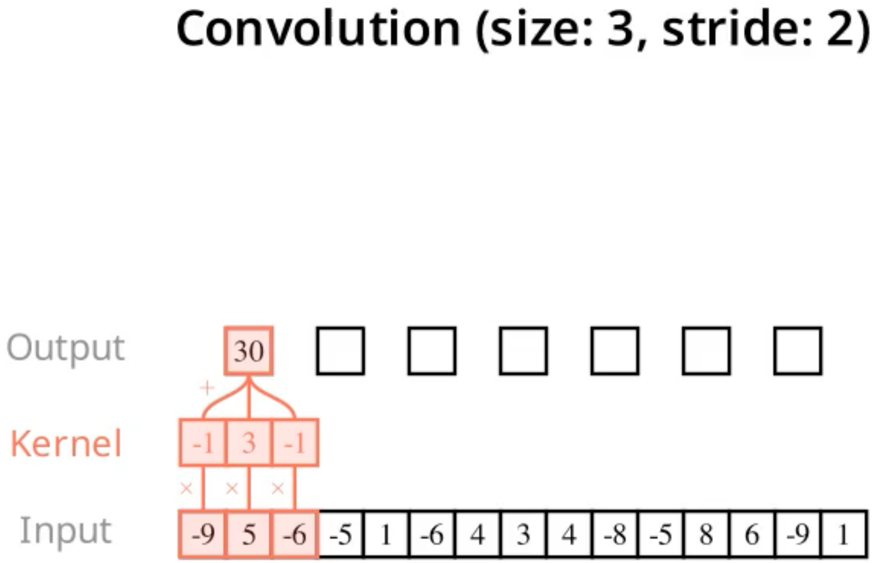
Reduce resolution → Easier and less expensive to find high-level features

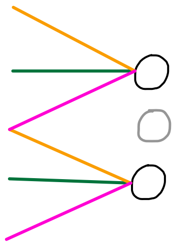


Strided Convolution (1-D)

- Take only $\frac{1}{\text{stride}}$ × Outputs



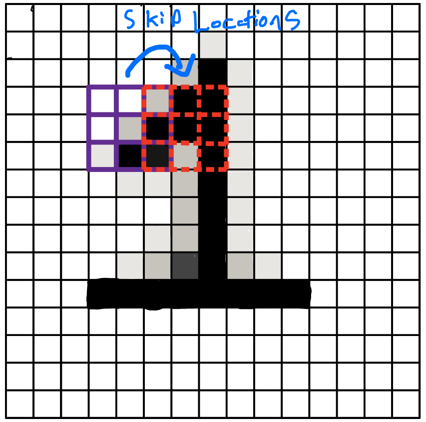


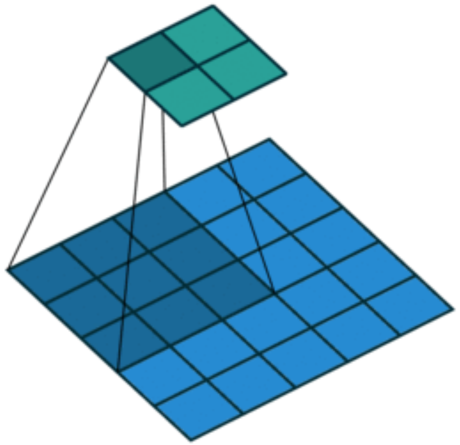


What if the optimal activation is here?

Strided Convolutions (2-D)

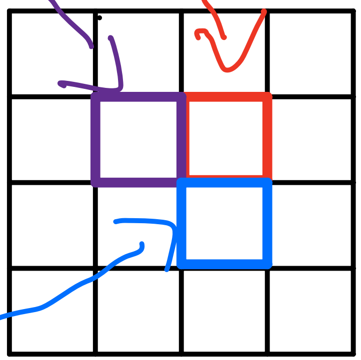
- Apply stride in each dimension





5	-1	3	8	4	6	-2
-3	4	9	1	1	7	-4
5	-6	3	2	1	-2	0
0	-8	5	5	4	-2	3
4	7	2	-7	3	1	4
4	-3	1	2	6	1	-1
0	0	3	-2	3	-1	0

