Convolutional Neural Networks

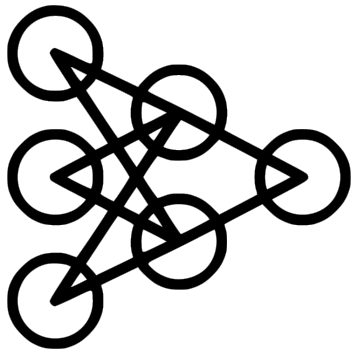
Lecture 12

Last time: A Guide to DNN configurations

- ✓ **Initialization:** how to initialize the weights so that they do not saturate?
- ✓ **Activation:** how to solve the vanishing/exploding gradient problem?
- ✓ **Normalization:** how to get the model to learn the optimal scale?
- ✓ **Regularization:** is it just using l1 and l2, or is it something more?
- ✓ **Optimization:** when gradient descent was too slow or not good enough?
- ✓ **Learning Rate**: what if convergence is too slow or sub-optimal?

Today: Learning Objectives

1. Look at the applications of Convolutional Neural Networks (CNNs)
2. Study the building blocks of CNNs: convolutional and pooling layers
3. Explore winning CNN architectures: LeNet-5, AlexNet, GoogLeNet, ResNet
4. Train with transfer learning



Computer vs. Trivial Sensory Tasks

Not until recently, computer were unable to reliably perform seemingly trivial tasks:

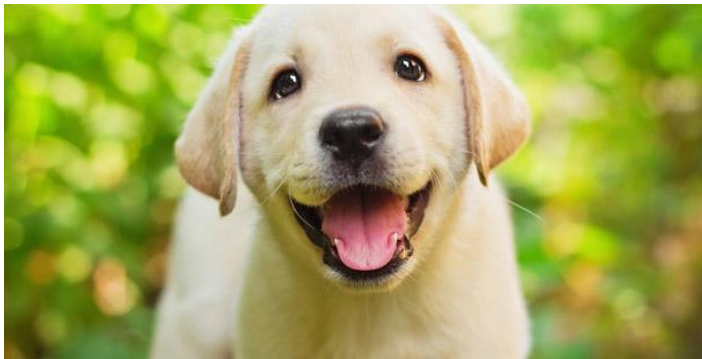
- Detecting a puppy in a picture
- Recognizing spoken words

Humans are really good at this, but cannot explain how they do it:

- Look at a picture of **puppy** ⇒ notice its **cuteness**

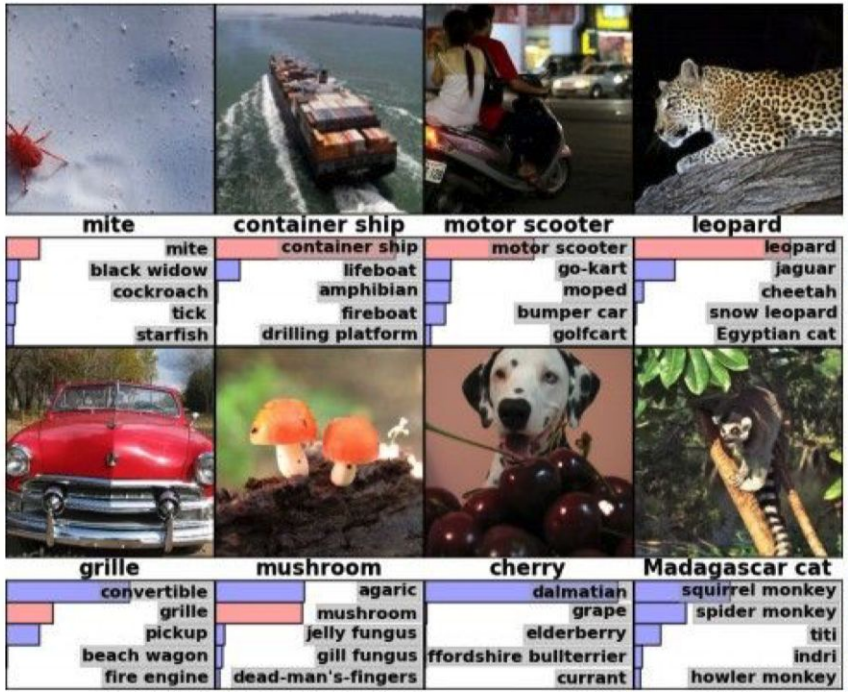
CNN has been used to tackle this challenge

- Increasing computing power
- Large amount of data
- Improved ML algorithms



CNNs are everywhere these days

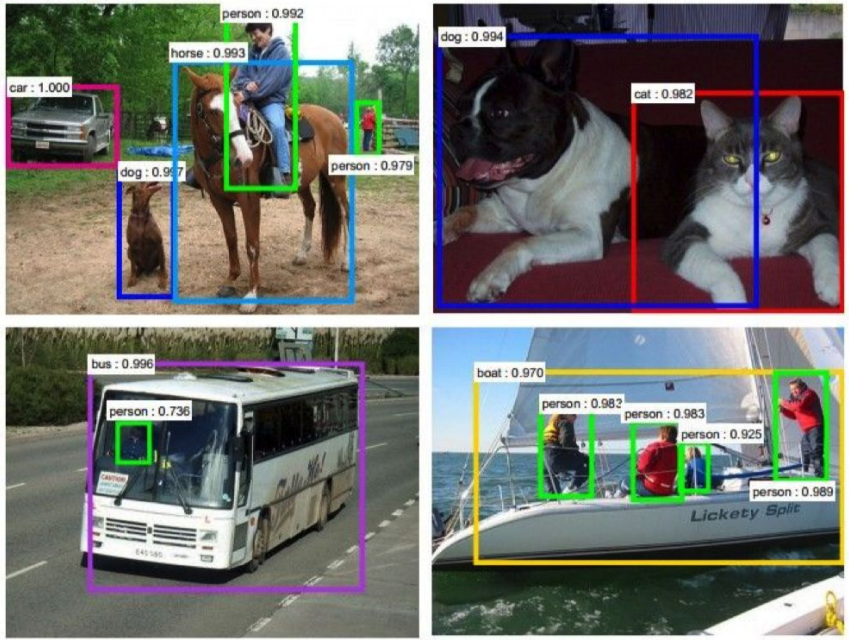
Classification



Retrieval

CNNs are everywhere these days

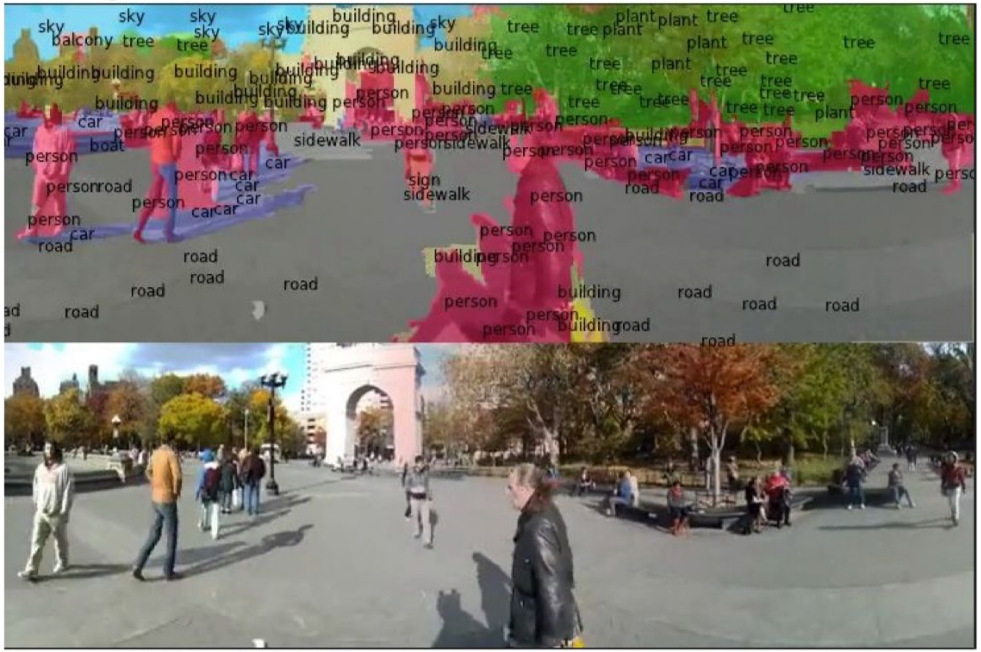
Detection



Figures copyright Shaoqing Ren, Kaiming He, Ross Girschick, Jian Sun, 2015. Reproduced with permission.

[Faster R-CNN: Ren, He, Girshick, Sun 2015]

Segmentation



Figures copyright Clement Farabet, 2012.
Reproduced with permission.

[Farabet et al., 2012]

CNNs are everywhere these days

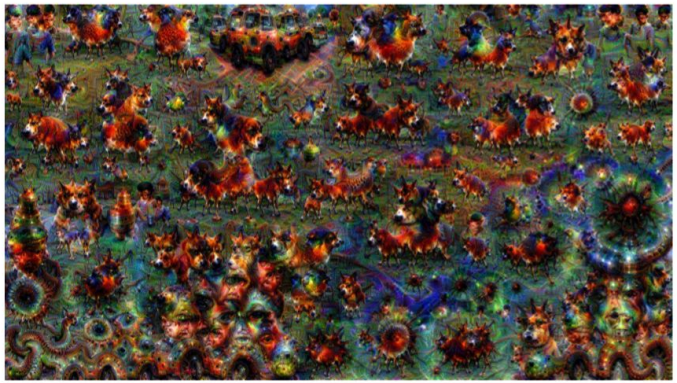
Photo by Lane McIntosh. Copyright CS231n 2017.

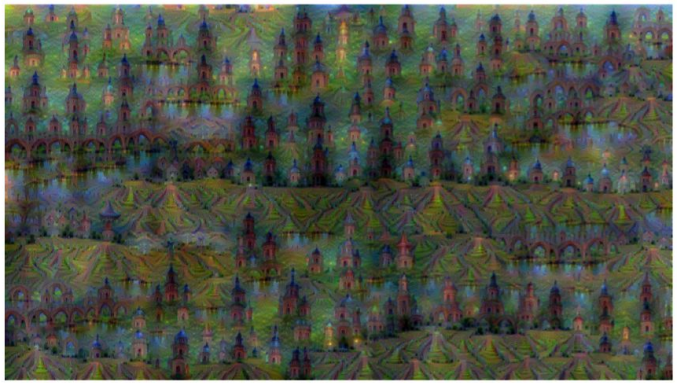
self-driving cars

NVIDIA Tesla line
(these are the GPUs on rye01.stanford.edu)

Note that for embedded systems a typical setup would involve NVIDIA Tegras, with integrated GPU and ARM-based CPU cores.

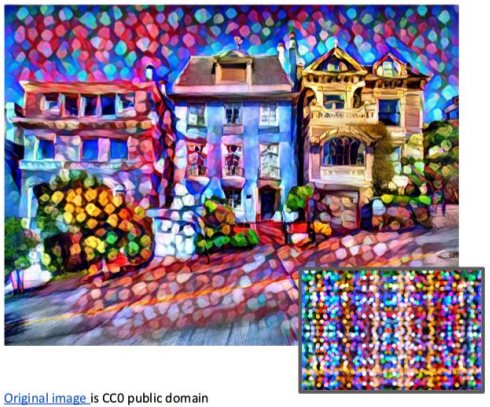
CNNs are everywhere these days





Figures copyright Justin Johnson, 2015. Reproduced with permission. Generated using the Inceptionism approach from a blog post by Google Research.

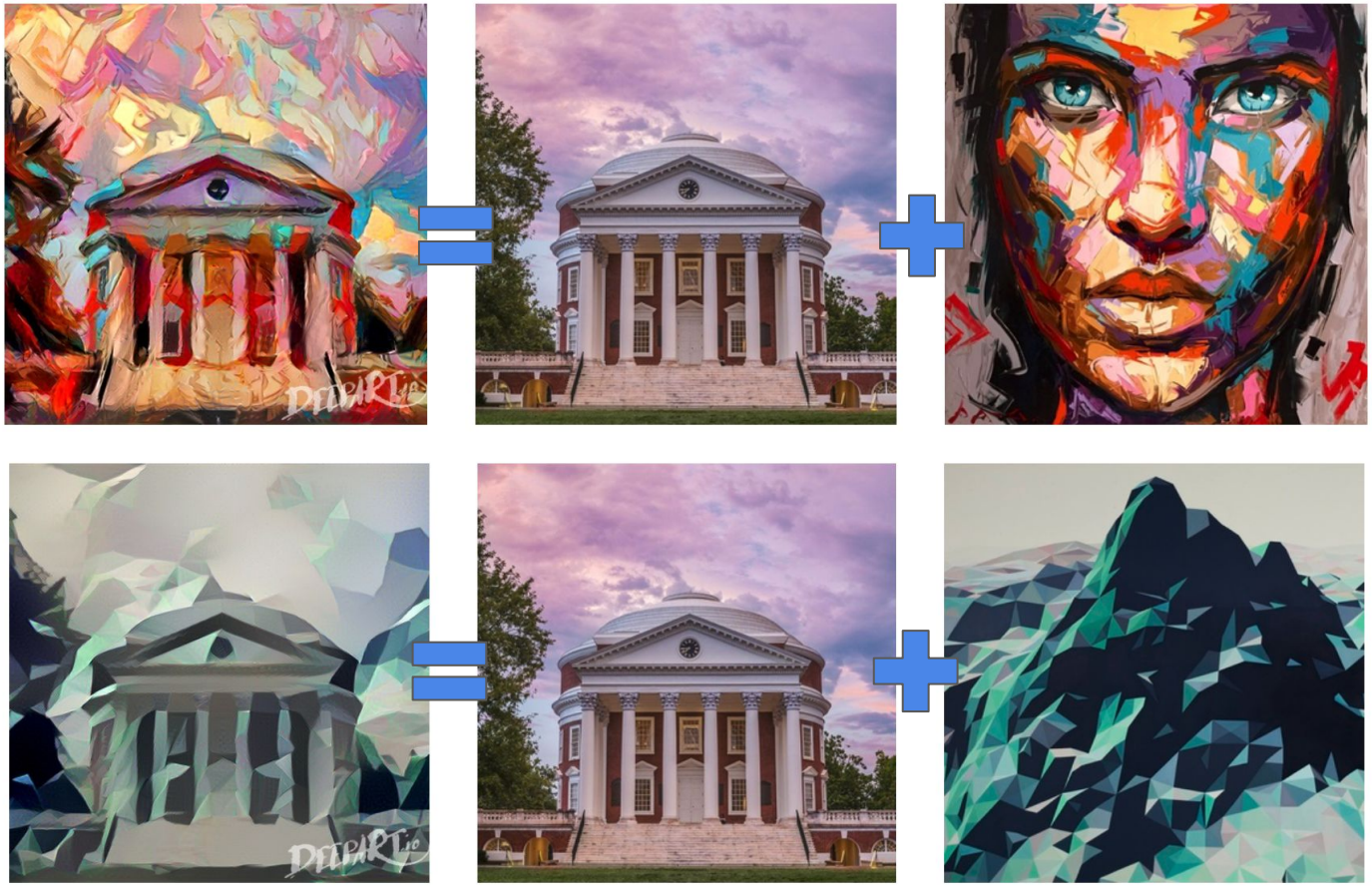
Style Transfer



Original image is CC0 public domain
Starry Night and Tree Roots by Van Gogh are in the public domain
Bokeh image is in the public domain
Stylized images copyright Justin Johnson, 2017;

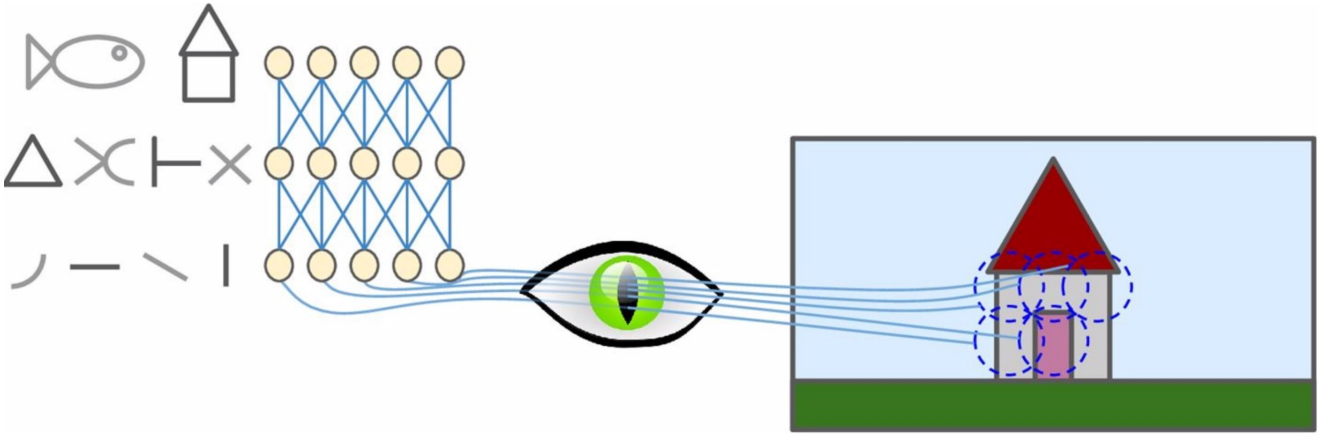
Gatys et al, "Image Style Transfer using Convolutional Neural Networks", CVPR 2016
Gatys et al, "Controlling Perceptual Factors in Neural Style Transfer", CVPR 2017

Style transfer is applied on the Rotunda



Style transfer is applied on my photo

History: The Architecture of the Visual Cortex



David Hubel and Torsten Wiesel shown that many neuron in the visual cortex has a small local receptive field. In 1981, they received a Nobel Prize for this discovery.

Some neurons only react to horizontal lines, while others reacts to lines with different orientations

Some neurons have larger receptive fields, so they react to more complex patterns based on output of lower-level neuron → powerful architecture is able to detect all sorts of complex patterns.

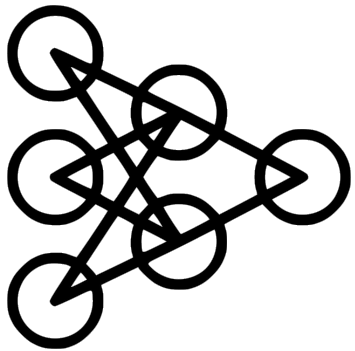
Convolutional Neural Networks (CNNs)

Inspired by the architecture of the visual cortex → convolutional neural network

First successful case of the CNNs: Yann LeCun in 1998 introduced the famous paper and LeNet-5 architecture, widely use to recognize handwritten check numbers.

Two building blocks: **Convolutional layers** and **Pooling layers**

Convolutional Layer



Convolutional Layers

- Neurons in the first convolutional layer are not connected to every pixel in the input image, but **only to pixels in their receptive fields**
- Each neuron in the second convolutional layer is connected only to **neurons located within a small rectangle** in the first layer.
- Concentrate on **low-level features in the first hidden layer,** then assemble them into higher-level features, and so on → work well for image recognition

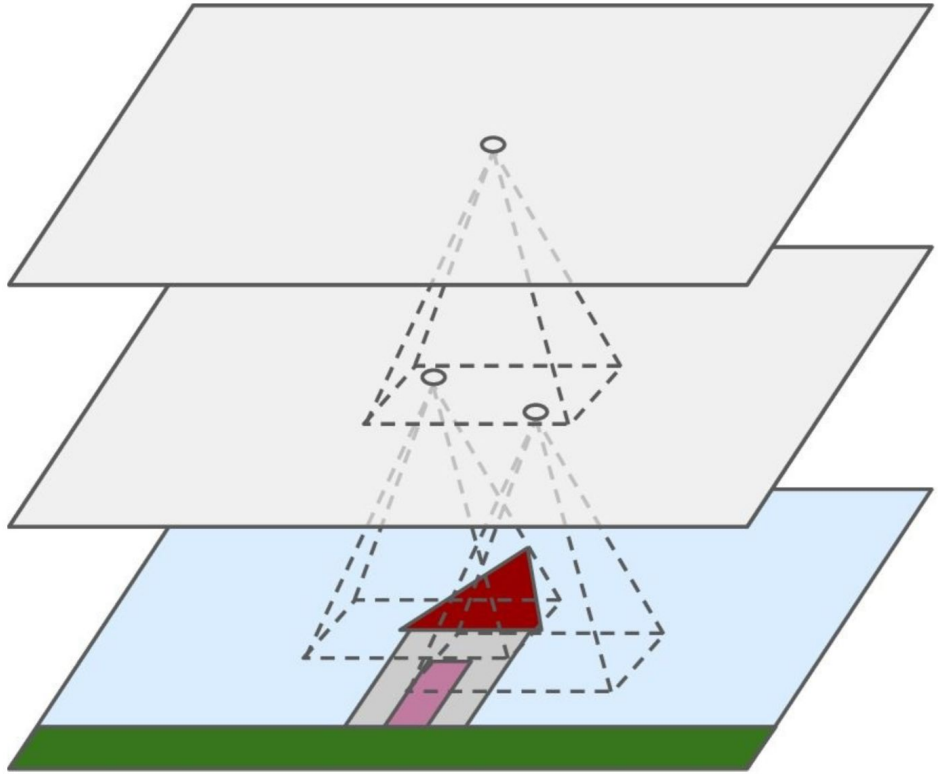
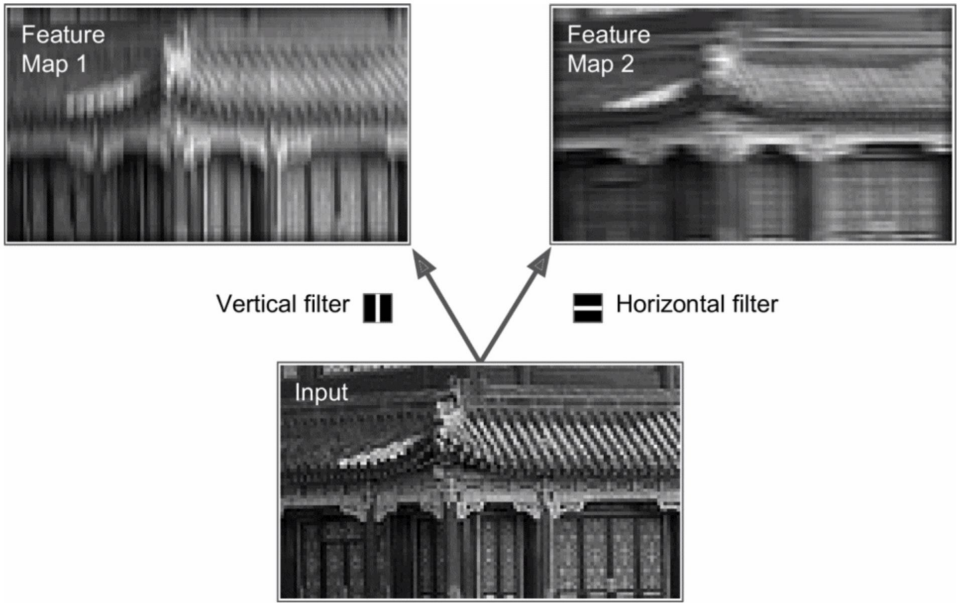
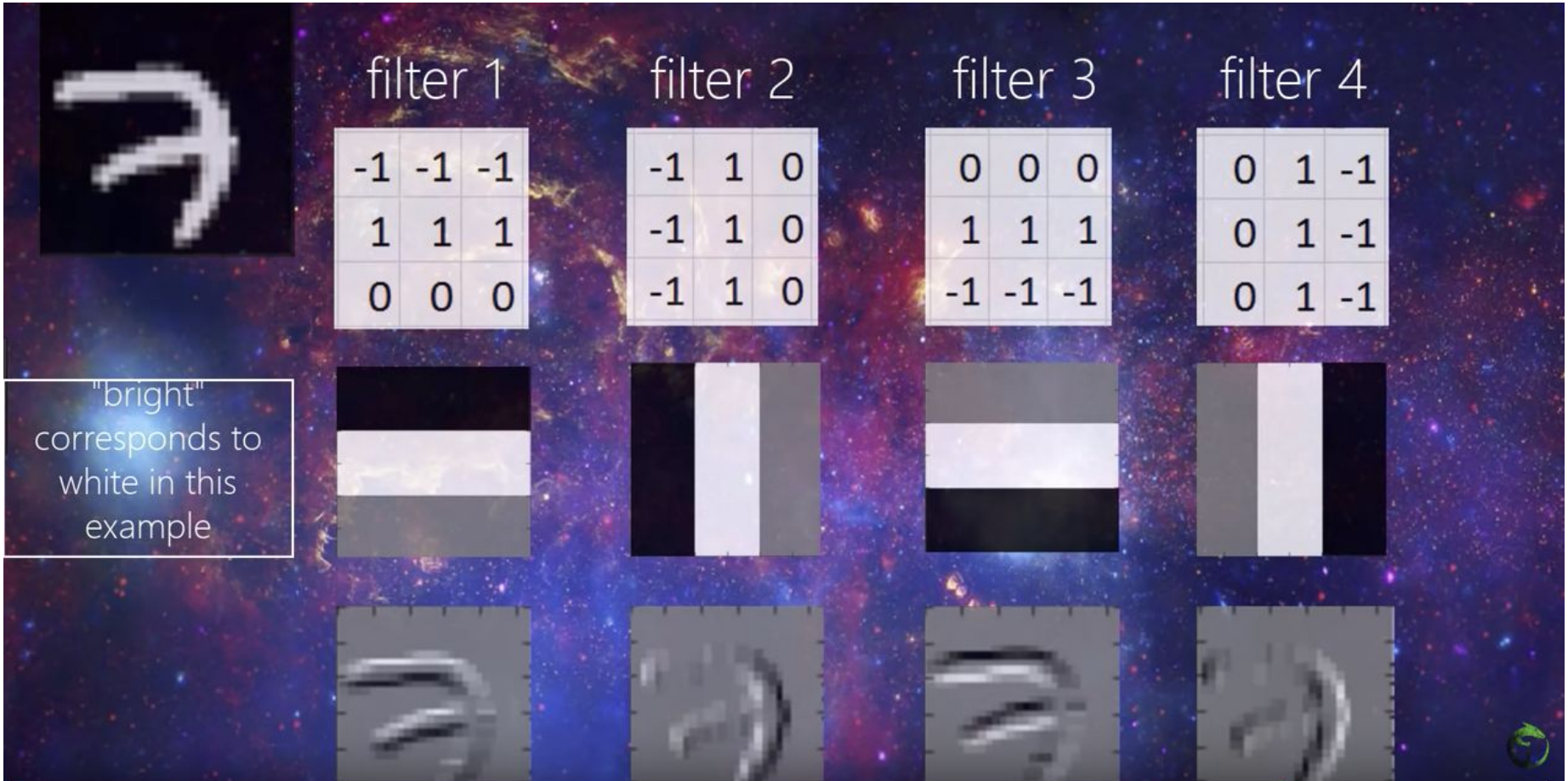


Image Filtering

- A neuron's weights can be represented as a small image the size of the receptive field. These weights are call filters (or convolutional kernels).
- **Vertical Filter:** neurons using these weights will ignore everything except for central vertical line. In Feature Map 1, vertical white line get enhanced while the rest is blurred out.
- Same for **Horizontal Filter** and horizontal line.

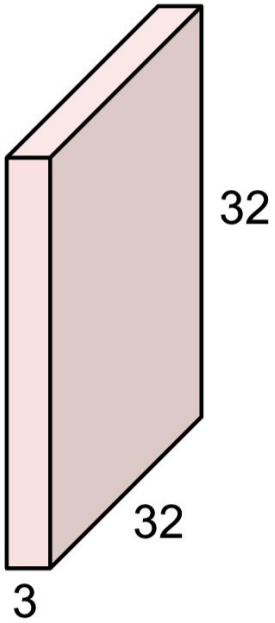


Convolution Filters for number 7



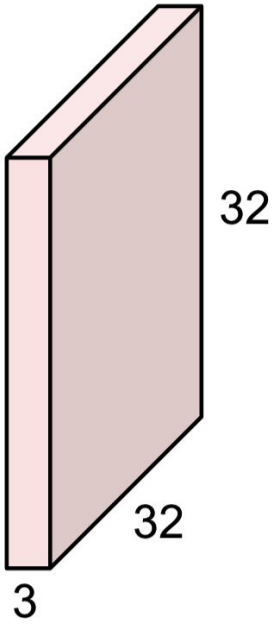
Input Image

32x32x3 image → preserve spatial structure

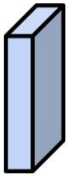


Filter

32x32x3 image → 5x5x3 filter → always extend the full depth of the input image

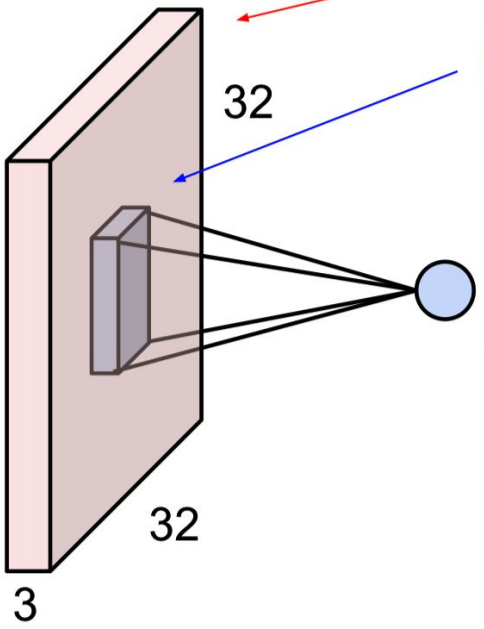


5x5x3 filter



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

Applying dot product



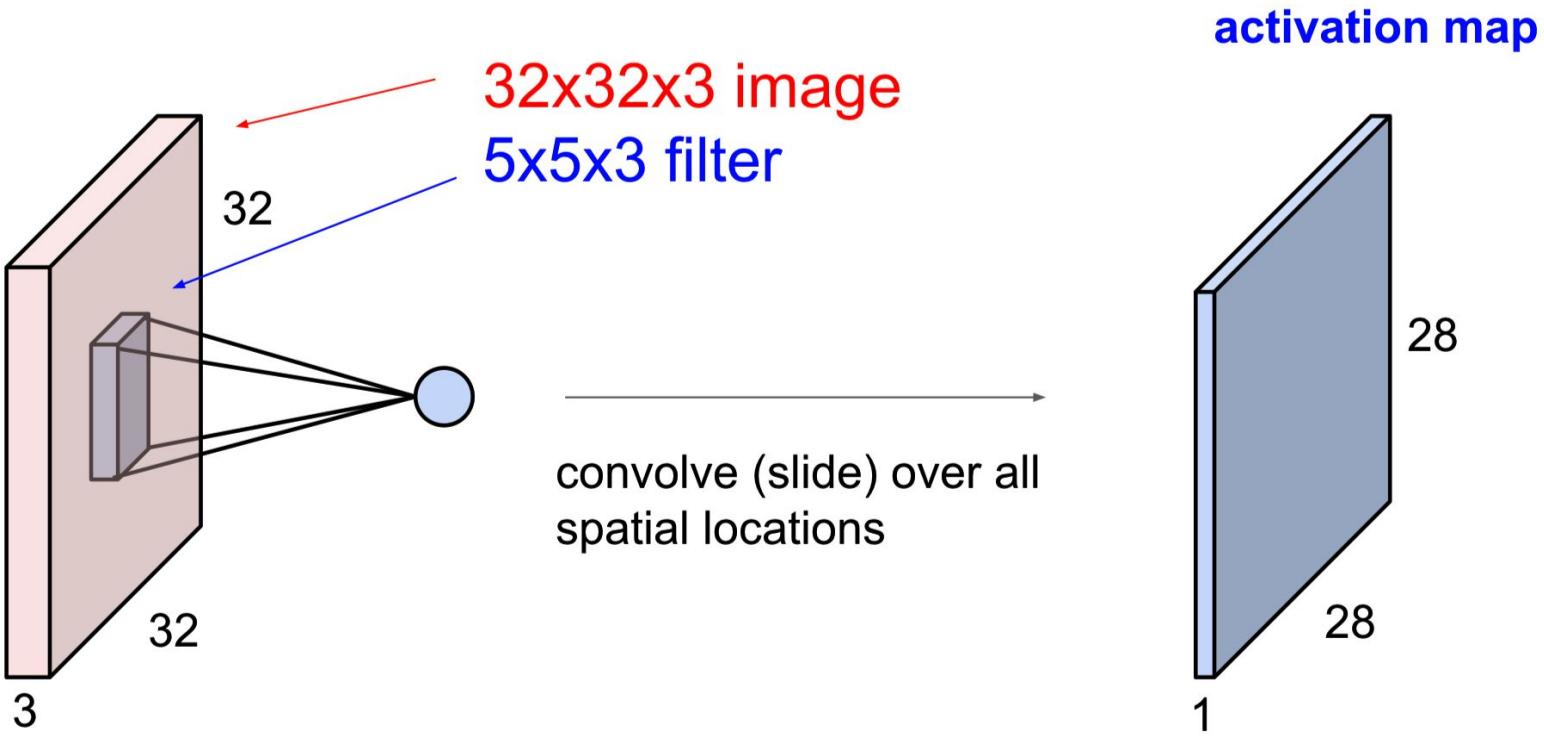
32x32x3 image

5x5x3 filter w

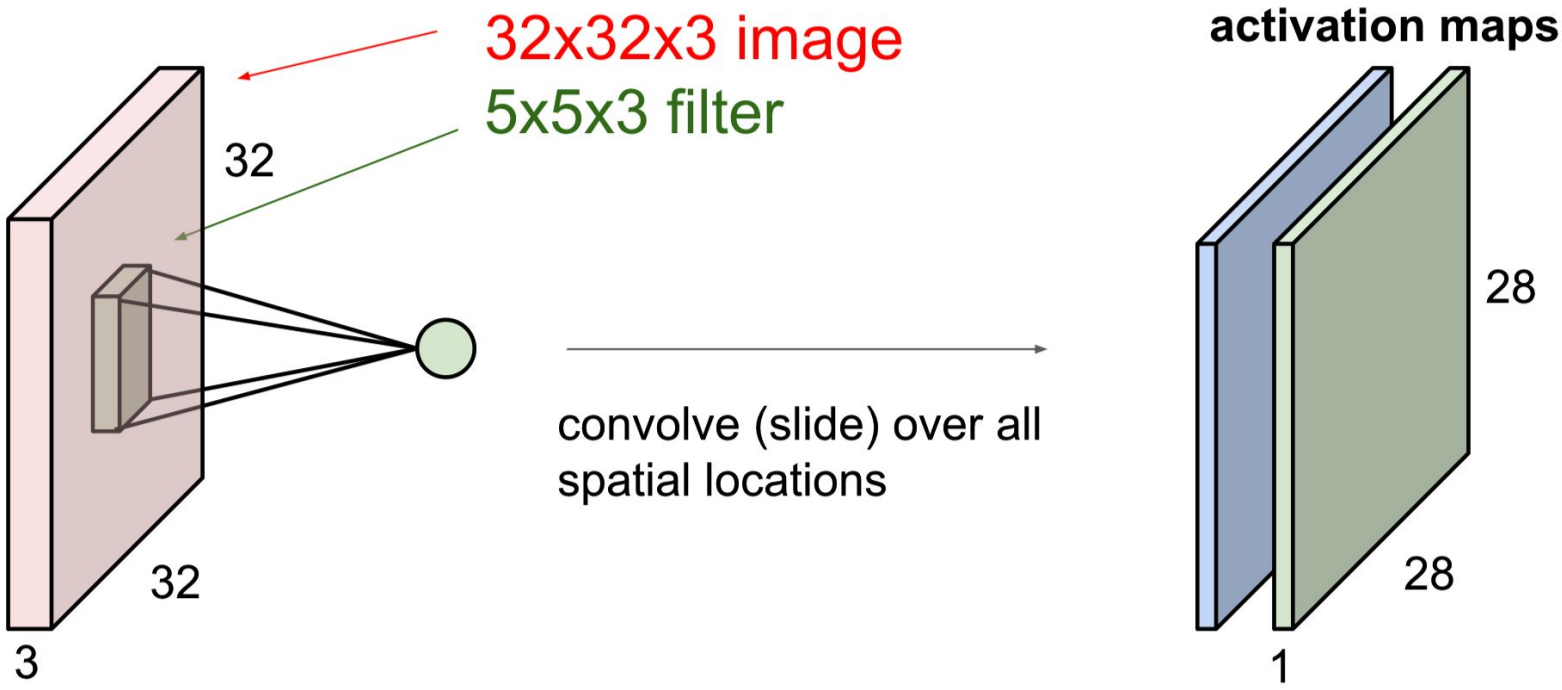
1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image (i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