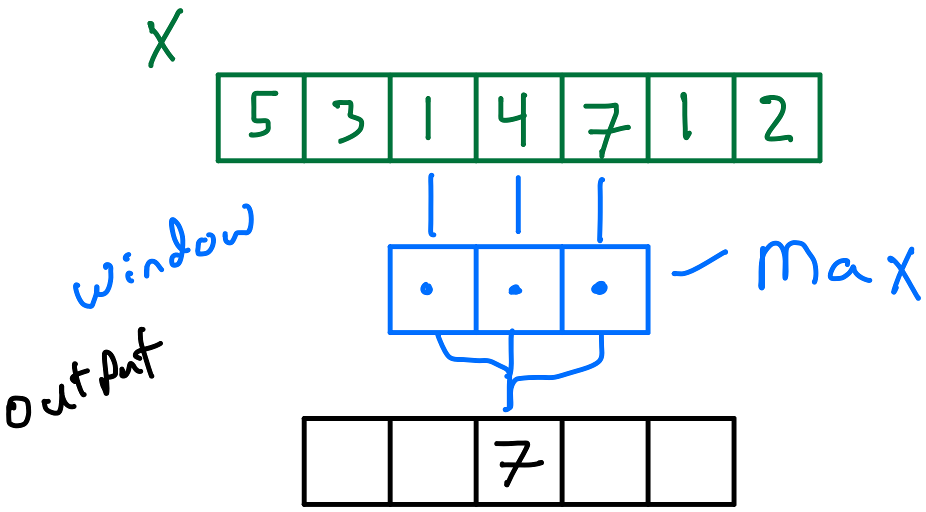
-1	-2	-1
0	0	0
1	2	1



Stride = 2

Pooling operator (1-D)

Inputs: x : Array of length $\underline{d}$
window size $\underline{s}$

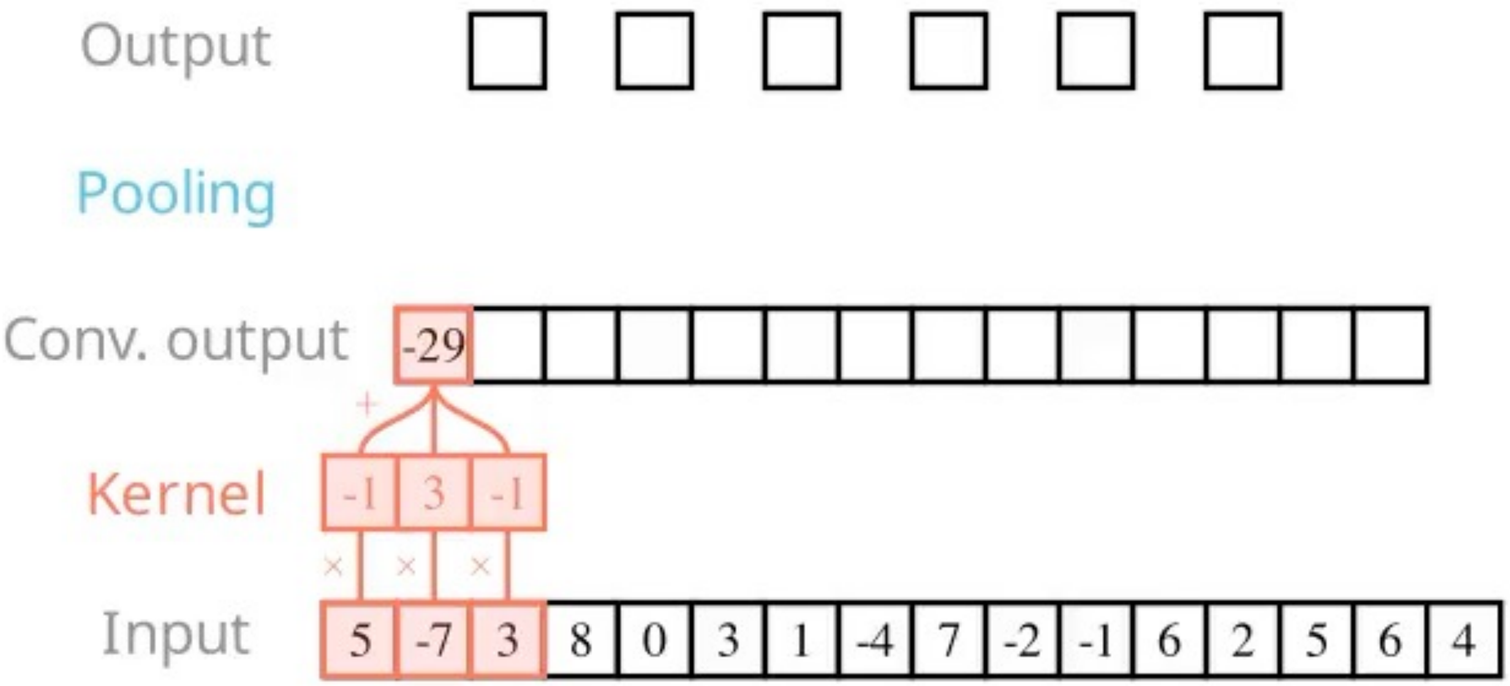


$7 = \max(1, 4, 7)$

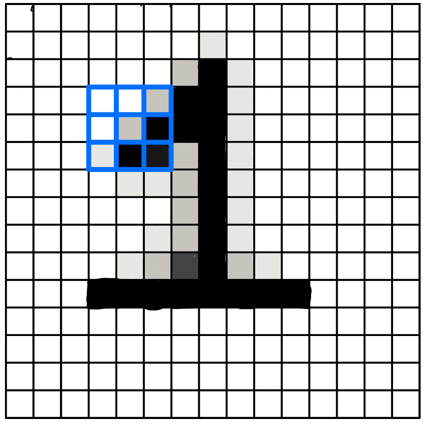
- Take max output aroud each location <u>before</u> downsampling
- Make sure we don't miss any high activations
- Can also take Avg. (Average pooling)

Convolution + MaxPooling Animated!

Convolution (size: 3) + Max Pooling (size: 3, stride: 2)



Pooling in 2-D



Align window in every 2-d location

$$\text{Max-Pool}(X)_{ij} = \max_{a=1}^{s}\left(\max_{b=1}^{s}\left(X_{i+a,j+b}\right)\right)$$

$$\text{Avg.-Pool}(X)_{ij} = \frac{1}{s^2}\sum_{a=1}^{s}\sum_{b=1}^{s} X_{i+a,j+b}$$

5	-1	3	8	4	6	-2
-3	4	9	1	1	7	-4
5	-6	3	2	1	-2	0
0	-8	5	5	4	-2	3
4	7	2	-7	3	1	4
4	-3	1	2	6	1	-1
0	0	3	-2	3	-1	0

max

			7	

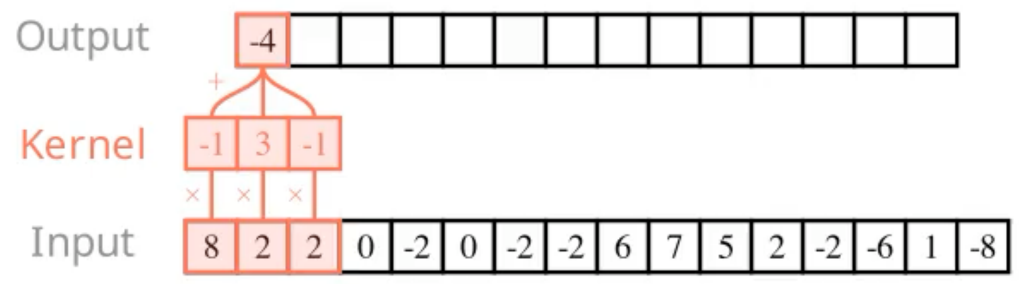
$$7 = \max(1, 1, \underline{7}, 2, 1, -2, 5, 4, -2)$$

Down Sampling Comparison

Pixel shuffle

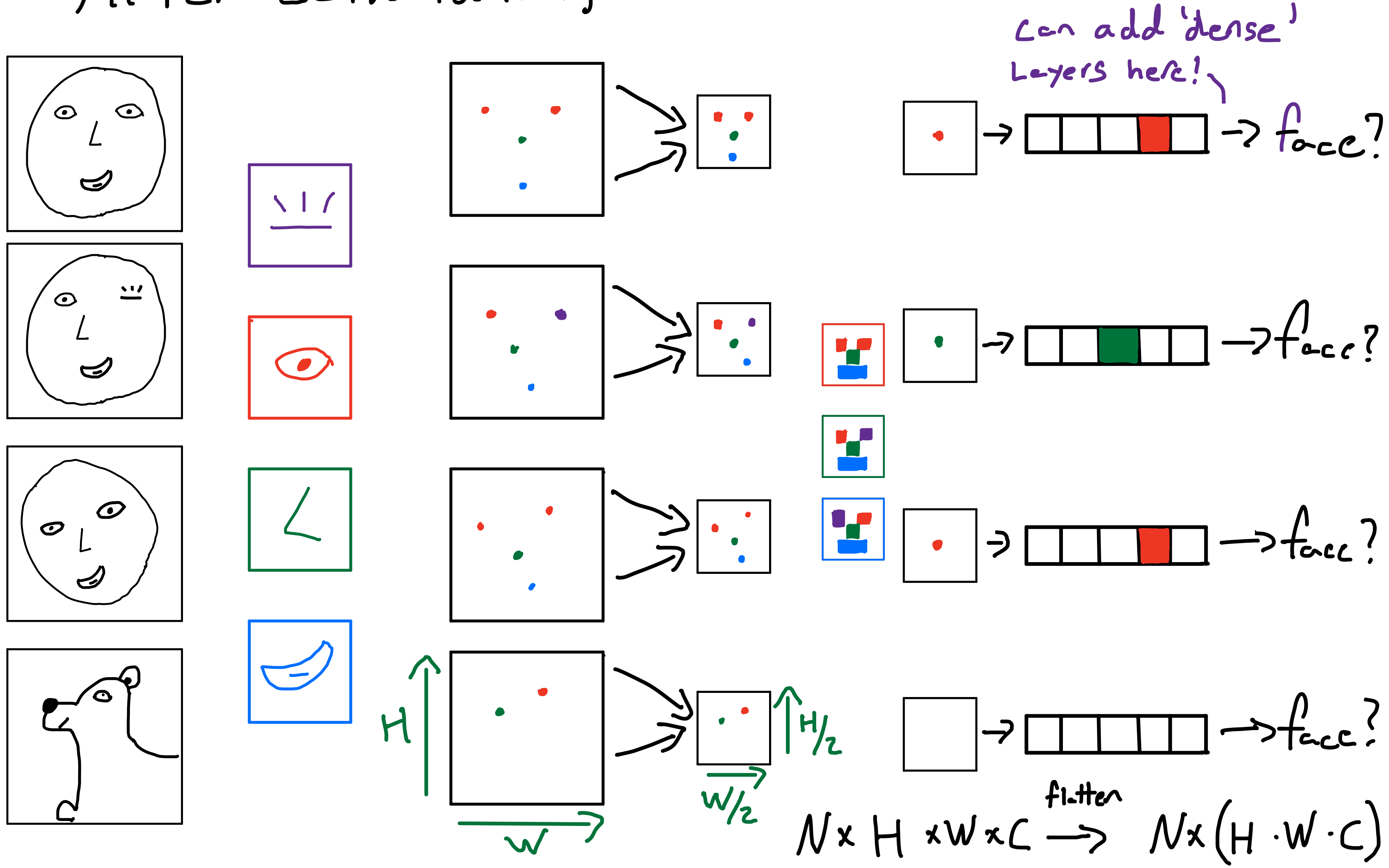
- Rearrange adjacent results into more channels – Expensive!