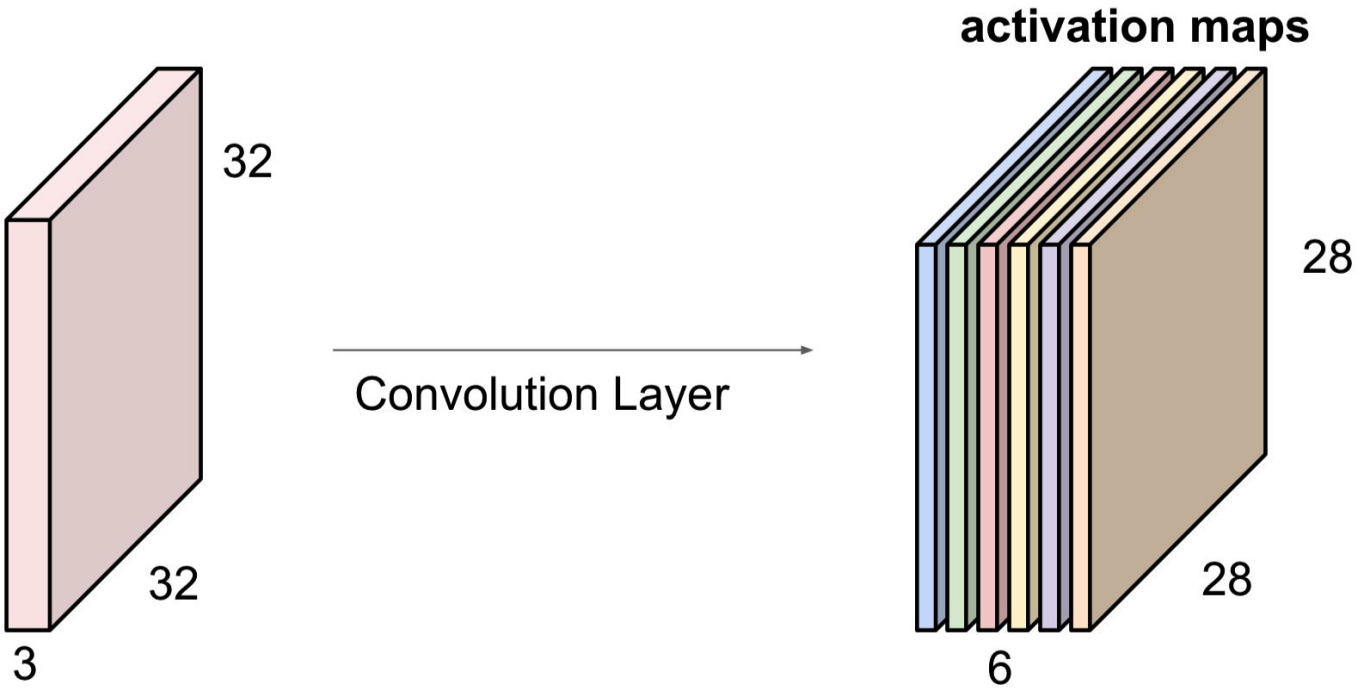
Sliding over all spatial locations



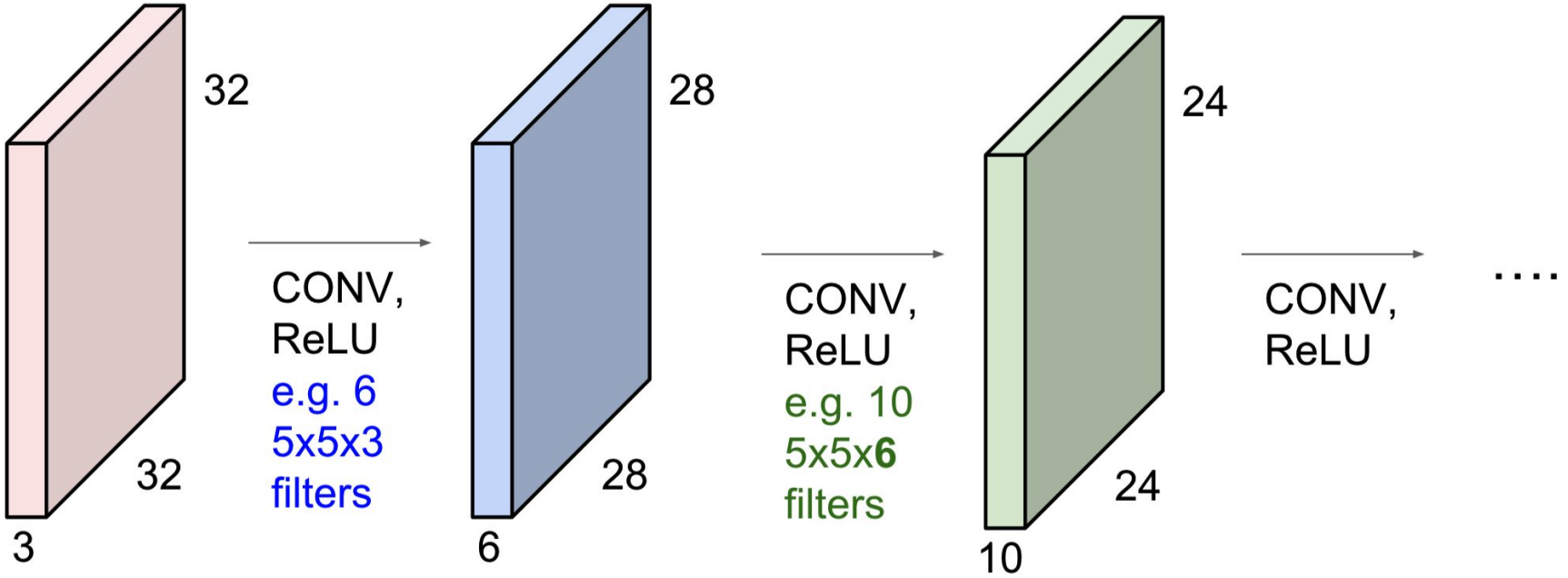
Consider another (green) filter



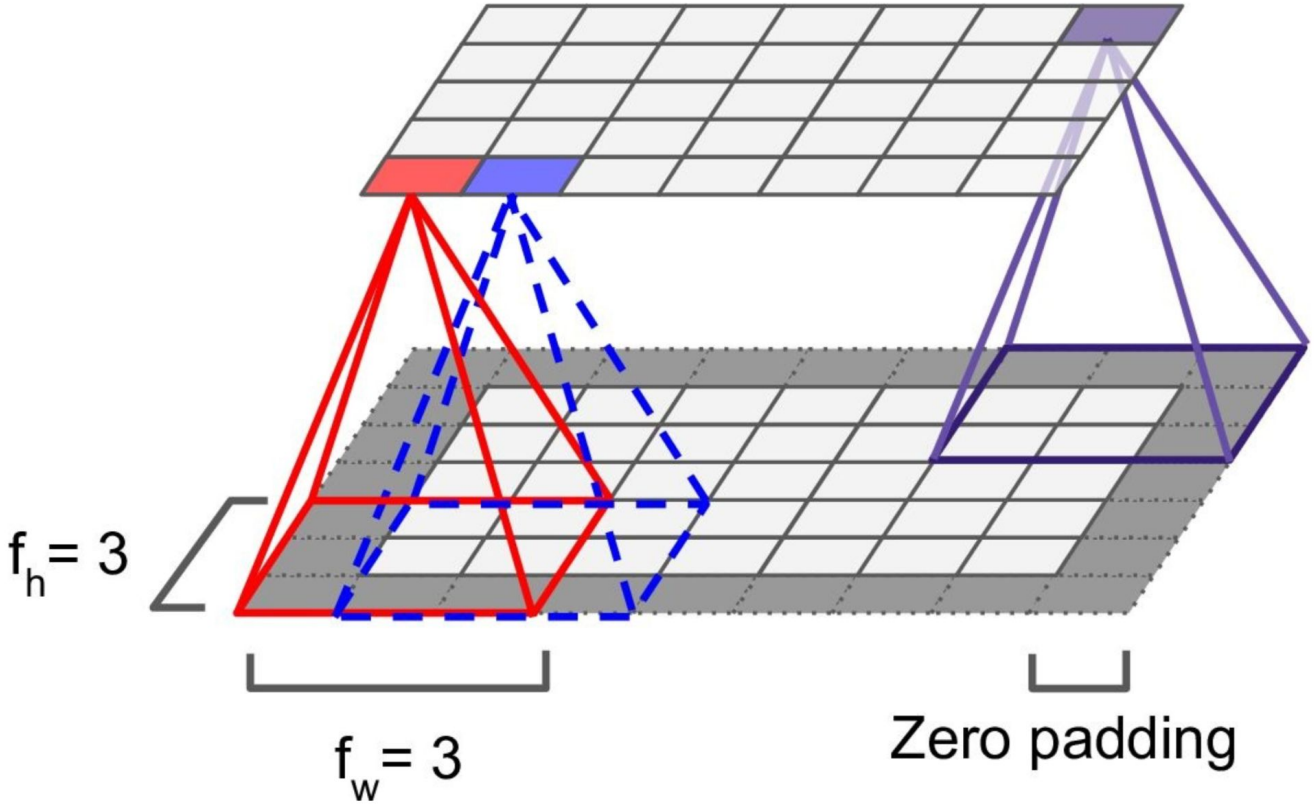
Applying multiple filters



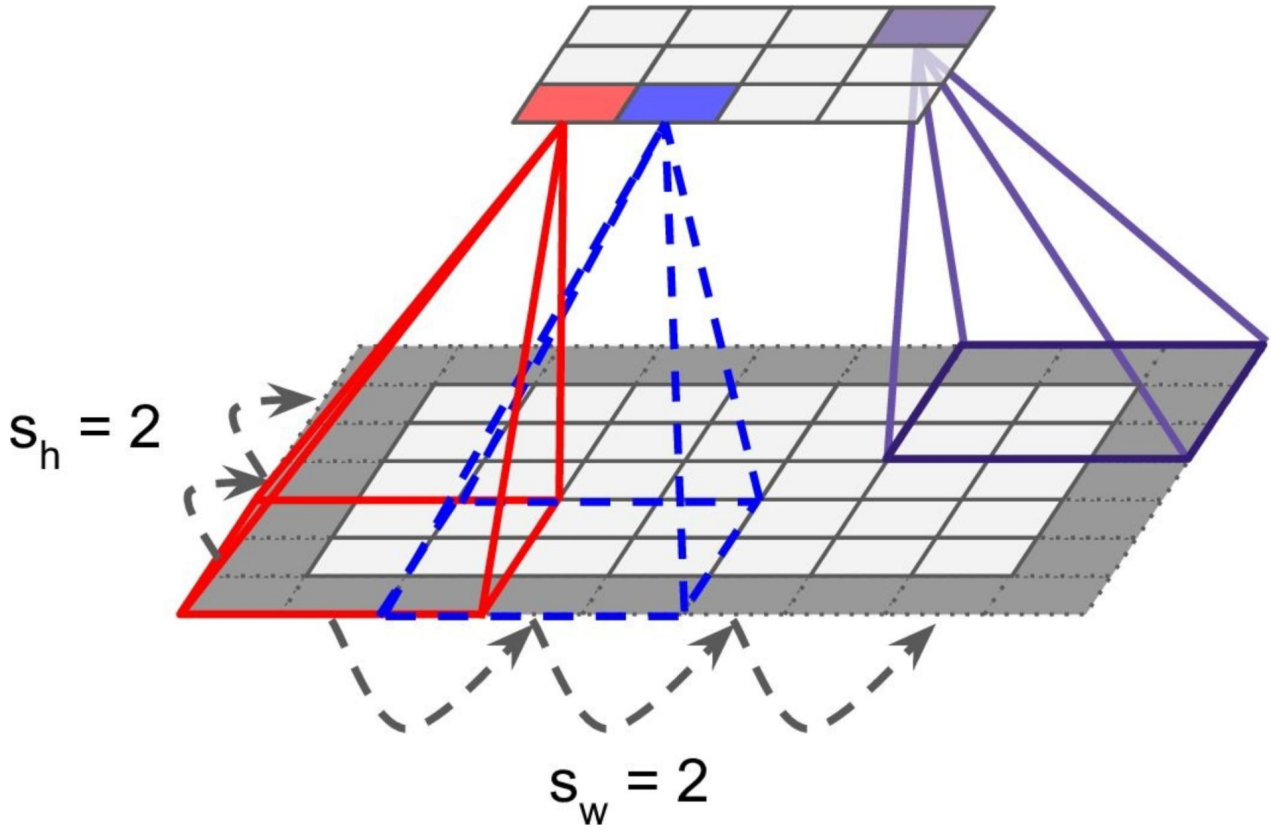
Stack the activation maps



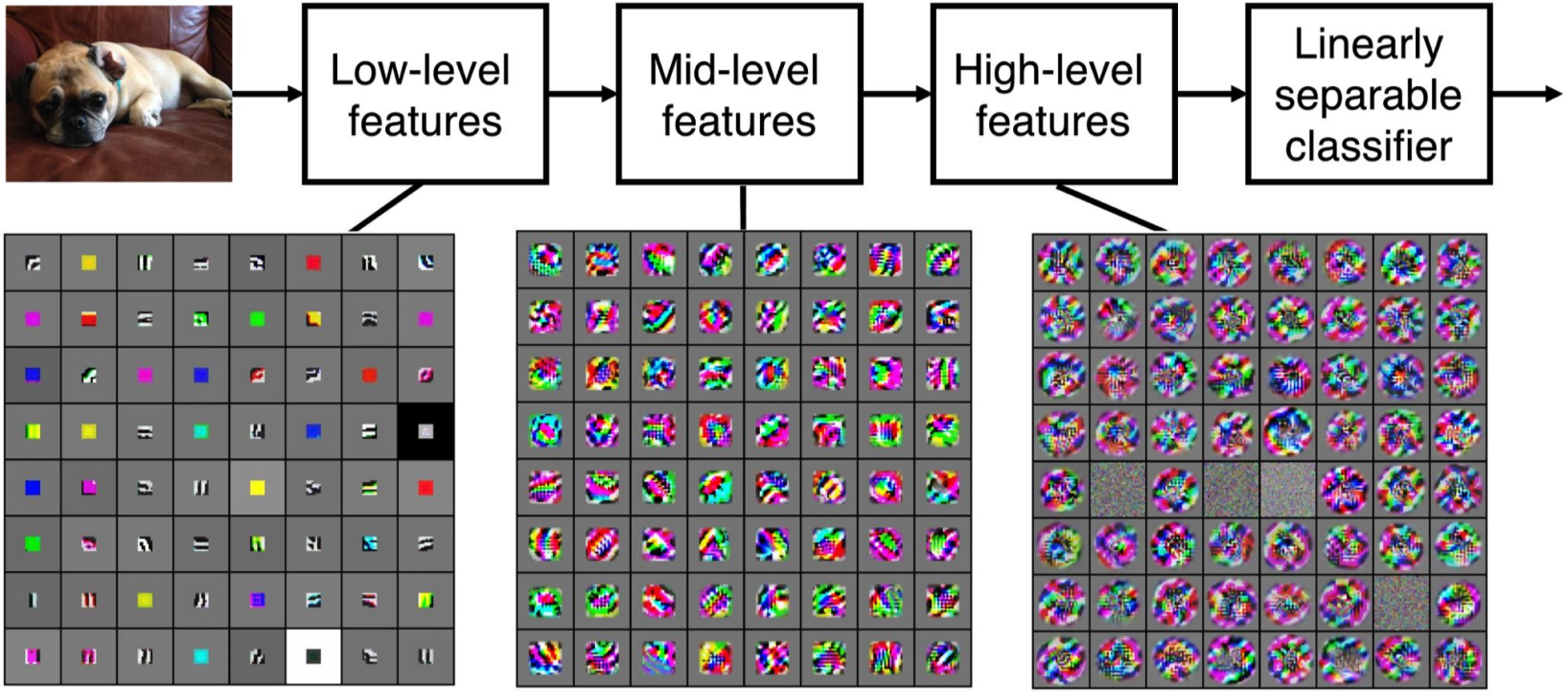
Zero Padding



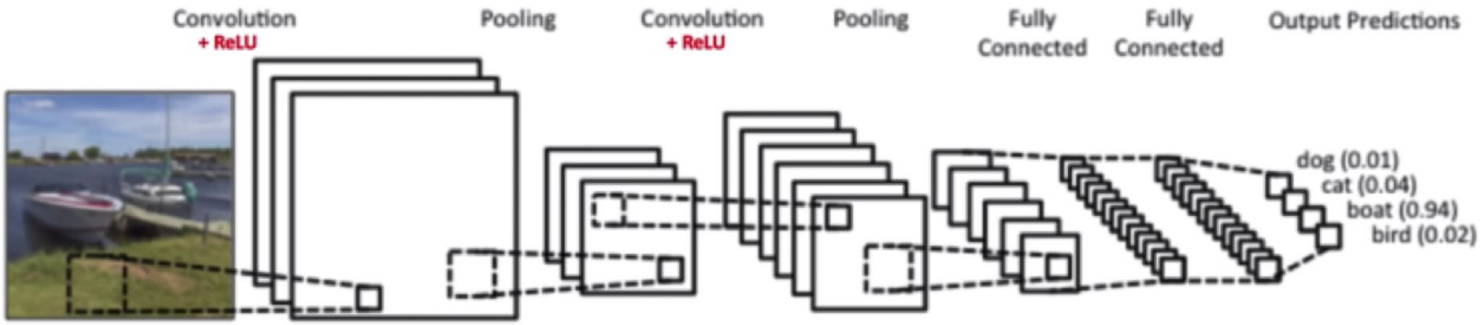
Stride



Low-level to high-level features



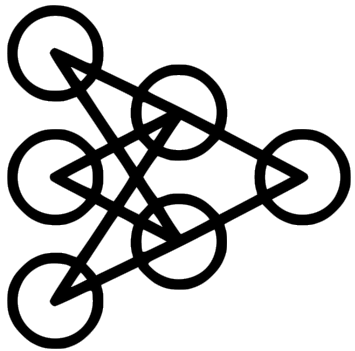
In real images



Hyperparameters and Memory Requirements

- Unfortunately, convolutional layers have lots of hyperparameters: number of filters, their height and width, strides, and padding types → finding the right set of parameter can be very time-consuming.
- Convolutional layers also requires a **huge amount of memory** during training because the backward pass requires all intermediate values computed.
  - Consider a layer with 5x5 kernels, outputting 200 feature maps of size 150 x 100 (which is 15,000 neurons), with stride 1 and SAME padding, each neuron computes the weighted sum of its 5x5x3 = 75 inputs. That's a total of **225 million** float multiplications.
  - Moreover, feature maps will occupy 200x150x100x32=96 millions bits (11.4 MB) of memory for each training instance. So for a batch of 100 instance, that's **1 GB**!

Pooling Layer

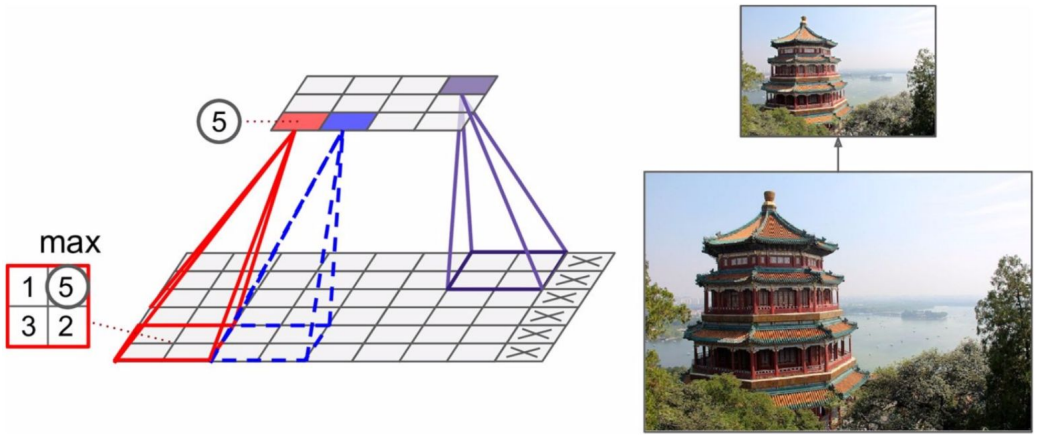


Pooling Layer

The goal is to subsample the input image in order to reduce the computational load, memory usage, and number of parameters.

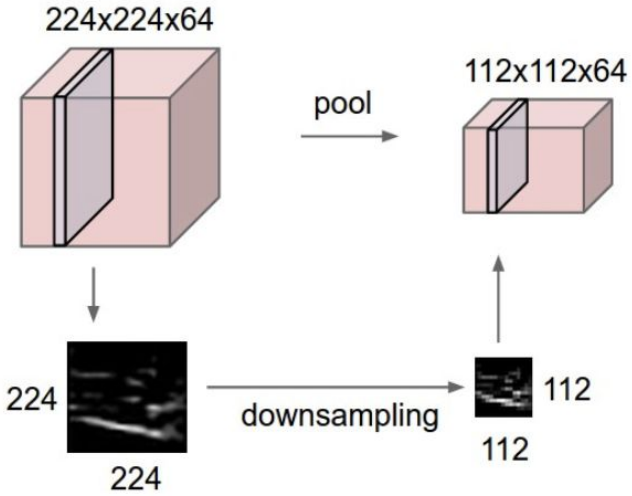
Each neuron in a pooling layer is connected to the outputs of a limited number of neurons in the previous layer (a small rectangular receptive field)

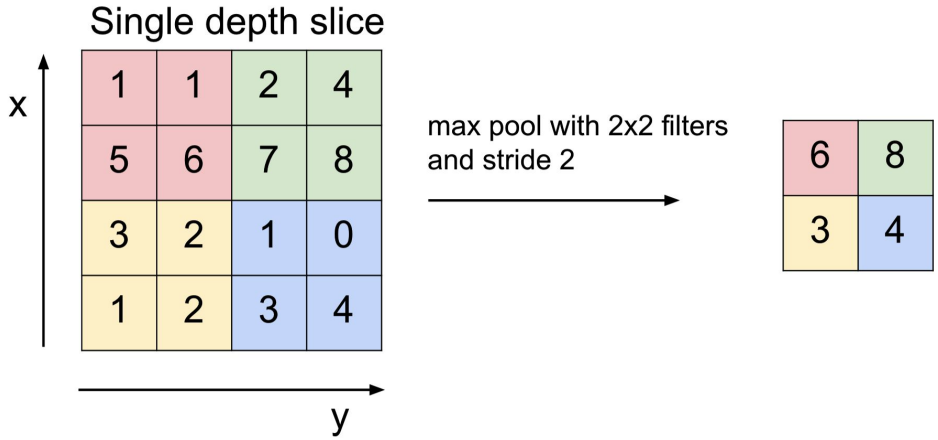
A pooling neuron has no weights, all it does is aggregate the inputs (using max or mean function)



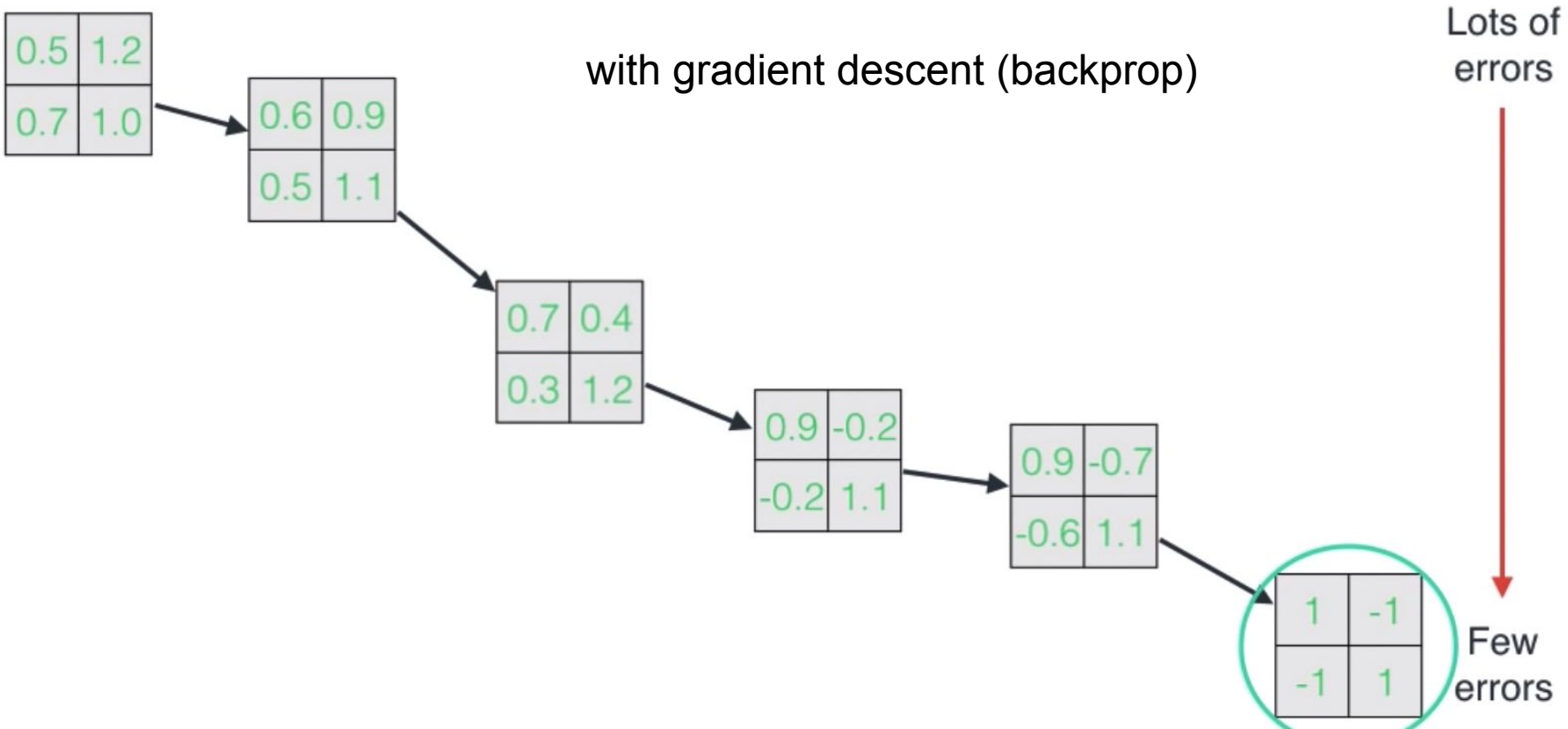
75% of the input values is dropped!