Convolution (size: 3) + Shuffle



Flattening

- After convolutions, flatten as before



Global Avg. Pooling

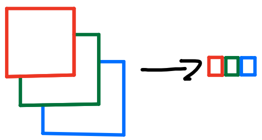
- Alt. to flattening, just Avg. over remaining Image

$$\text{Global Avg.-Pool}(x) = \frac{1}{W \cdot H} \sum_{a=1}^{W} \sum_{b=1}^{H} X_{a,b}$$

5	-1	3	8	4	6	-2
-3	4	9	1	1	7	-4
5	-6	3	2	1	-2	0
0	-8	5	5	4	-2	3
4	7	2	-7	3	1	4
4	-3	1	2	6	1	-1
0	0	3	-2	3	-1	0

$\rightarrow \boxed{3}$

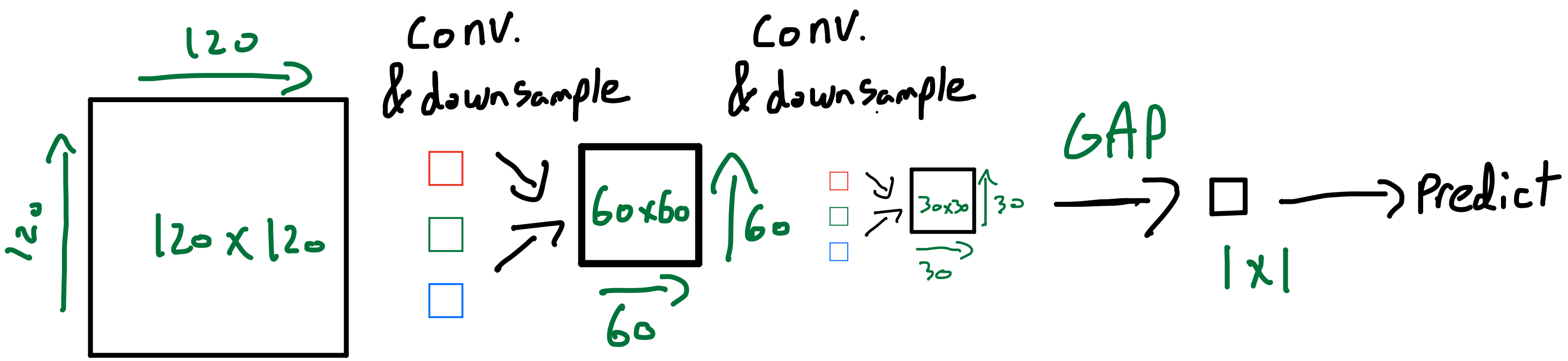
Remember: still multiple channels!

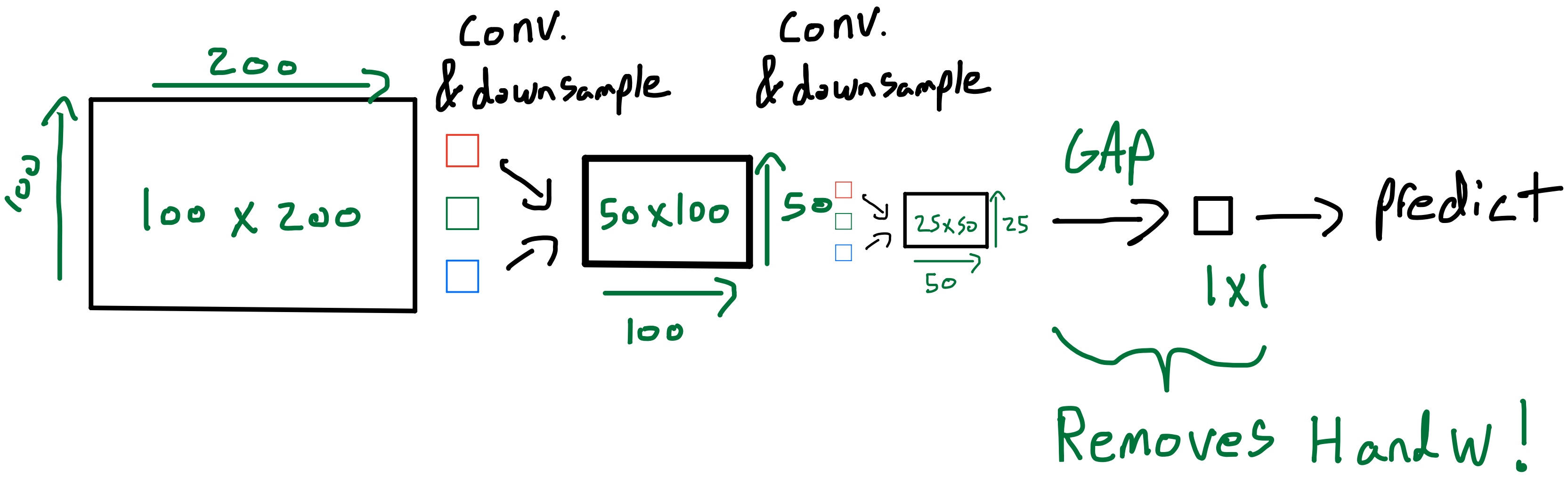


$$N \times H \times W \times C \xrightarrow{GAP} N \times C$$

Global Avg. Pooling

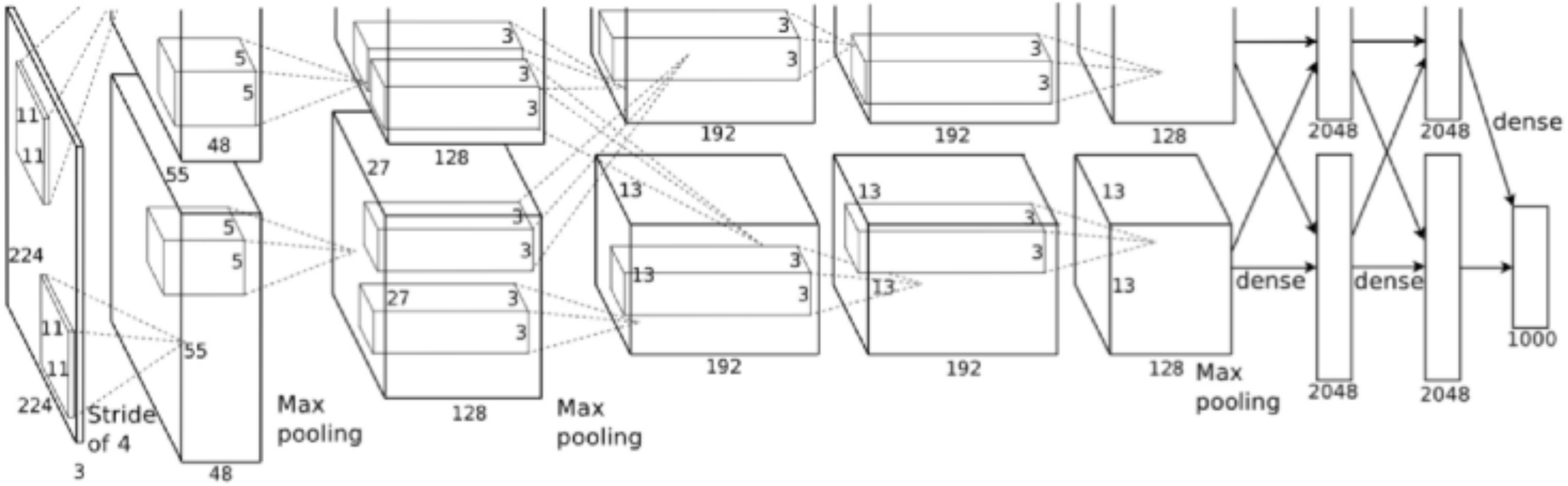
- Allows for inputs of different sizes!



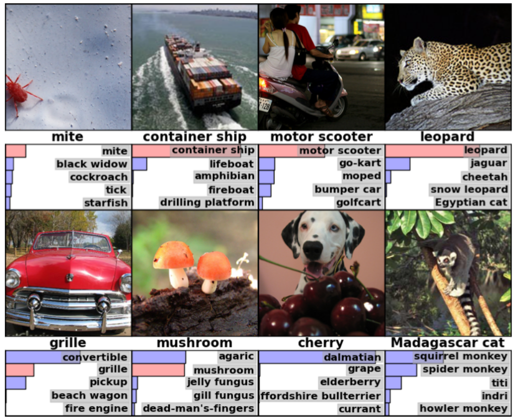


AlexNet (2012)