Max Pooling

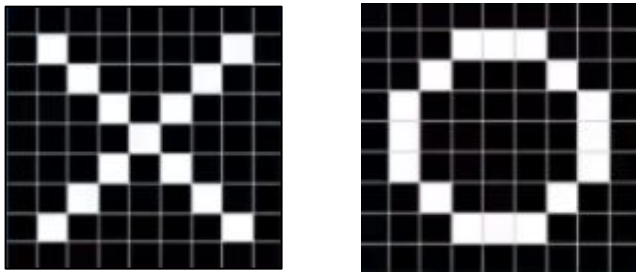

224x224x64
pool
112x112x64
224
224
downsampling
112
112


Single depth slice
x
y
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters
and stride 2
6 8
3 4

Learning the weights in the filters

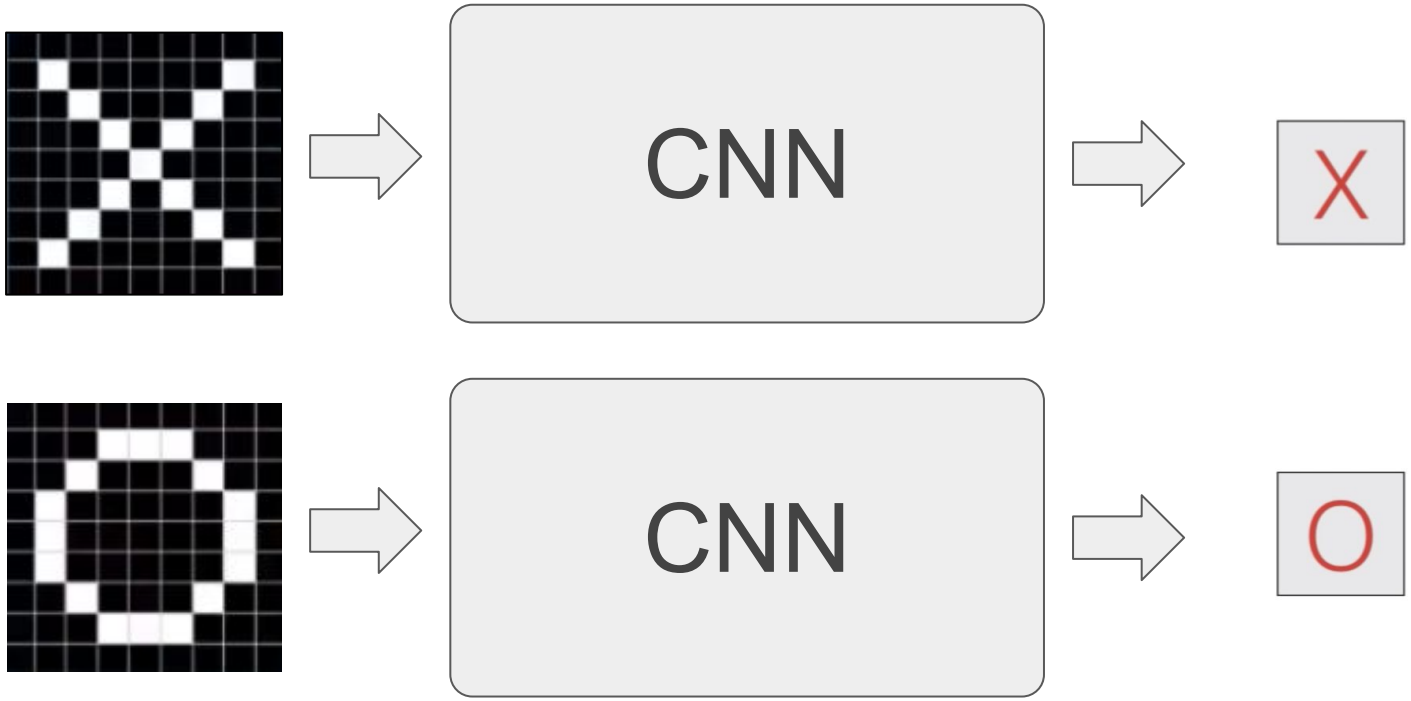


A toy example

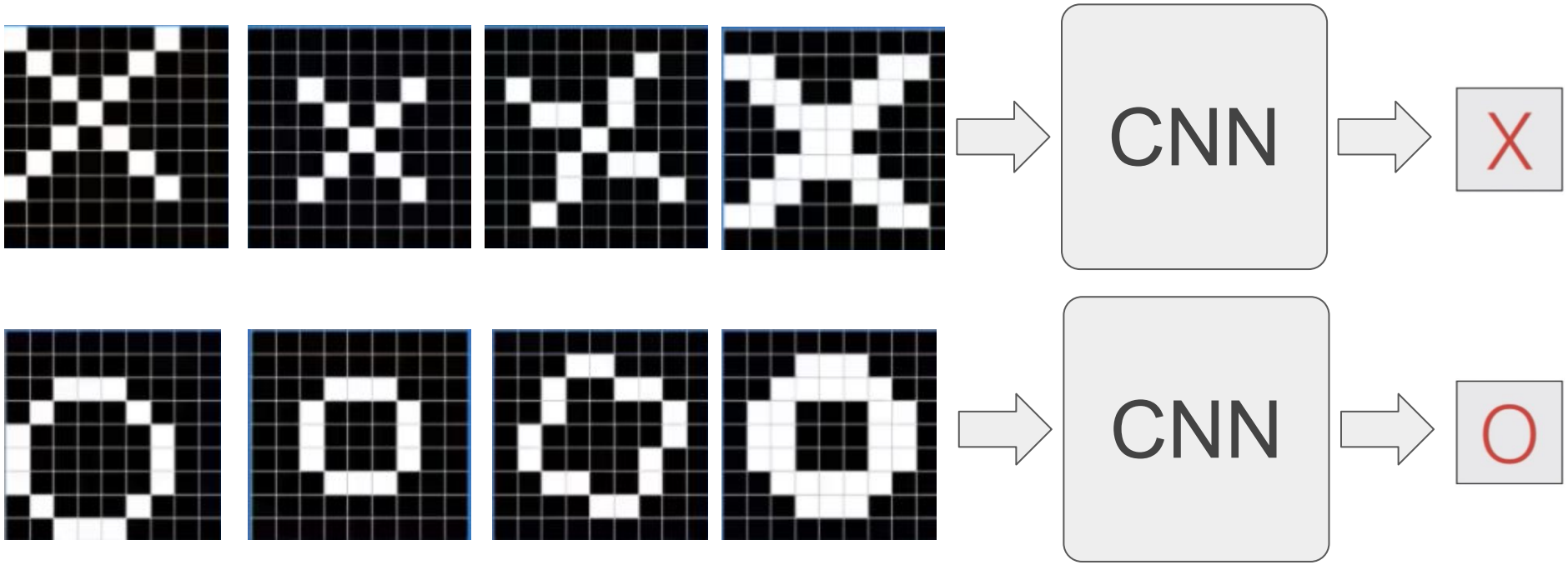


A simple example of classifying X or O

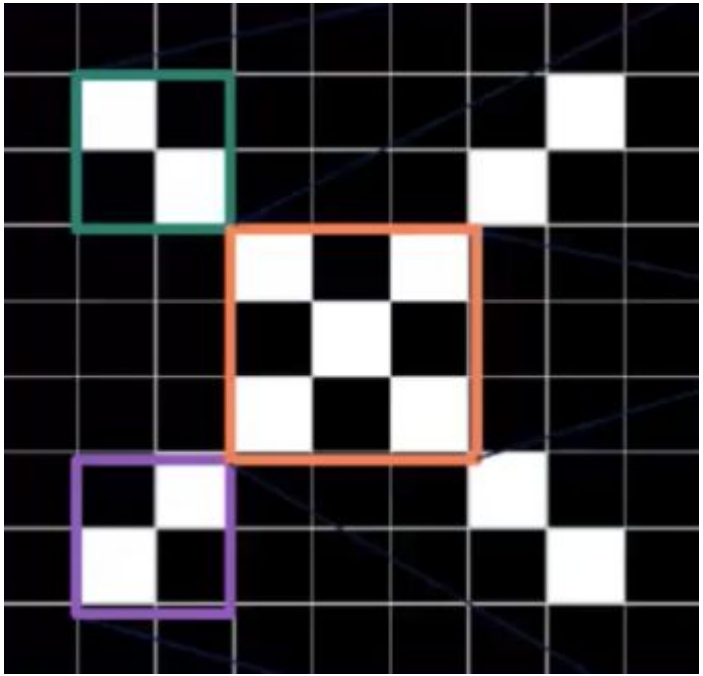
A simple example of classifying X or O

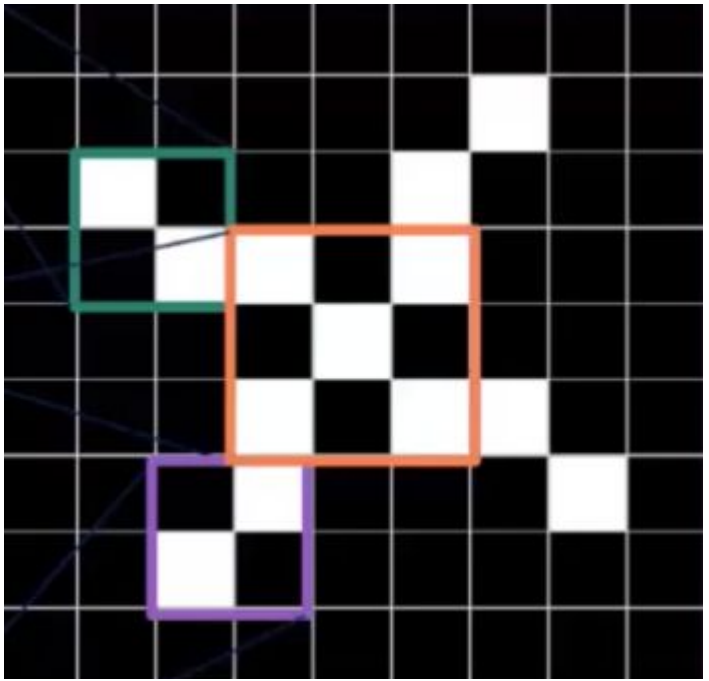


Tricky cases

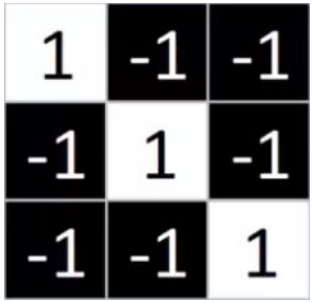


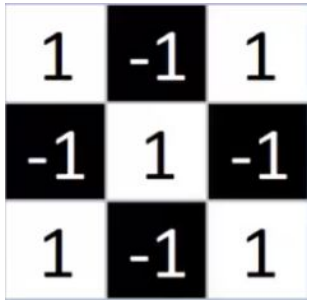
CNNs match pieces of the image

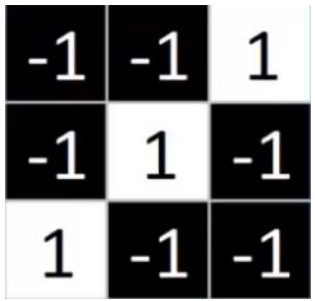




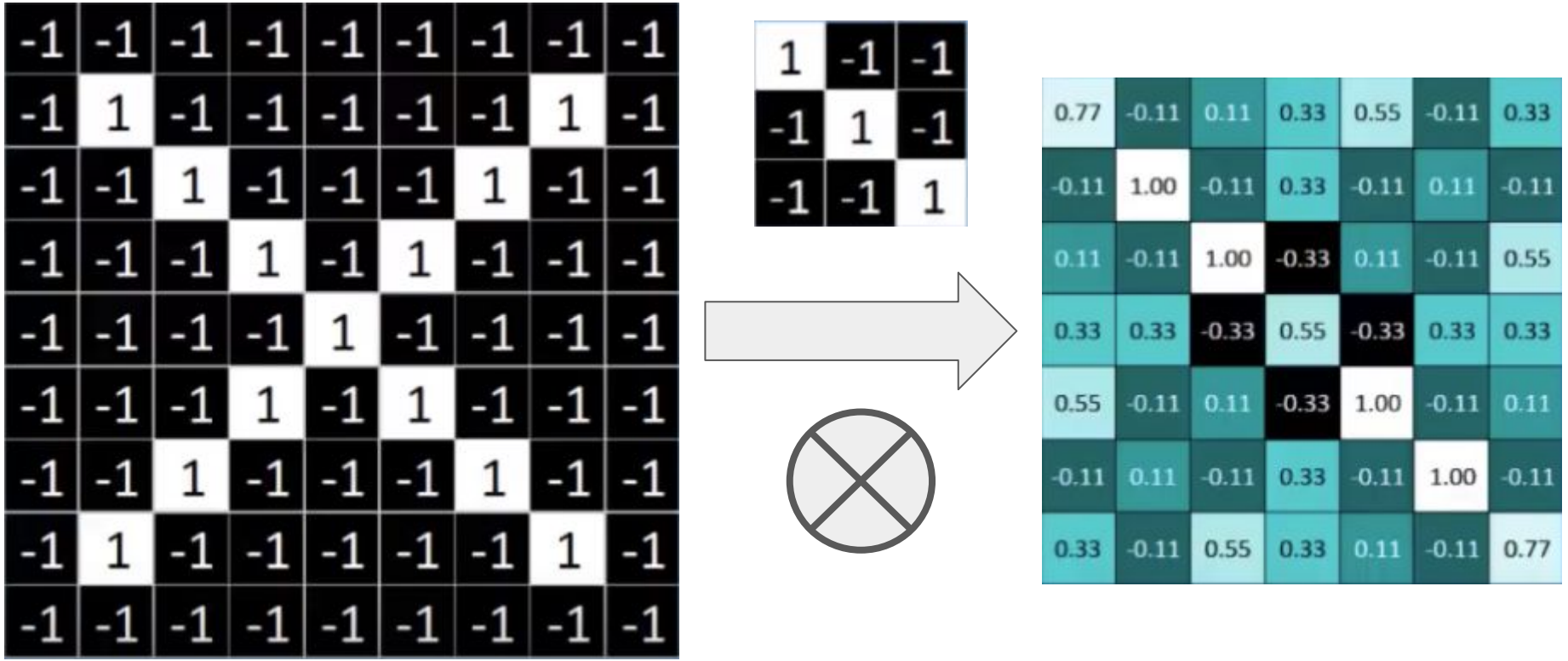
Features match pieces of the image





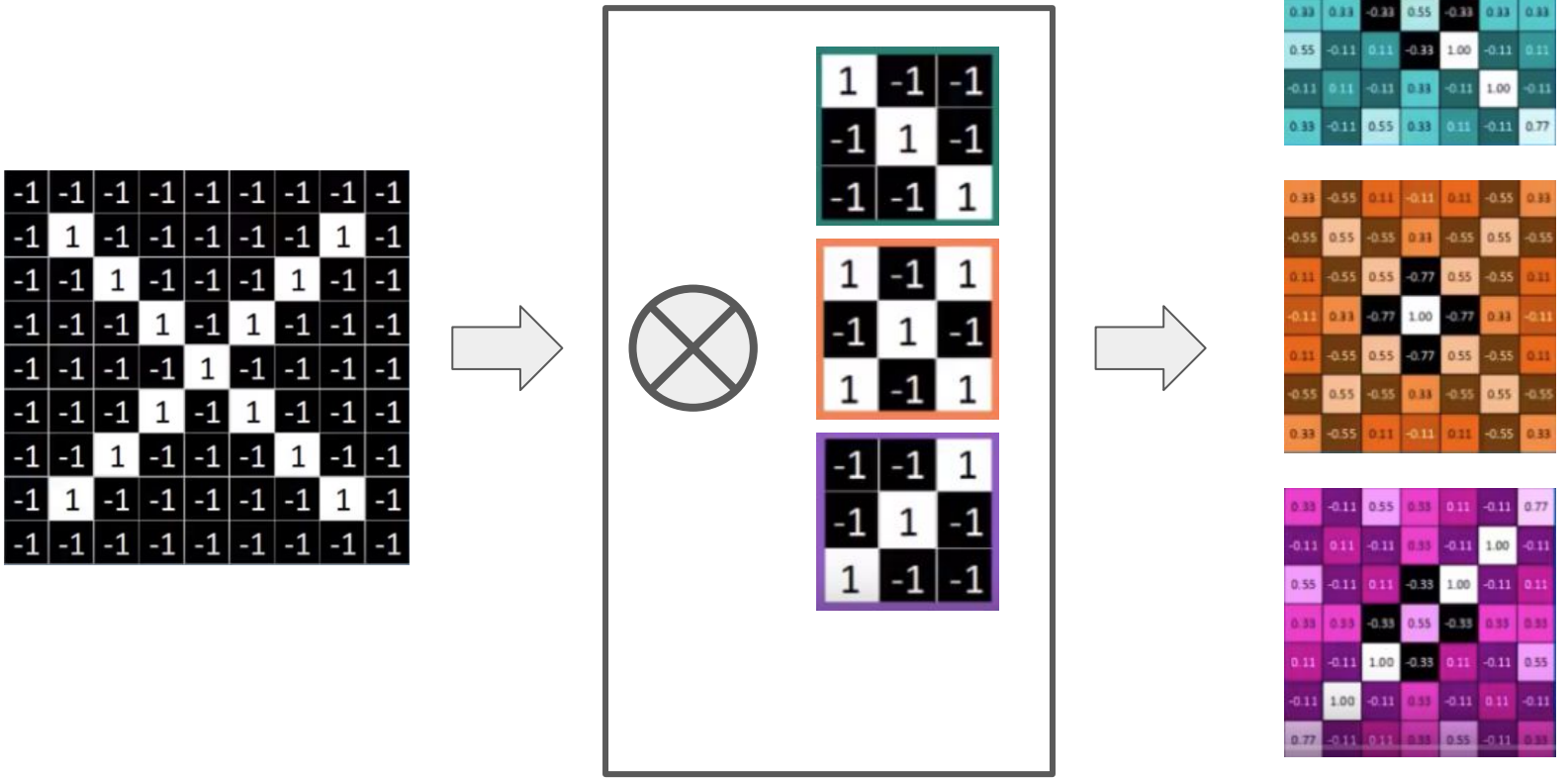


Convolution: Trying a match at every location



Convolution Layer

Input image becomes a stack of filtered images



ReLU

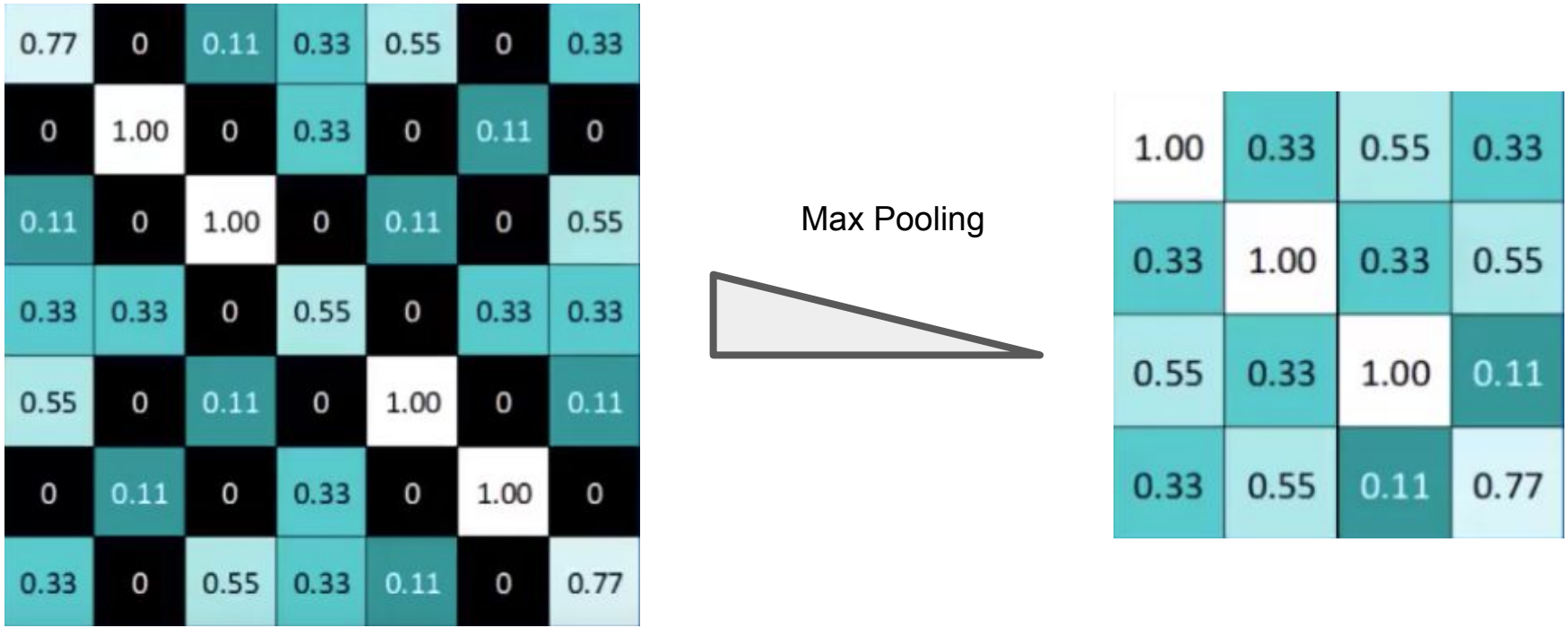
Image becomes one with no negative values

0.77	-0.11	0.11	0.33	0.55	-0.11	0.33
-0.11	1.00	-0.11	0.33	-0.11	0.11	-0.11
0.11	-0.11	1.00	-0.33	0.11	-0.11	0.55
0.33	0.33	-0.33	0.55	-0.33	0.33	0.33
0.55	-0.11	0.11	-0.33	1.00	-0.11	0.11
-0.11	0.11	-0.11	0.33	-0.11	1.00	-0.11
0.33	-0.11	0.55	0.33	0.11	-0.11	0.77

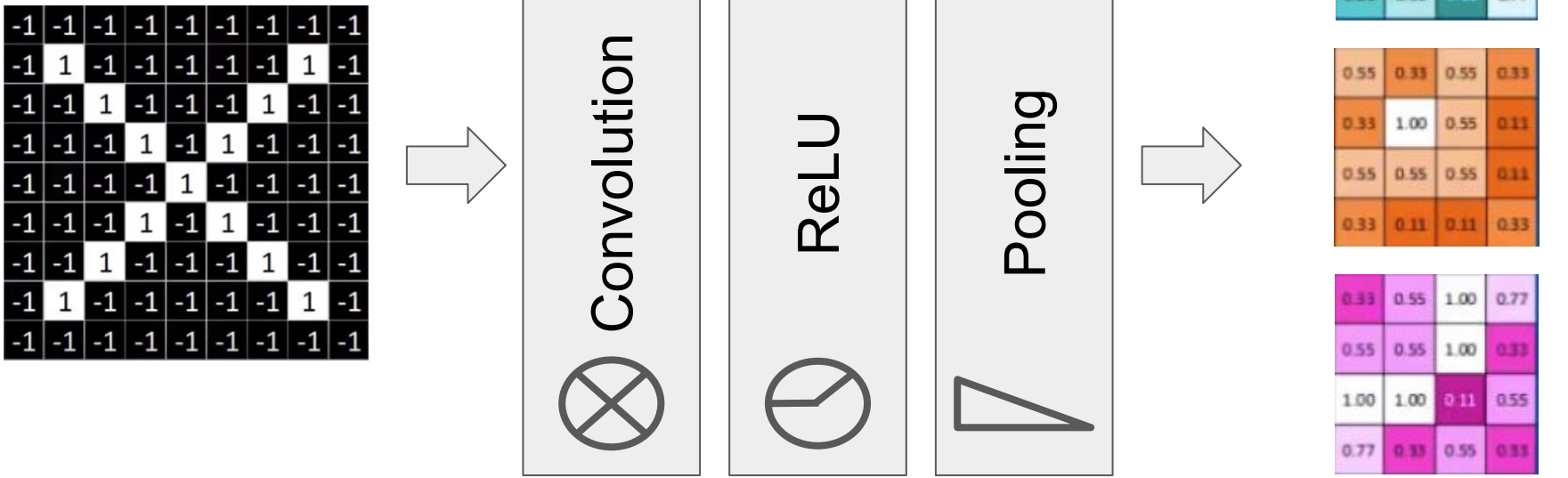