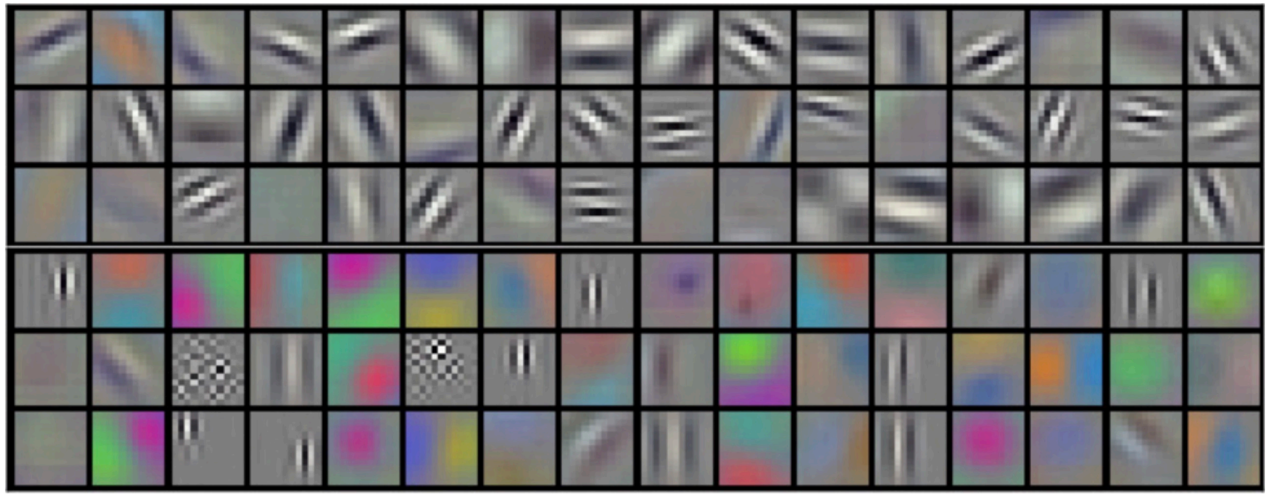
Architecture:



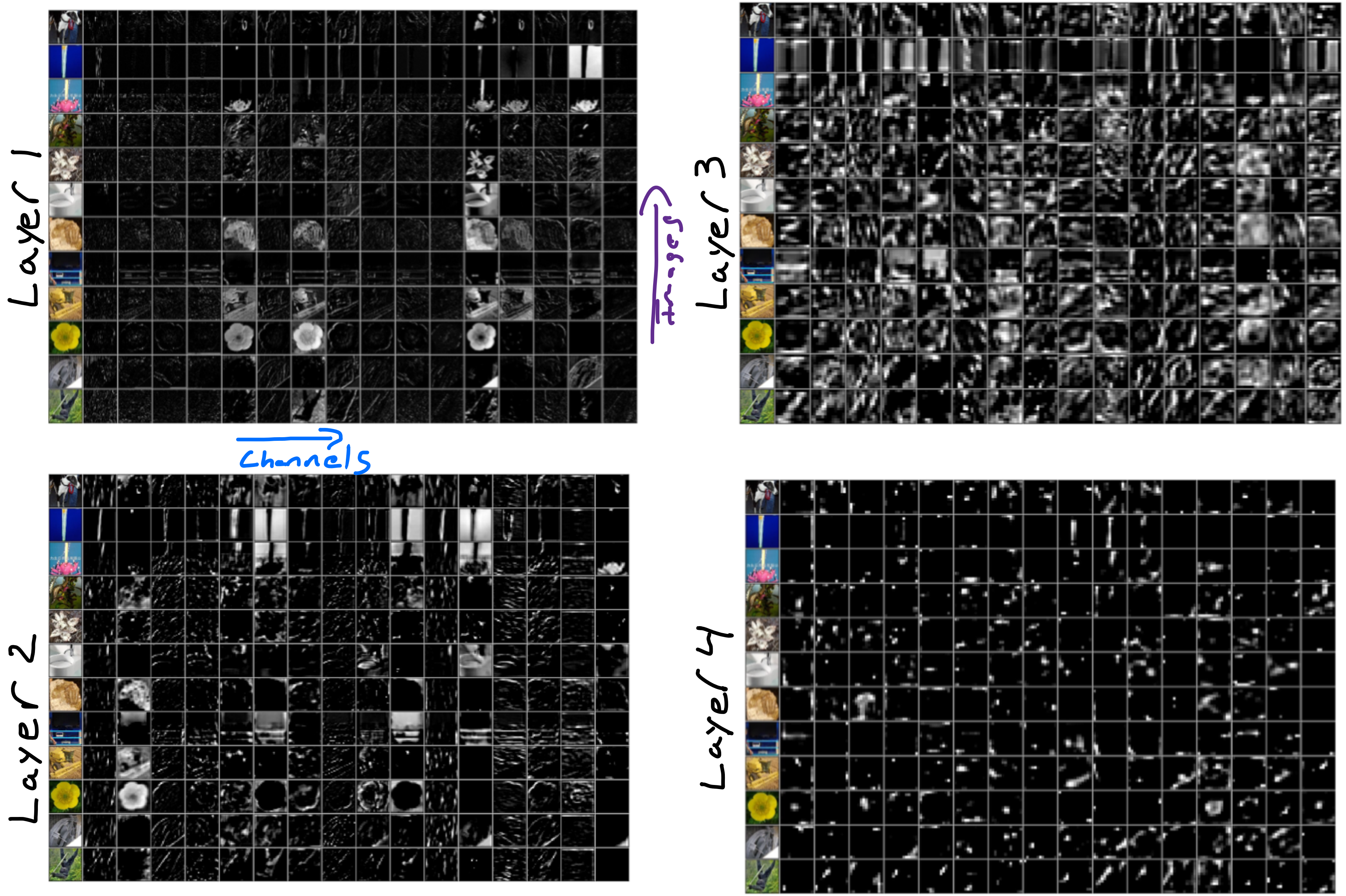
1000 Classes



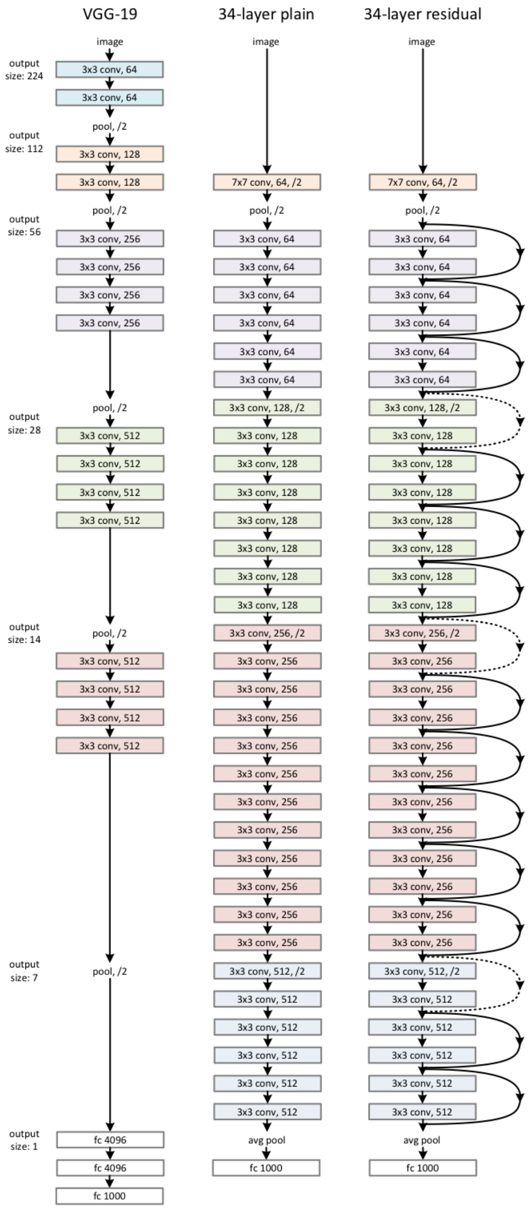
Learned kernels



CNN Features by Layer

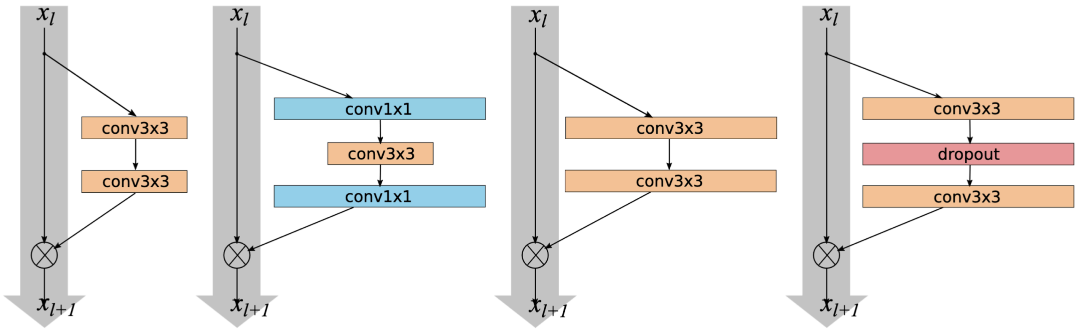


Other Architectures



← VGG, Resnet-50, etc.

Wide resnet ↓



(a) basic (b) bottleneck (c) basic-wide (d) wide-dropout

Figure 1: Various residual blocks used in the paper. Batch normalization and ReLU precede each convolution (omitted for clarity)

Many, Many more!

Data Augmentation

- It takes a lot of data to train good Image classifiers!
  - ~ millions to Billions of Images for general object recognition (1000+ Classes)

Data Augmentation

- CNNs (mostly) invarient to translation

  - What about scale, rotation, color etc.?

Astronaut

Still Astronaut!

- Still shouldn't change class!

Data Augmentation

- Augment existing data by randomly scaling, rotating, shifting colors etc.
- Much easier than collecting ~10x more data

- Can do this as we train!