ReLU

0.77	0	0.11	0.33	0.55	0	0.33
0	1.00	0	0.33	0	0.11	0
0.11	0	1.00	0	0.11	0	0.55
0.33	0.33	0	0.55	0	0.33	0.33
0.55	0	0.11	0	1.00	0	0.11
0	0.11	0	0.33	0	1.00	0
0.33	0	0.55	0.33	0.11	0	0.77

Pooling

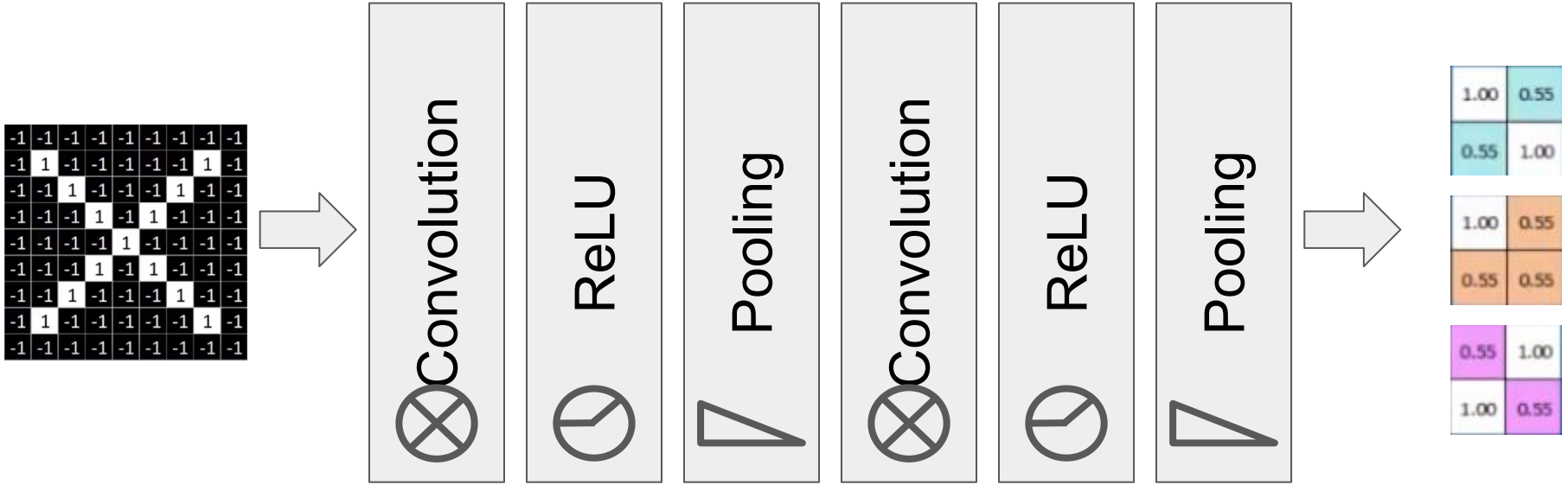

0.77 0 0.11 0.33 0.55 0 0.33
0 1.00 0 0.33 0 0.11 0
0.11 0 1.00 0 0.11 0 0.55
0.33 0.33 0 0.55 0 0.33 0.33
0.55 0 0.11 0 1.00 0 0.11
0 0.11 0 0.33 0 1.00 0
0.33 0 0.55 0.33 0.11 0 0.77
Max Pooling
1.00 0.33 0.55 0.33
0.33 1.00 0.33 0.55
0.55 0.33 1.00 0.11
0.33 0.55 0.11 0.77

Layers get stacked



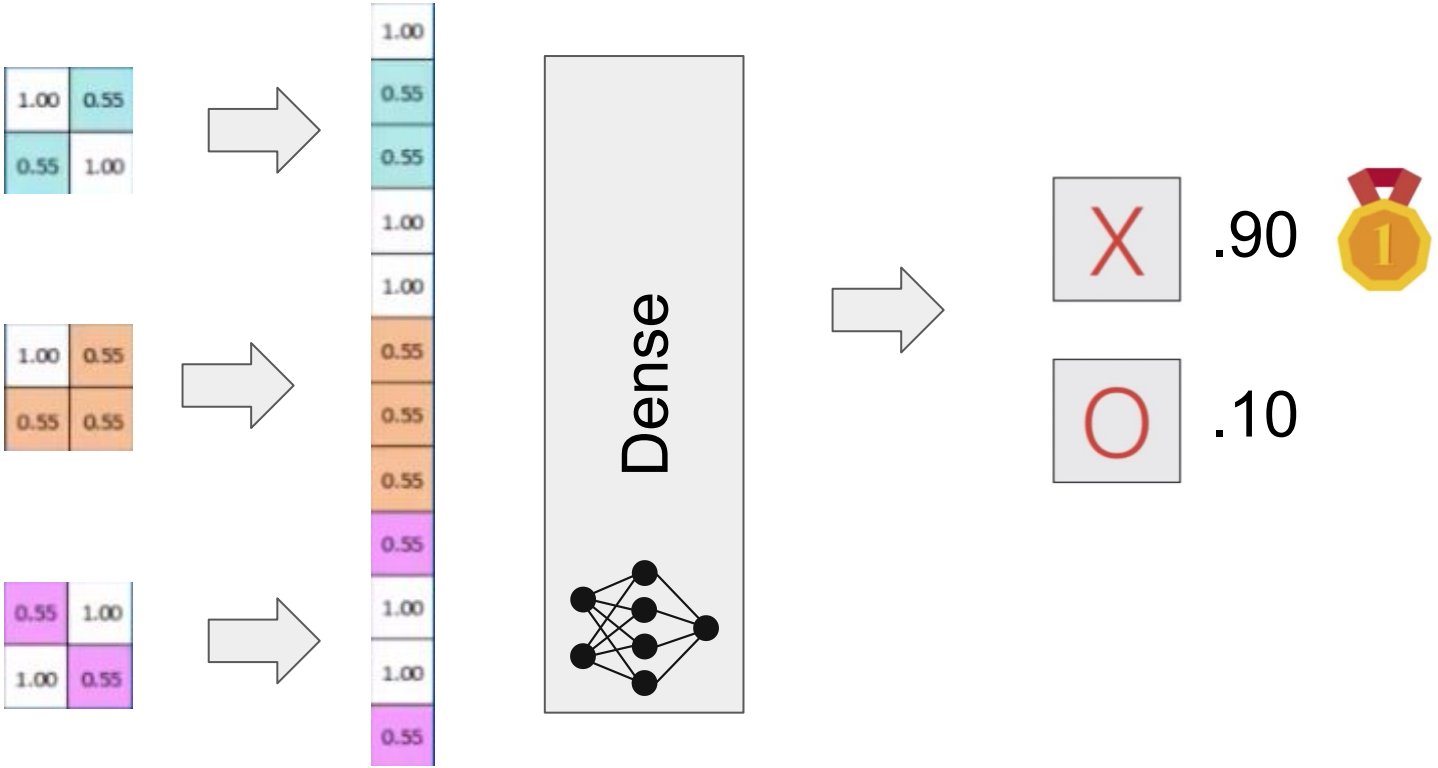
Deep Stacking

Layers can be repeated several times



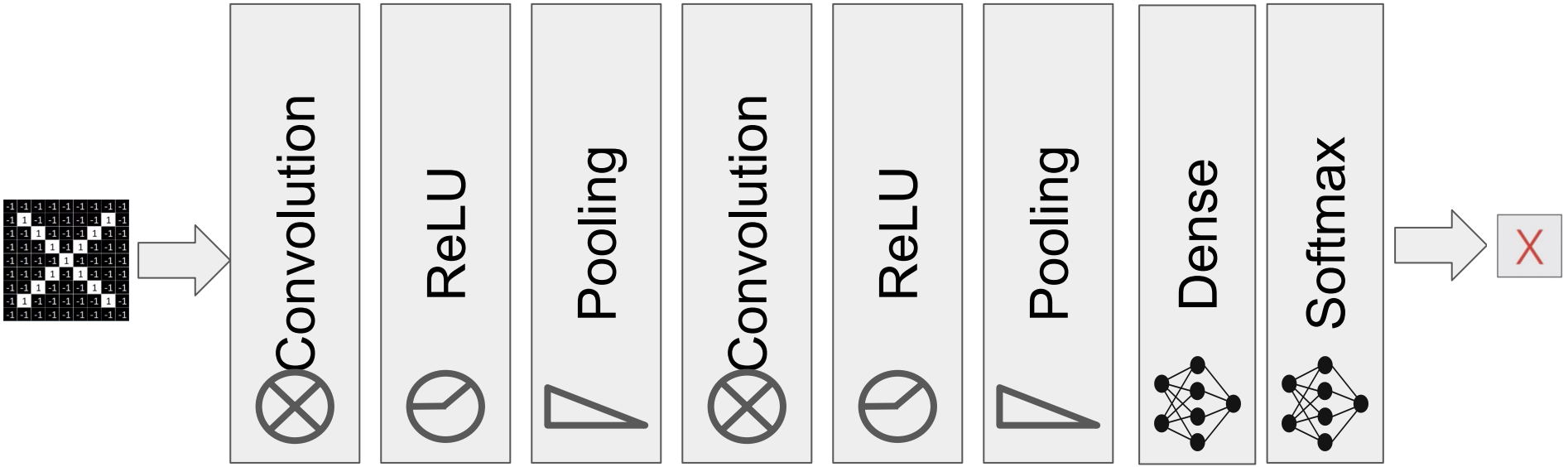
Fully Connected Layer

Every values gets a vote

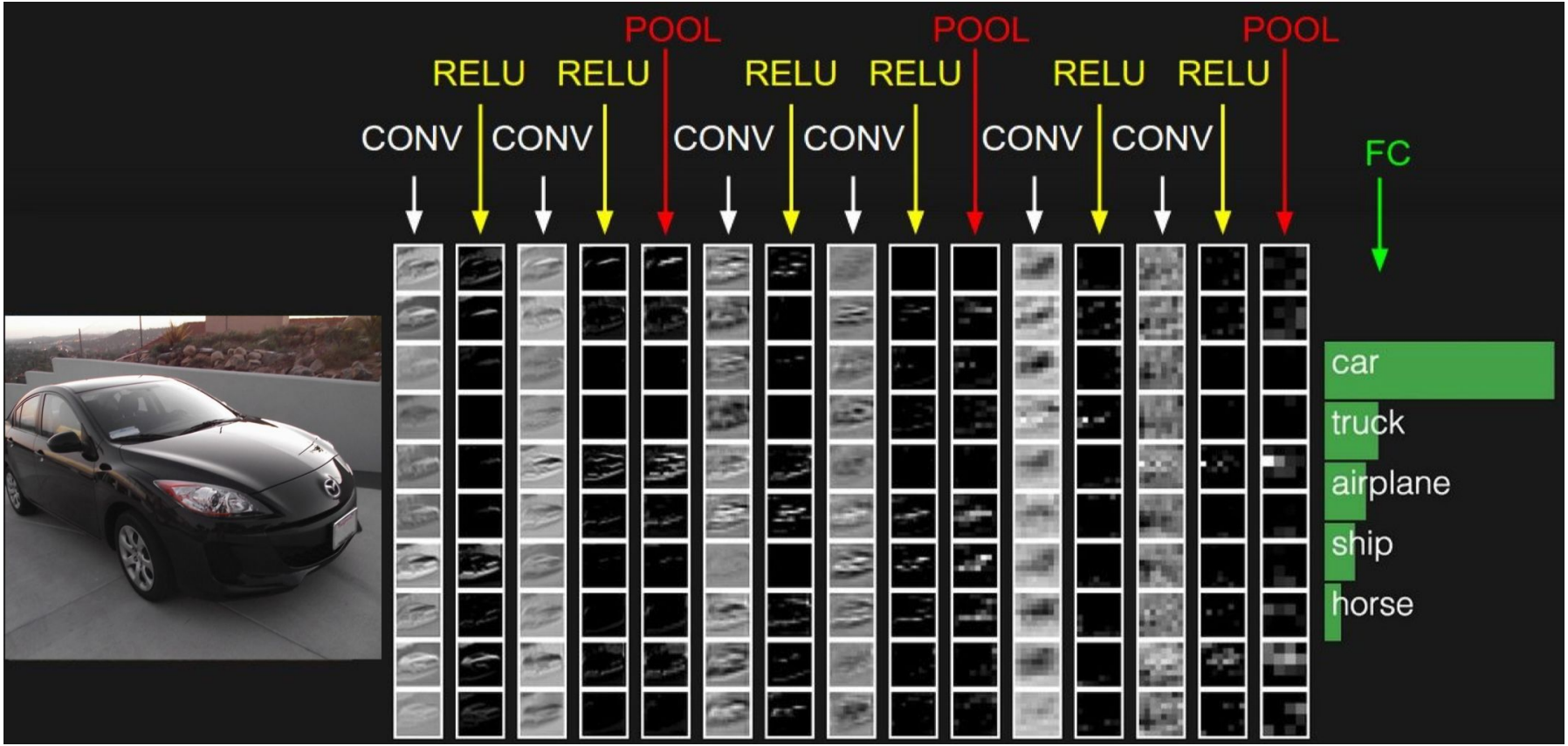


Putting together in blocks

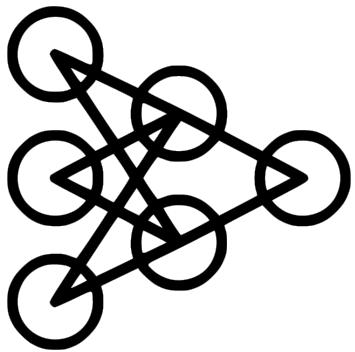
A set of pixels becomes a set of votes



Real example

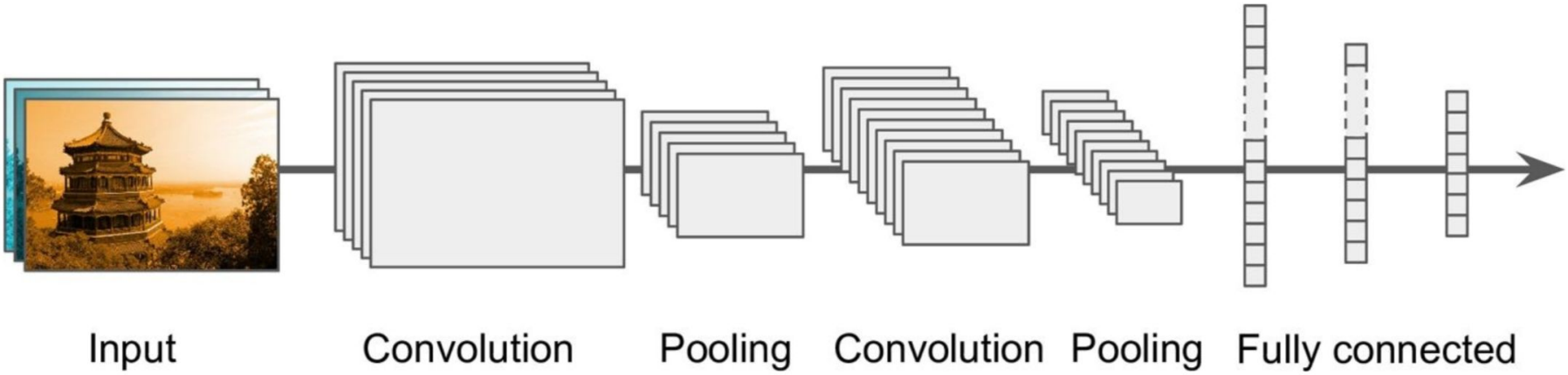


CNN Architectures



Typical CNN Architectures

Stack a few convolutional layers (each followed by a ReLU layer), then a pooling layer, then another few convolutional layers (+ReLU), then pooling layer, and so on getting deeper and smaller. At the top of the stack is the regular feedforward neural network of fully connected layers (+ReLU), and the final softmax layer.