Common Augmentations

Rotation

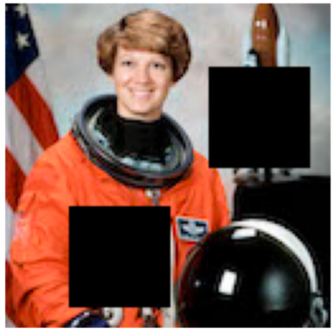
Crop

Color shift

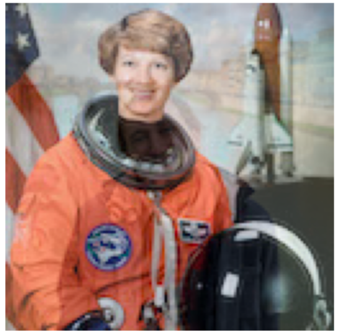
Shift / scale / shear

Flip

Cut-out



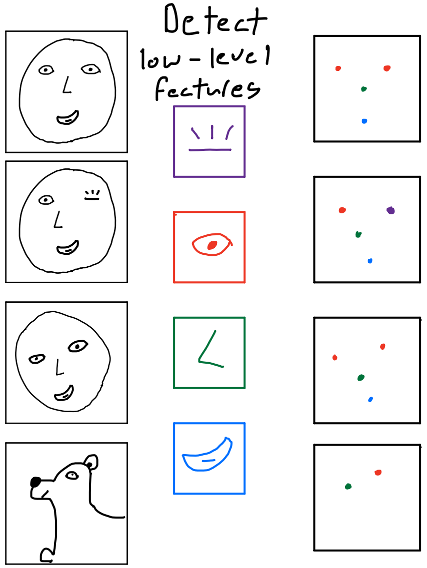
Mix-up



Fine Tuning

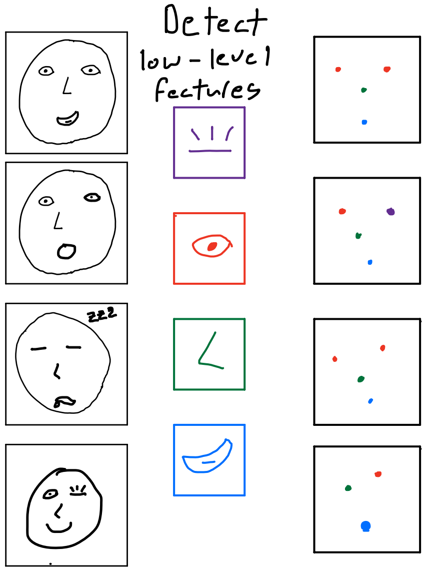
- Low-level features can often be shared between Models
- Exploit this by copying Convolutional layers from a previously trained model

Old model



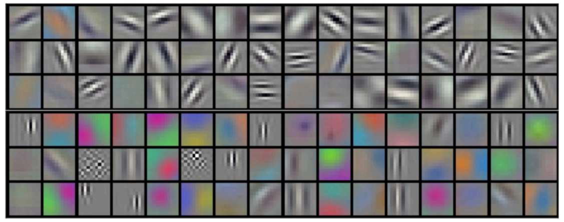
... → Face? ...

New model



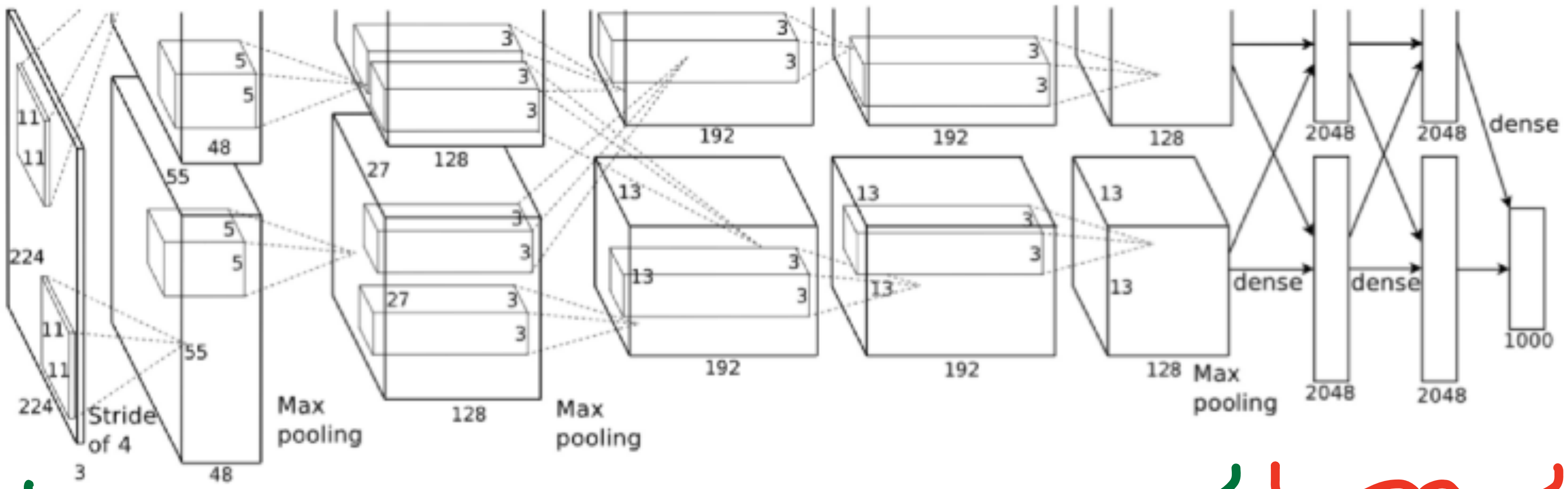
... → Smiling? ...

Actually more like this! ↓



Fine Tuning

For new task:



Keep

Convolutional Layers

Replace

Dense Layers

Typical Approach, but could keep more or less

Fine Tuning

- Better starting point for optimization