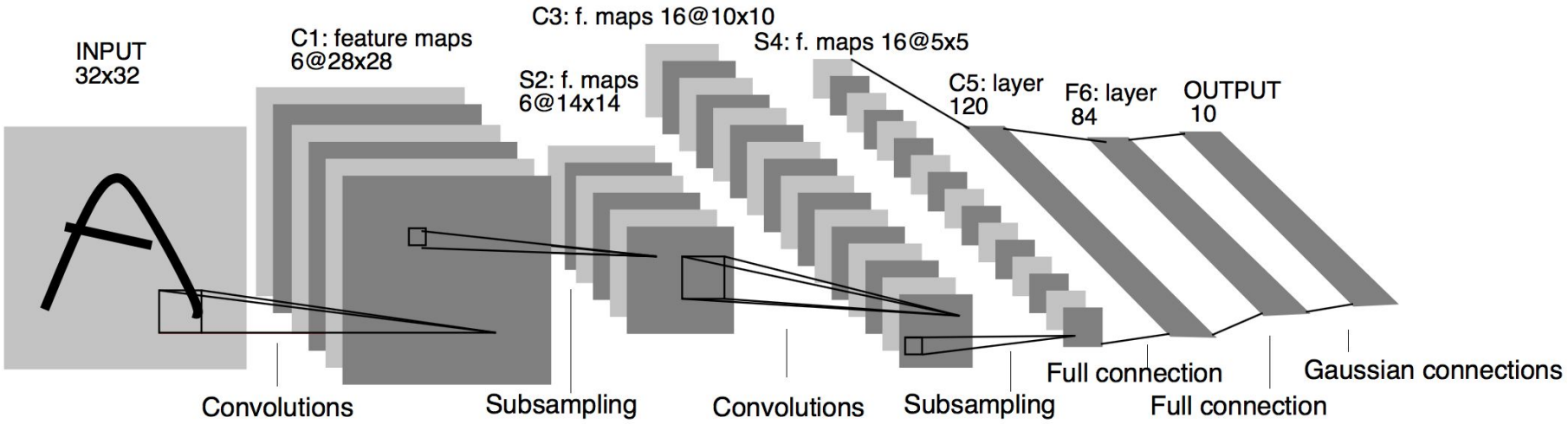
A simple CNN to tackle the Fashion-MNIST

```
model = keras.models.Sequential([
    keras.layers.Conv2D(64, 7, activation="relu", padding="same",
                        input_shape=[28, 28, 1]),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation="softmax")
])
```

This CNN achieves 92% accuracy (not state of the art, but pretty good)

LeNet-5

Landmark paper: LeCun et al., Gradient-based learning applied to document recognition, 1998



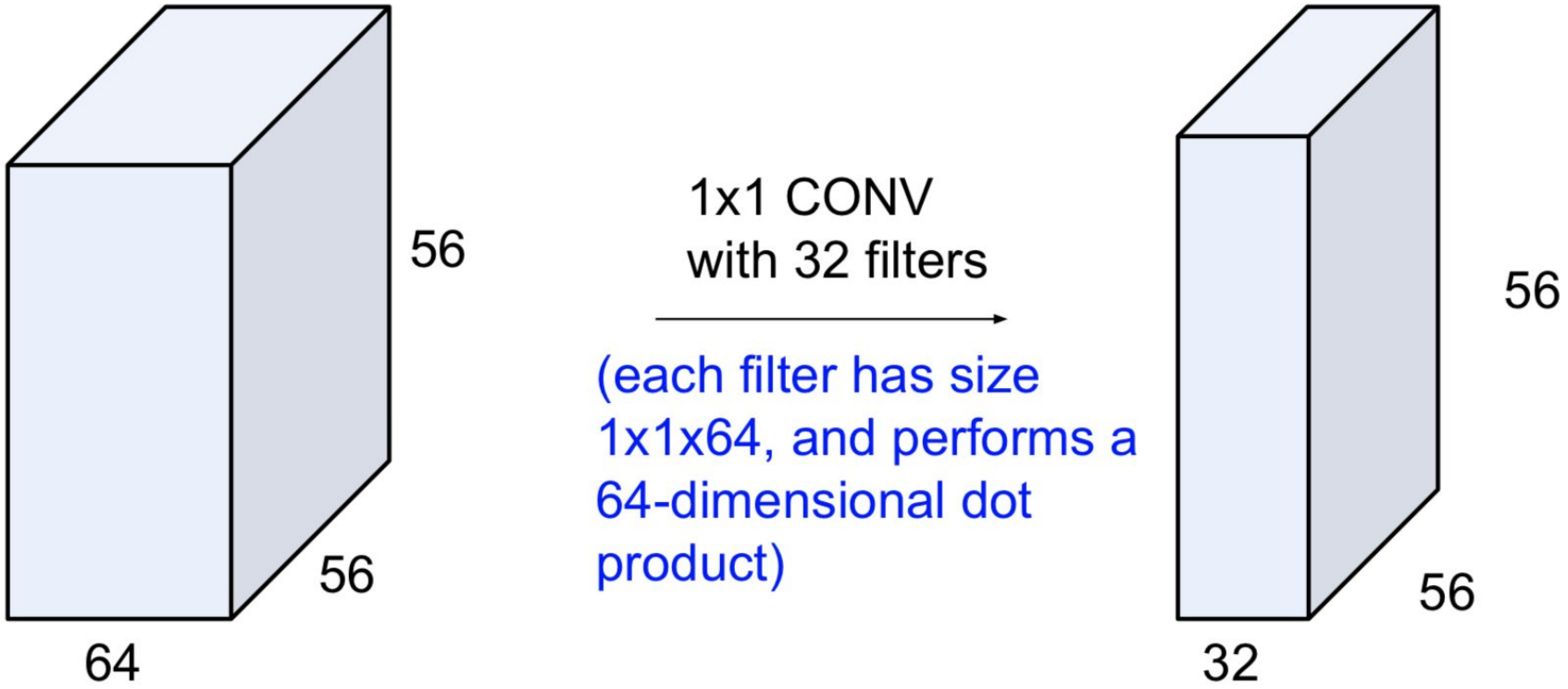
LeNet-5

Most widely know CNN architecture, created by Yann LeCun in 1998

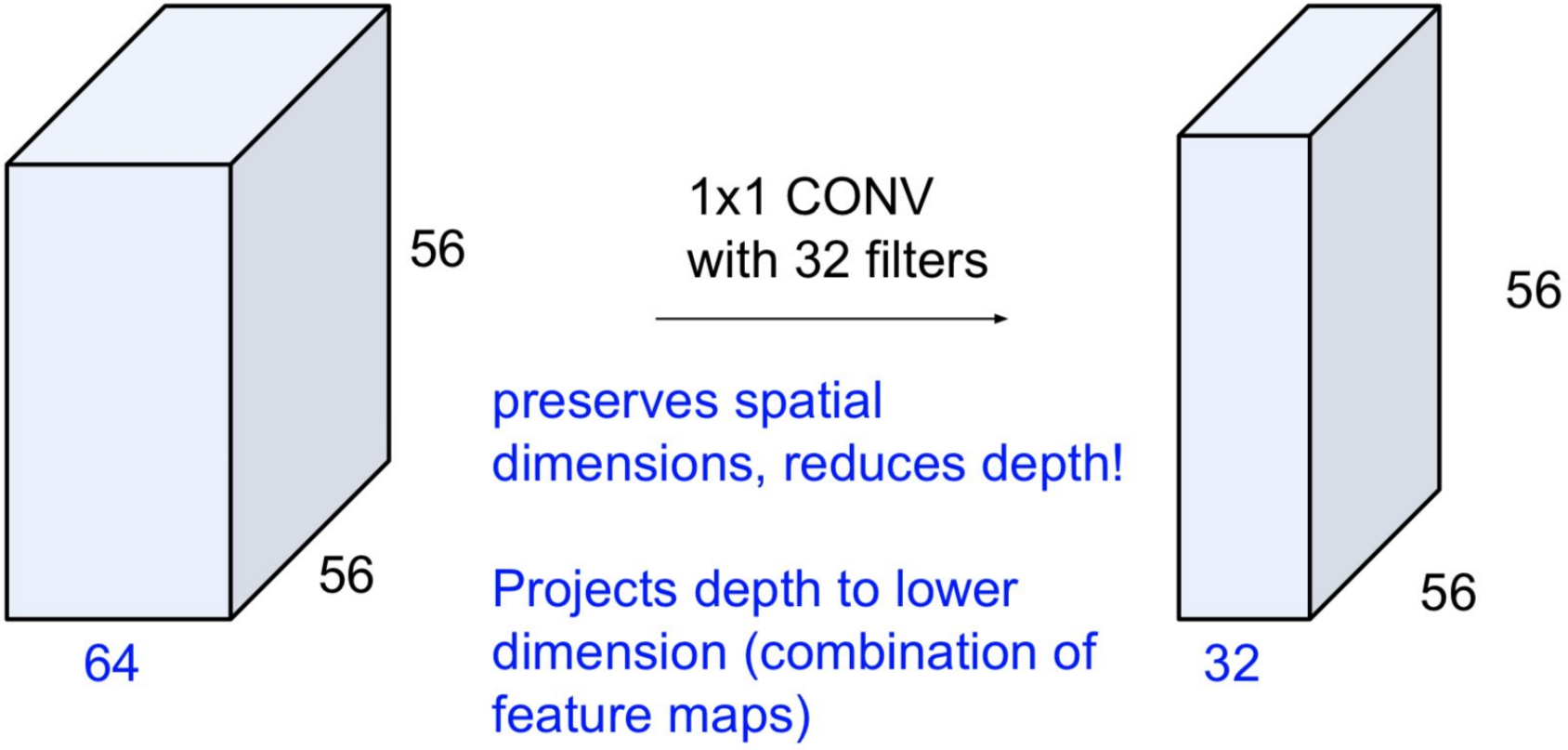
Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully Connected	–	10	–	–	RBF
F6	Fully Connected	–	84	–	–	tanh
C5	Convolution	120	1 × 1	5 × 5	1	tanh
S4	Avg Pooling	16	5 × 5	2 × 2	2	tanh
C3	Convolution	16	10 × 10	5 × 5	1	tanh
S2	Avg Pooling	6	14 × 14	2 × 2	2	tanh
C1	Convolution	6	28 × 28	5 × 5	1	tanh
In	Input	1	32 × 32	–	–	–

Demo on MNIST

Note on 1x1 convolution



1x1 convolution



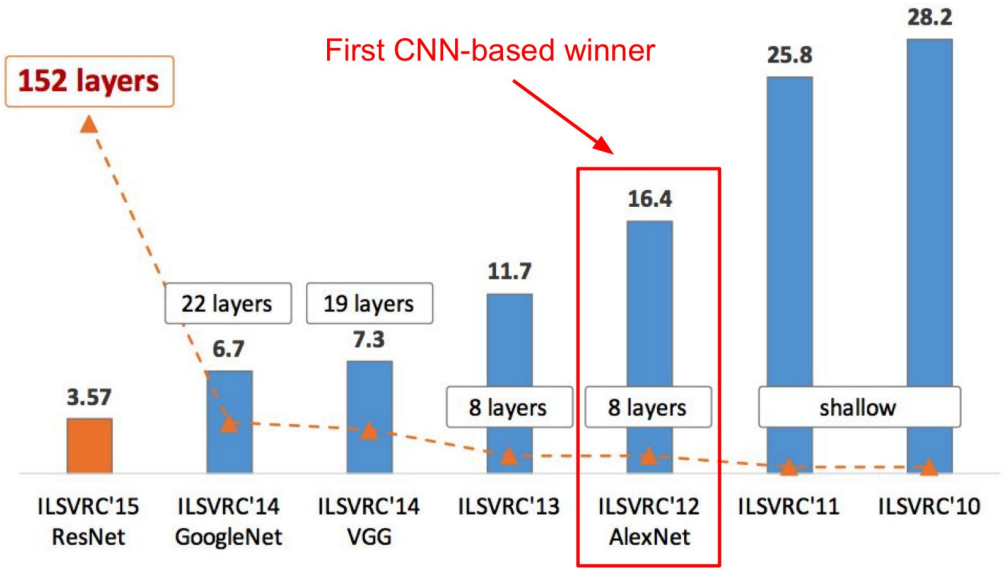
ImageNet Challenge

- Over the years, variants of the CNN architecture have been developed, leading to amazing advances in the field.
- A good measure of this progress is the top-5 error in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC).
- The ImageNet contain large number of images (14.2 million) of over 1000 classes, some of which are really subtle (try distinguishing 120 dog breeds)
- The top-5 error rate is the number of test images for which the system's top 5 predictions did not include the correct answers.

Winners of the ImageNet Challenges

Looking at the evolution of the ImageNet winning entries is a good way to understand how CNNs work.

- AlexNet (2012 winner)
- GoogLeNet (2014 winner)
- ResNet (2015 winner)



AlexNet

Developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton

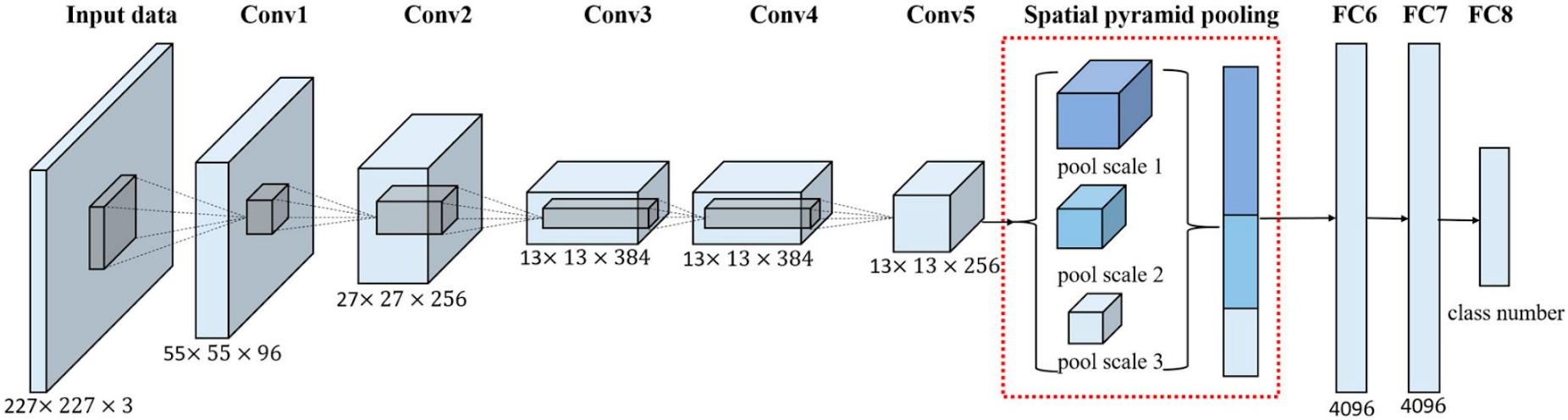
17% top-5 error rate of ImageNet Challenge in 2012

Much larger and deeper then LeNet-5

Stack convolutional layers directly on top of each other

Layer	Type	Maps	Size	Kernel size	Stride	Padding	Activation
Out	Fully Connected	–	1,000	–	–	–	Softmax
F9	Fully Connected	–	4,096	–	–	–	ReLU
F8	Fully Connected	–	4,096	–	–	–	ReLU
C7	Convolution	256	13 × 13	3 × 3	1	SAME	ReLU
C6	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
C5	Convolution	384	13 × 13	3 × 3	1	SAME	ReLU
S4	Max Pooling	256	13 × 13	3 × 3	2	VALID	–
C3	Convolution	256	27 × 27	5 × 5	1	SAME	ReLU
S2	Max Pooling	96	27 × 27	3 × 3	2	VALID	–
C1	Convolution	96	55 × 55	11 × 11	4	SAME	ReLU
In	Input	3 (RGB)	224 × 224	–	–	–	–

AlexNet Architecture



- First use of ReLU
- Heavy data Augmentation
- Dropout at 0.5

- Batchsize 128
- SGD Momentum 0.9
- Learning rate 0.01, reduced by 10 manually when val accuracy plateaus

GoogLeNet

Developed by Christian Szegedy et al. from Google Research

Winner of the ImageNet challenge with below 7% top-5 error rate

10 times fewer parameters then AlexNet (**6M** instead of 60M)

Much deeper than previous CNNs by introducing **inception (network within a network)** module

Leo's Inception moment at the Oscar 2016

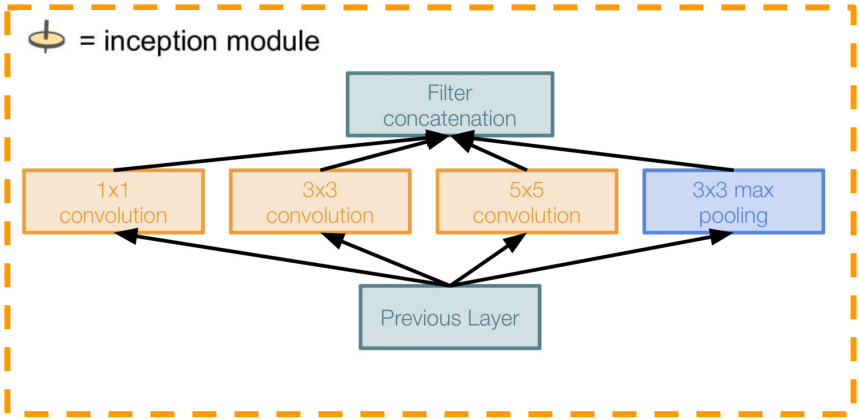
GoogLeNet Inception Module

Design a good local network topology (**network within a network**), stack them on top of each other, and concatenate all filter outputs together depth-wise

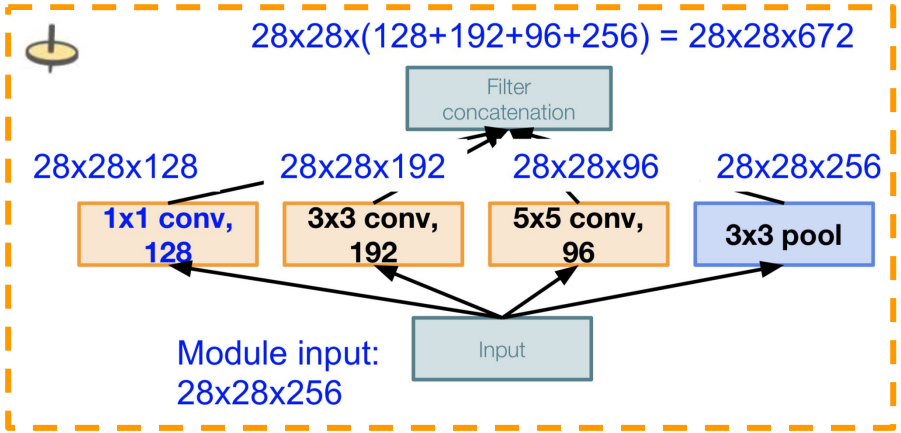
Apply parallel filtering operations on the input from previous layer

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

Concatenate all filter outputs together depthwise



Naive Inception Computational Complexity



Conv Ops:

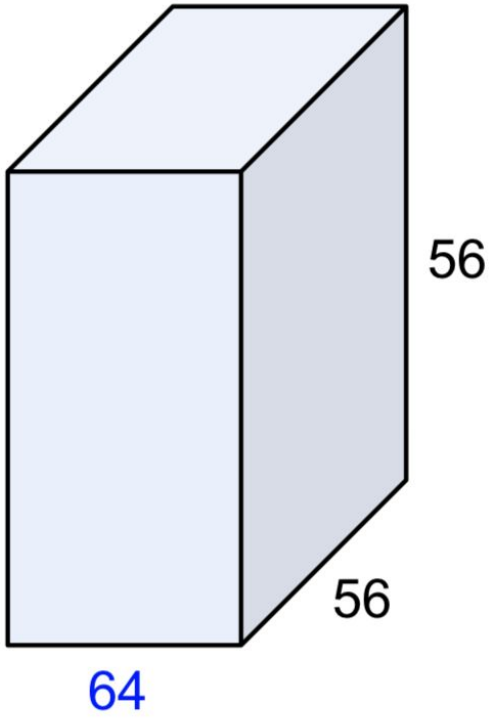
- [1x1 conv, 128] 28x28x128x1x1x256
- [3x3 conv, 192] 28x28x192x3x3x256
- [5x5 conv, 96] 28x28x96x5x5x256
- Total: **854M ops**

Very expensive to compute

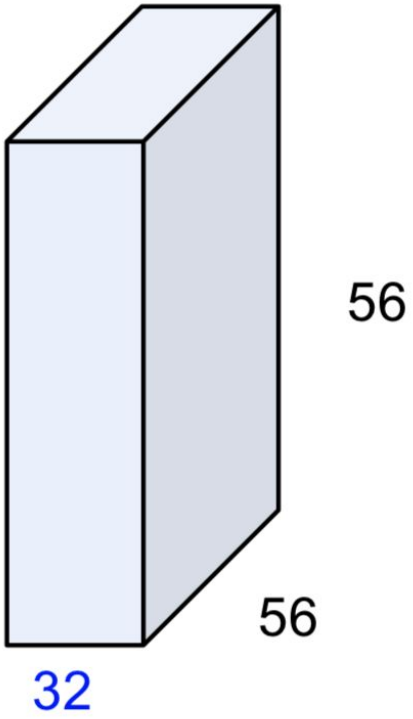
Pooling layer preserves feature depth, which means the total depth after concatenation can only **grow** at every layer!

How to reduce feature depth growing at every layer?

Solution: use 1x1 CONV filters



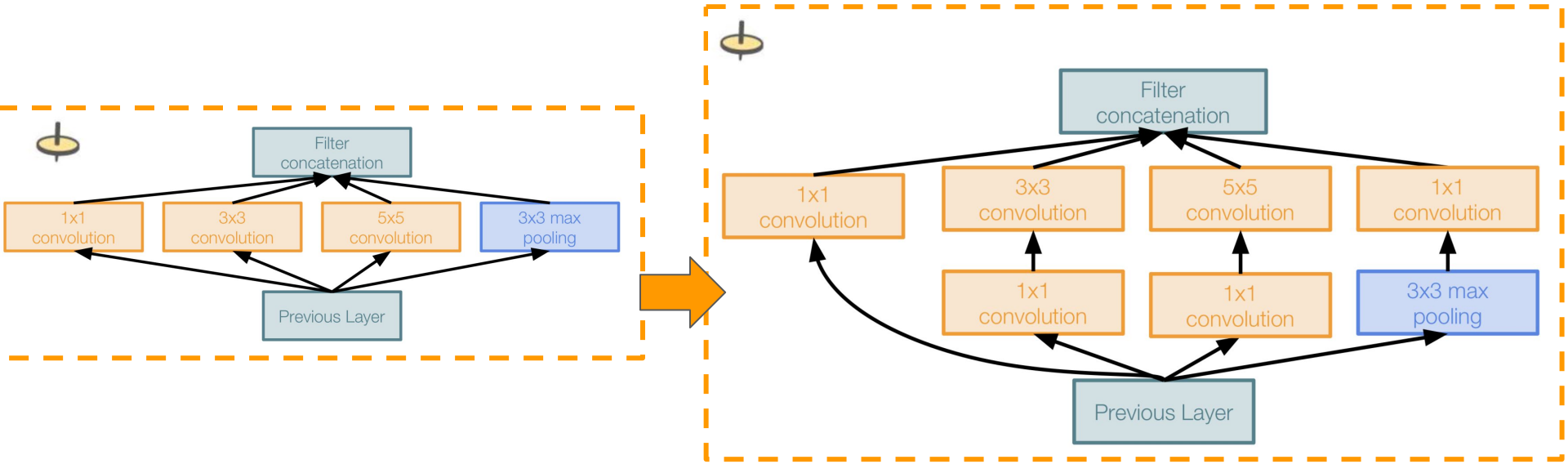
1x1 CONV
with 32 filters



preserves spatial dimensions, reduces depth!

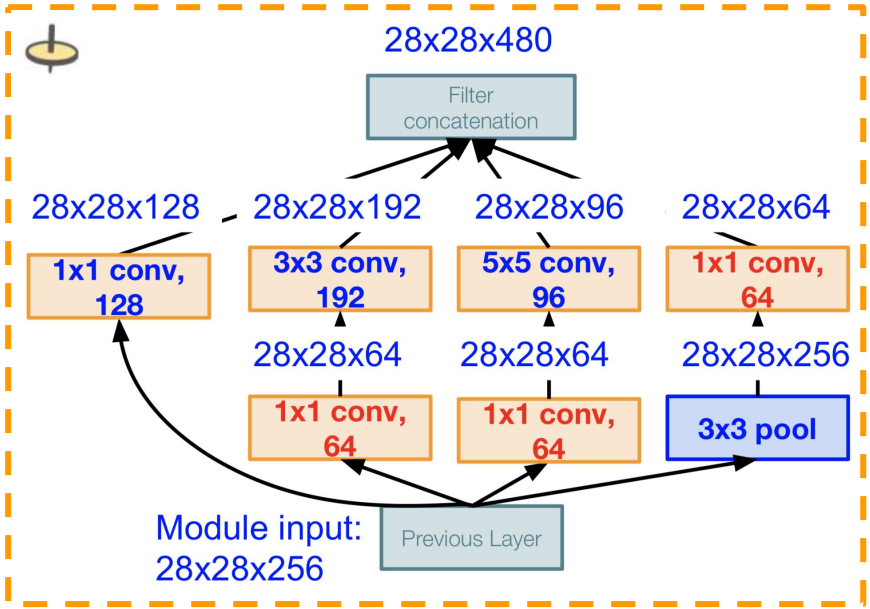
Projects depth to lower dimension (combination of feature maps)

Inception Modele with Dimensionality Reduction



Solution: Use **1x1 convolutions** to reduce feature depth

Inception Module Computational Complexity



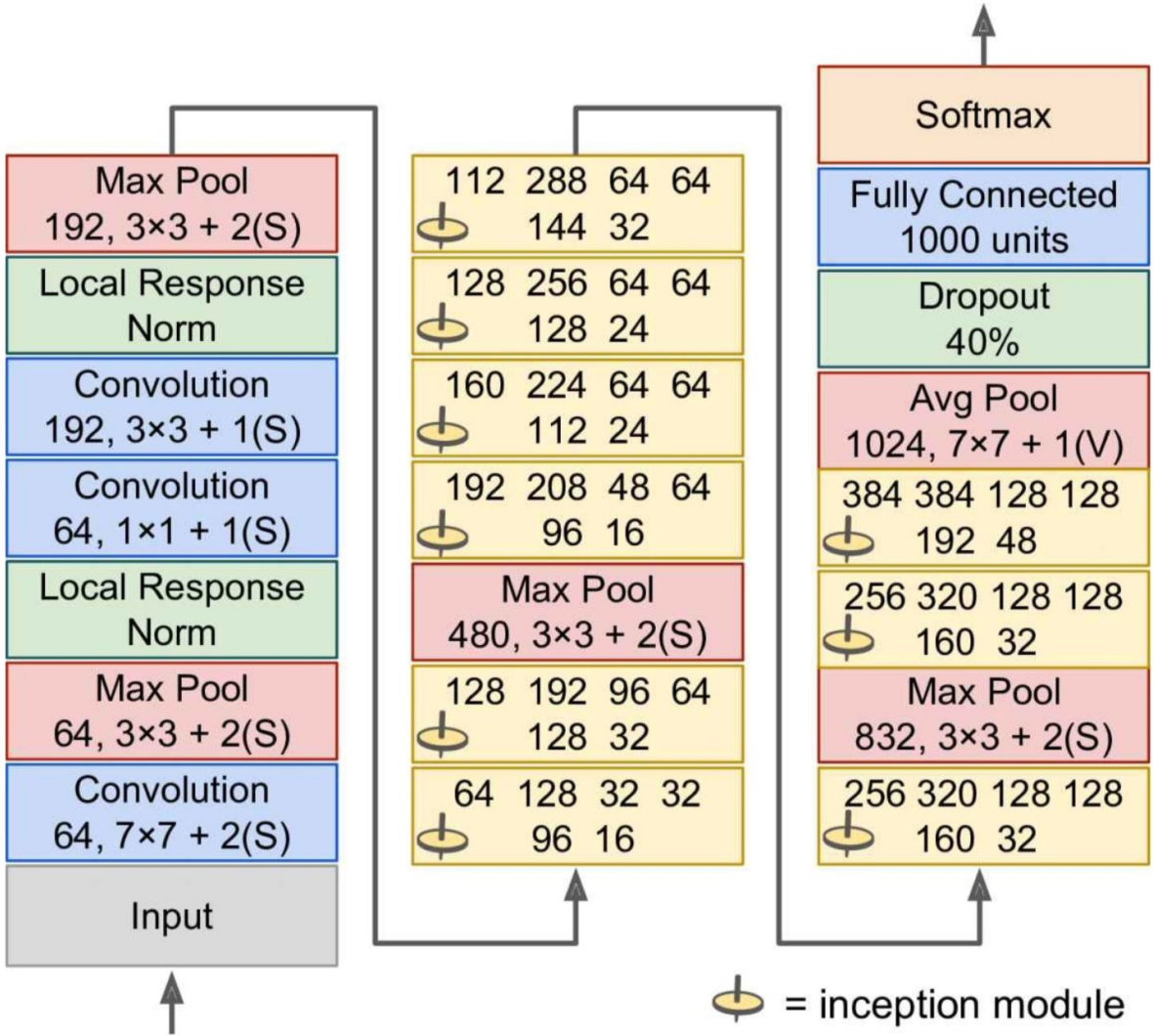
Conv Ops:

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

Comparing to 854M ops for naive version.

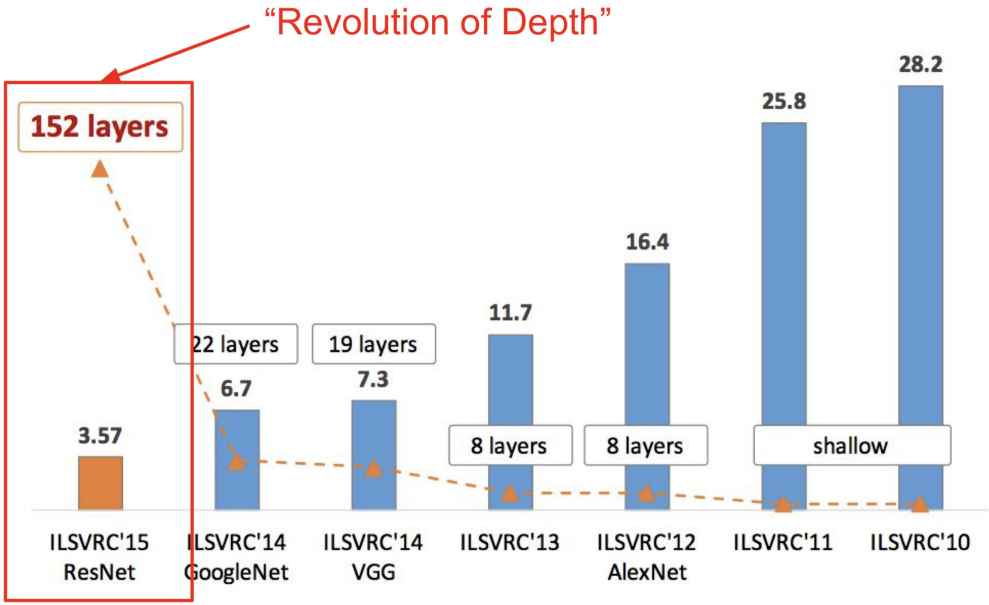
GoogLeNet

- Deeper network (22 layers)
- Stack efficient Inception modules on top of each other.
- No FC layers (except outputs)
- 10x less params than AlexNet
- Some variants were proposed after including Inception-v3 and Inception-v4.

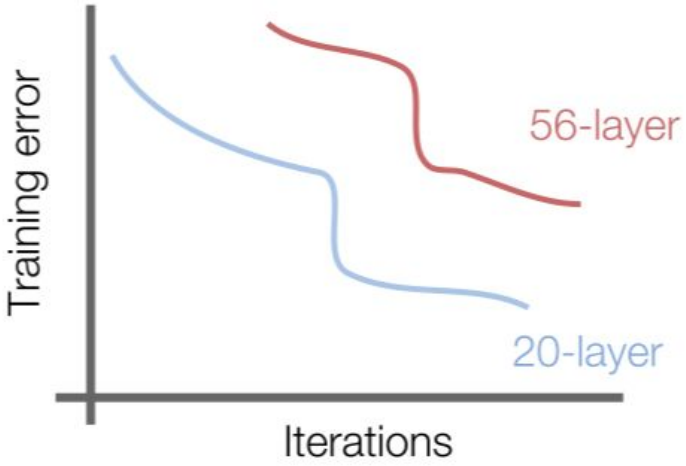


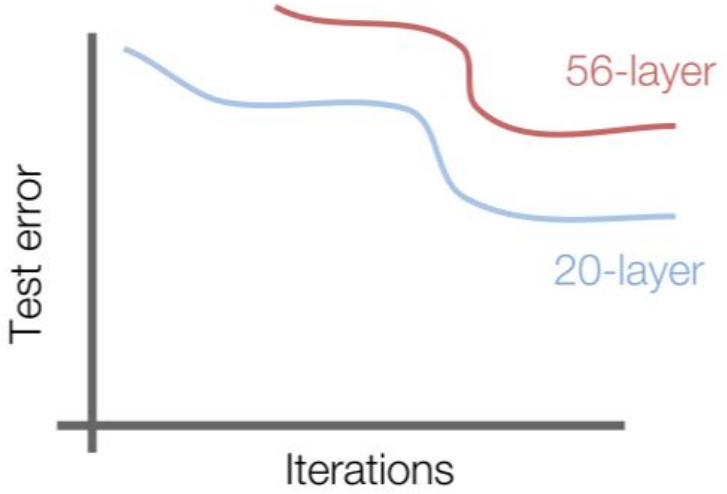
ResNet

- Developed by Kaiming He et al., extremely deep CNN with **152 layers**
- Winner of ImageNet Challenge in 2015 with **3.6%** top-5 error rate
- 2-3 weeks of training time on a 8 GPU machine
- Does more layers means better?



More layers equals better?





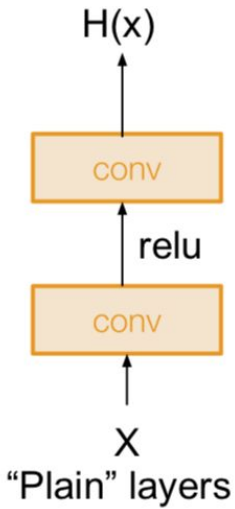
He *et al* 2015 shows that 56-layer model performs worse that 20-layer model.

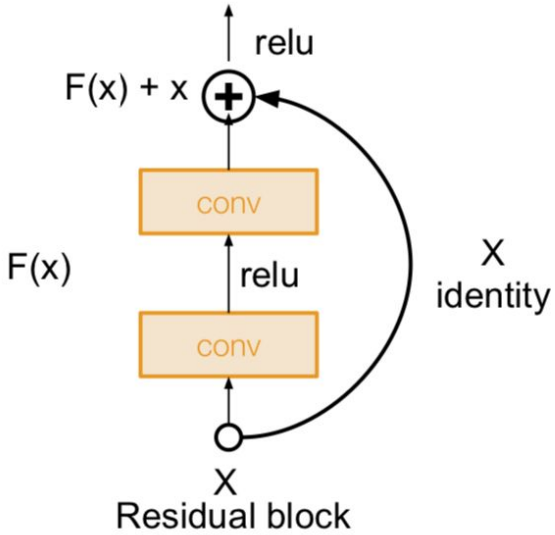
Does that cause by overfitting?

It is the optimization -- deeper network is harder to optimize!

ResNet

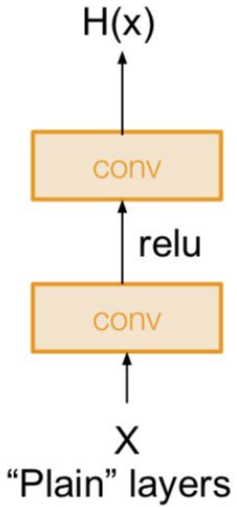
- The deeper model should be able to perform at least as well as the shallow one.
- Usage of **skip (shortcut)** connections: Copy a solution from the shallower model and setting additional layers to identity mapping.
- Network will be forced to model F(x) = H(x) -x rather than H(x) → **residual learning**

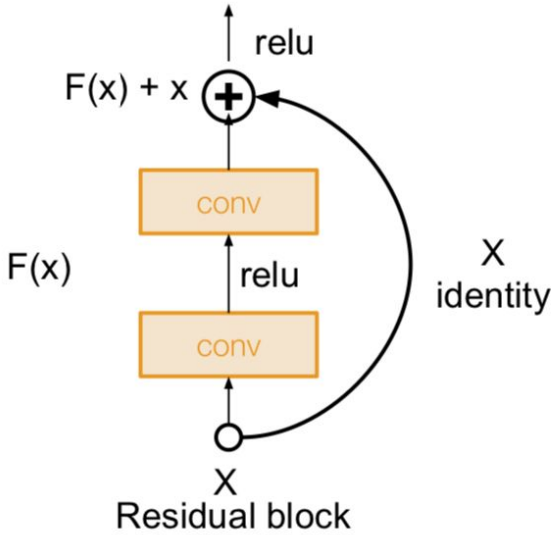




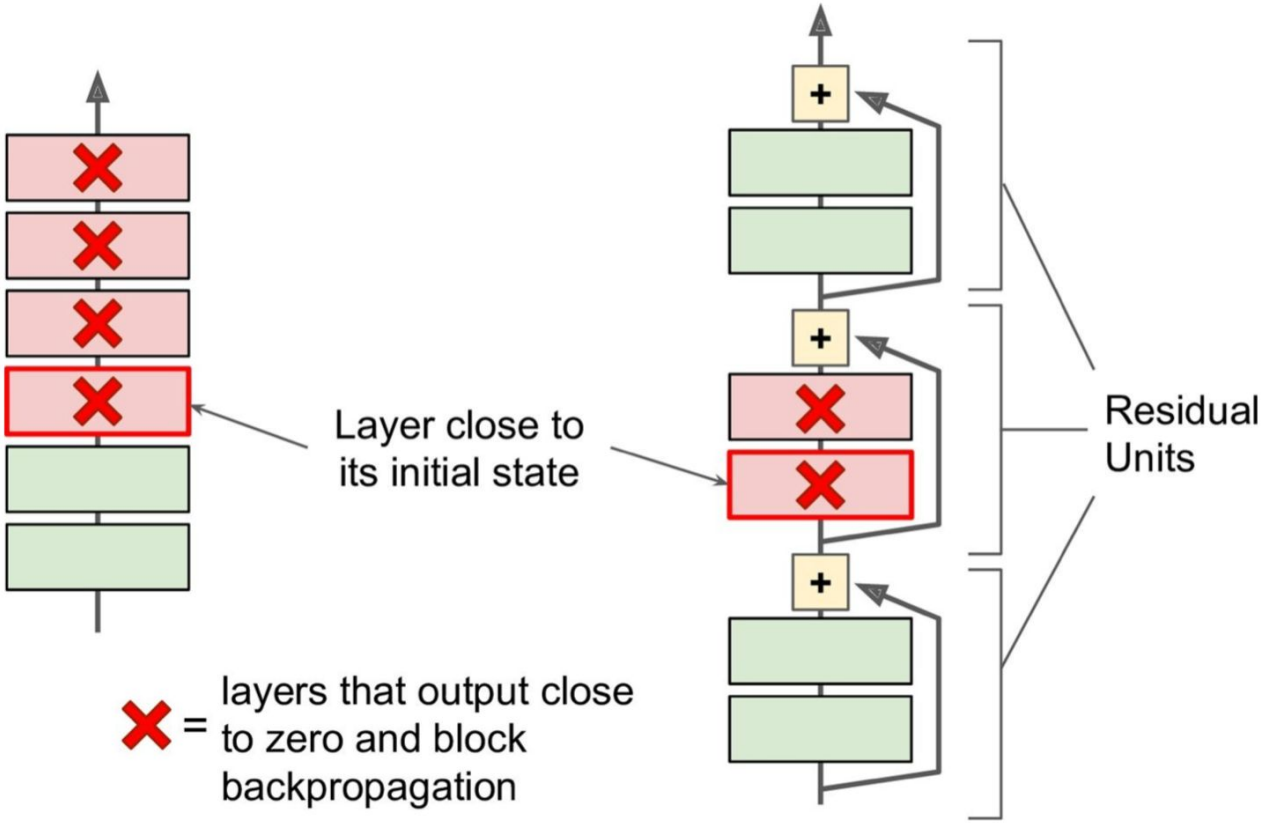
ResNet Config

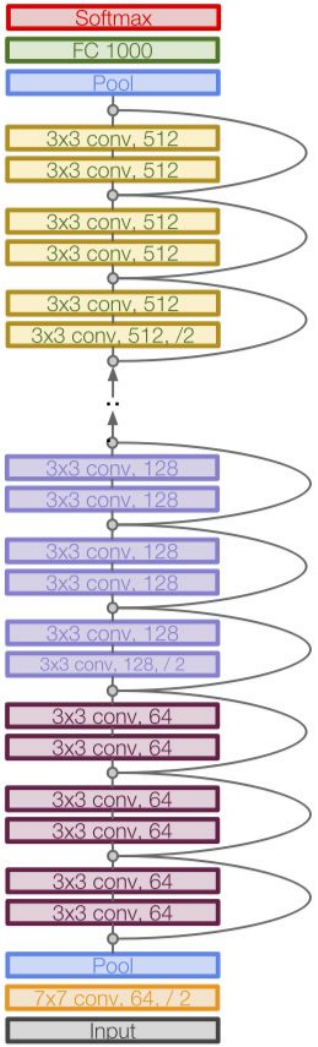
- Batch Normalization after every CONV layer
- He Initialization
- SGD + Momentum (0.9)
- Learning rate (0.1), divided by 10 when validation error plateaus
- Mini-batch size 256
- No dropout used



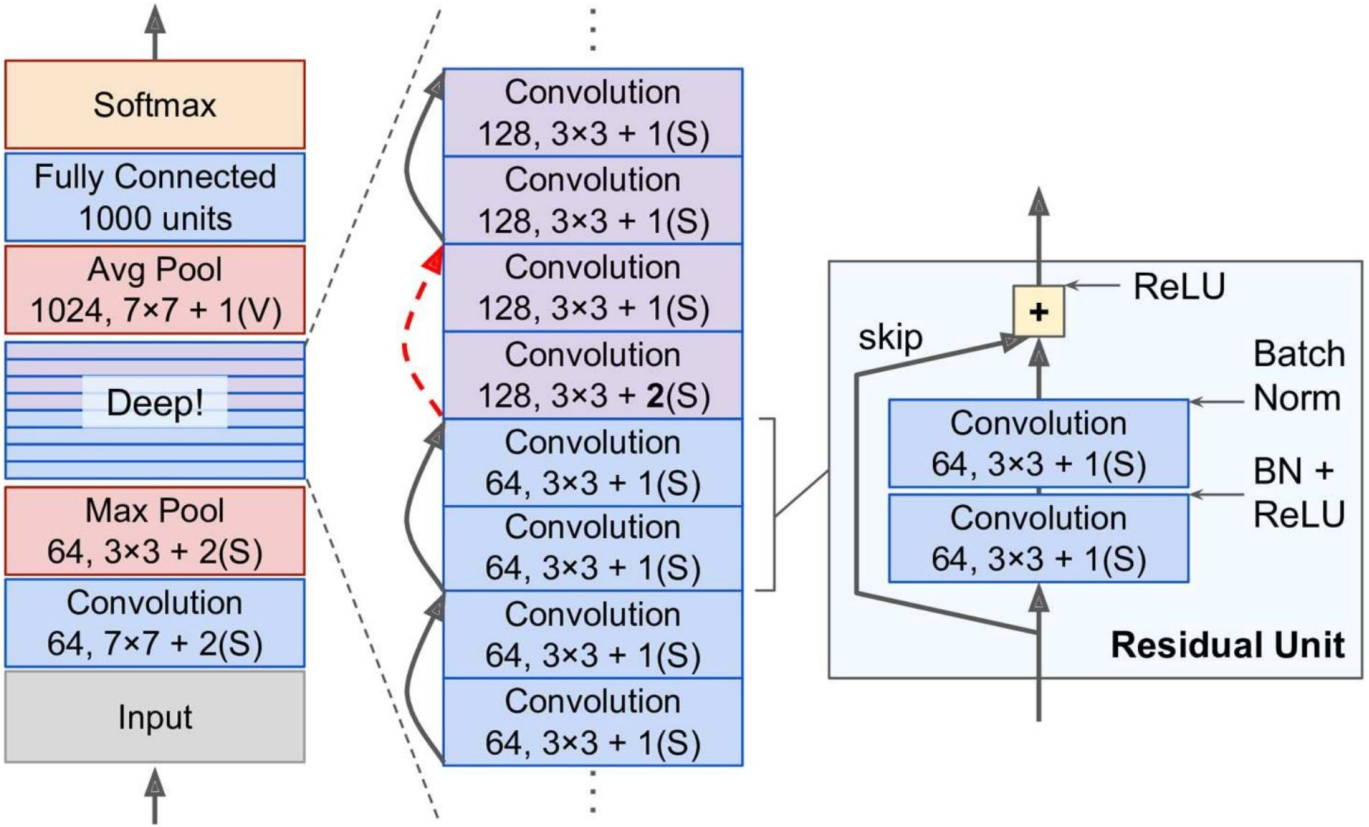


Regular DNN vs. ResNet

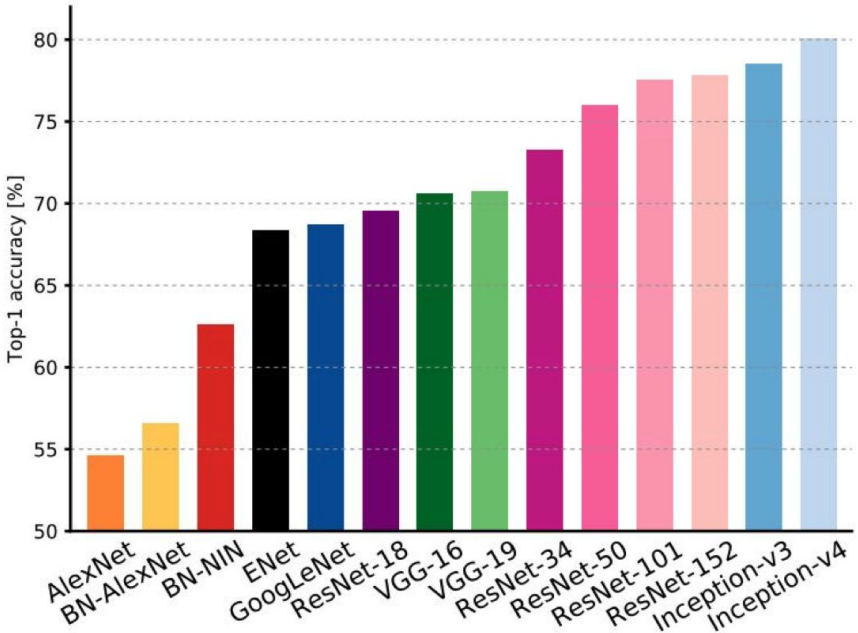


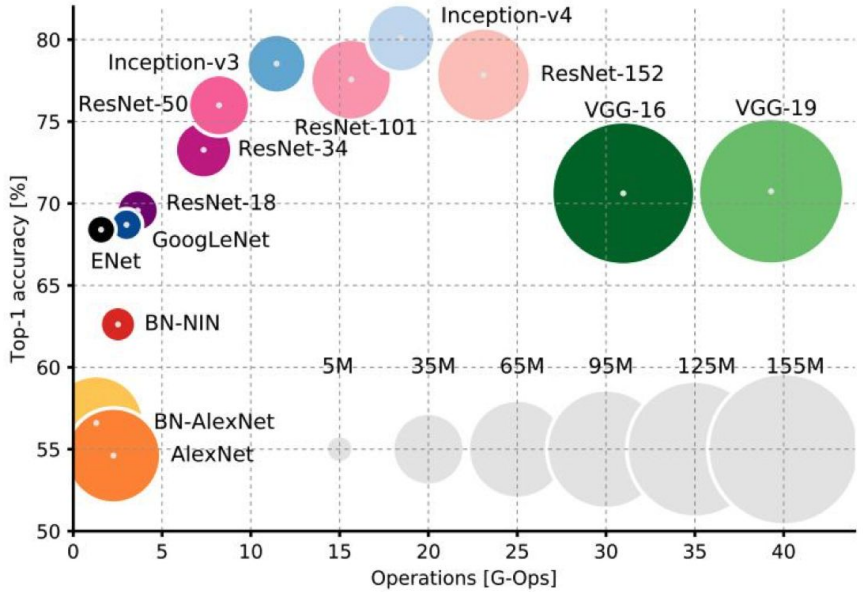


ResNet Architecture



Complexity vs. Accuracy





An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Xception

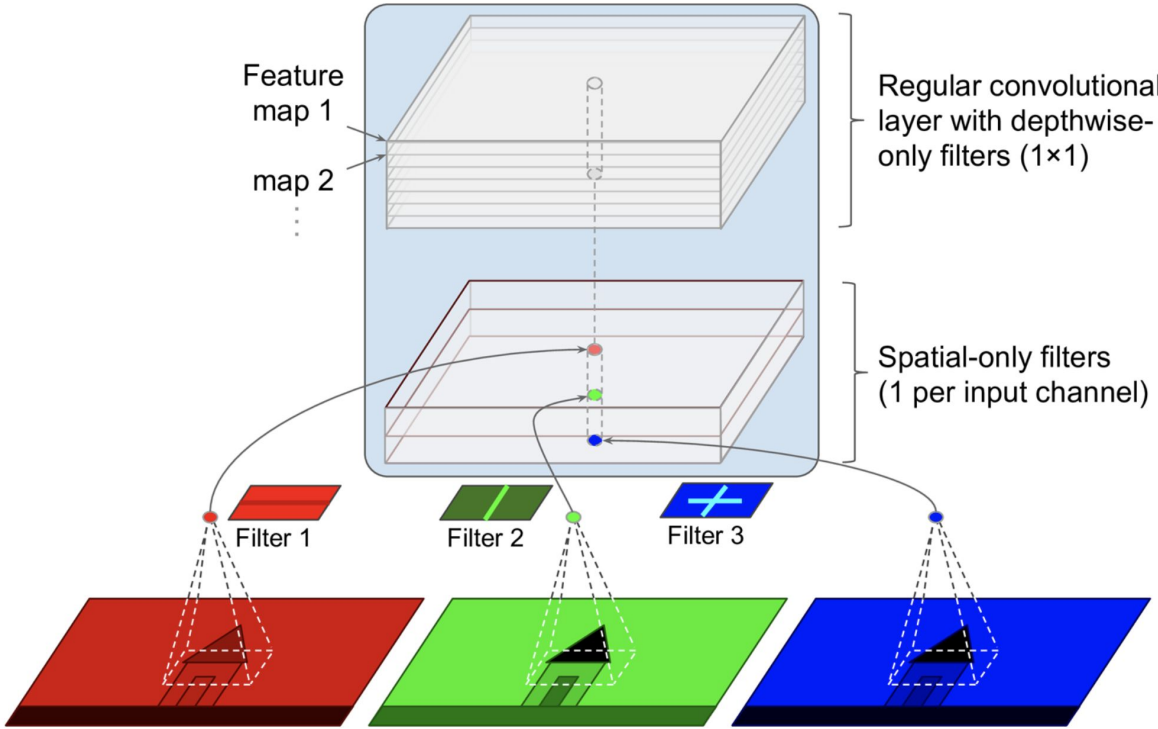
Another variant of GoogLeNet architecture is Xception (Extreme Inception)

Proposed in 2016 by Francois Chollet (the author of Keras)

Merges the ideas of GoogLeNet and ResNet, but replaces the inception modules with “depthwise separable convolution layer”

Make a strong assumption that spatial patterns and cross-channel patterns can be modeled separately.

Separate convolutional layers



It uses fewer parameters, less memory, and few computations than regular conv layer, and it performs better in general.

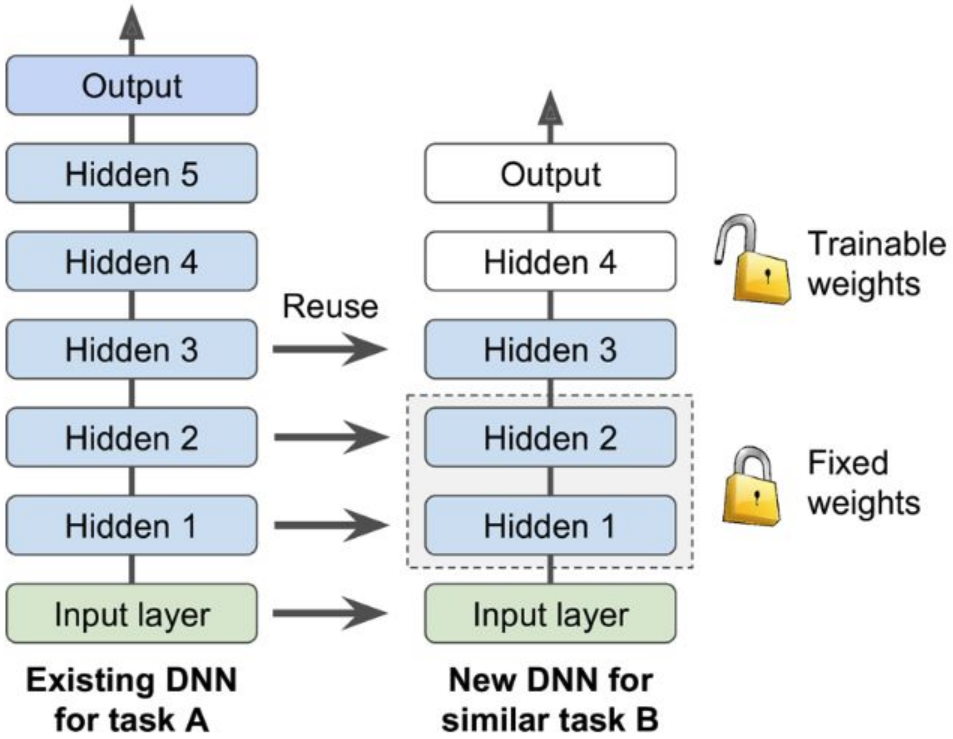
Transfer Learning

Transfer Learning with Keras

It is generally not a good idea to train a very large DNN from scratch, instead try to use an existing network that accomplishes a similar task

Not only speed up training, but also requires significantly less training data

New Input layer must have the same size. Hidden layers can be kept. New Output layer must be replaced and retrained for the new task



Using Pre-trained Models from Keras

In general, you won't have to implement models like GoogLeNet, or ResNet

Load models from keras.applications

```
model = keras.applications.resnet50.ResNet50(weights="imagenet")
```

To use it, you need to ensure your images have the right size since ResNet-50 model expects 224x224 pixel image.

```
images_resized = tf.image.resize(images, [224, 224])
```

```
inputs = keras.applications.resnet50.preprocess_input(images_resized * 255)
```

Train using Transfer Learning

To build an image classifier without lots of training data, it's often a good idea to reuse the lower layers of a pre-trained model. We use Xception model pre-trained on ImageNet. We exclude the top layers of the network and add our own global average pooling layer, followed by the dense output layer with softmax.

After that, you can use pre-trained model to make predictions:

```
base_model = keras.applications.xception.Xception(weights="imagenet",
                                                  include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
output = keras.layers.Dense(n_classes, activation="softmax")(avg)
model = keras.Model(inputs=base_model.input, outputs=output)
```

Training the added top layers

It’s a good idea to freeze the weights of pretrained layers, in order to train the top layers. (Tip: remember to switch hardware acceleration to GPU on Colab)

```
for layer in base_model.layers:
    layer.trainable = False

optimizer = keras.optimizers.SGD(lr=0.2, momentum=0.9, decay=0.01)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set,
                    steps_per_epoch=int(0.75 * dataset_size / batch_size),
                    validation_data=valid_set,
                    validation_steps=int(0.15 * dataset_size / batch_size),
                    epochs=5)
```

Training the entire network

Now that the top layers in well trained, we are ready to unfreeze all the layers and continue training with a much lower learning rate so that you don’t damage the pre-trained weights

```
for layer in base_model.layers:
    layer.trainable = True

optimizer = keras.optimizers.SGD(learning_rate=0.01, momentum=0.9,
                                 nesterov=True, decay=0.001)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set,
                    steps_per_epoch=int(0.75 * dataset_size / batch_size),
                    validation_data=valid_set,
                    validation_steps=int(0.15 * dataset_size / batch_size),
                    epochs=40)
```

Summary

The field is moving rapidly, with all sort of architectures popping out every year

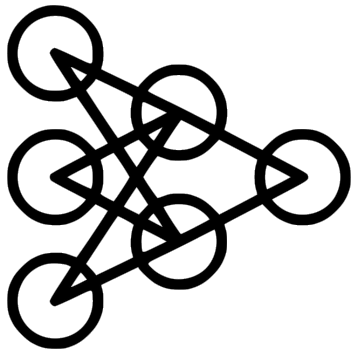
AlexNet, GoogLeNet, ResNet and their variations are all widely used

One clear trend that CNNs keep getting deeper and lighter, requiring less parameters (ResNet is both simplest and most powerful)

2016 ImageNet winner, GBD-Net, uses ensemble learning to train combinations of the previous models to achieve < 3% top-5 error

Today: Learning Objectives

- ✓ Look at the applications of Convolutional Neural Networks (CNNs)
- ✓ Study the building blocks of CNNs: convolutional and pooling layers
- ✓ Explore winning CNN architectures: LeNet-5, AlexNet, GoogLeNet, ResNet
- ✓ Training with transfer learning



Acknowledgements

All materials and graphics are intended to use solely for educational purposes.

Some of materials from the slides are adopted from **Prof. Fei-Fei Li's group** at Stanford. Big thanks to her group!

Many graphics use in the toy examples are modified from **Luis Serrano** at Udacity and **Brandon Rohrer** at End-To-End Machine Learning.

Bonus Slides

Deep Learning Frameworks

Caffe
(UC Berkeley)

Caffe2
(Facebook)

Torch
(NYU / Facebook)

PyTorch
(Facebook)

Theano
(U Montreal)

TensorFlow
(Google)

CNTK
(Microsoft)

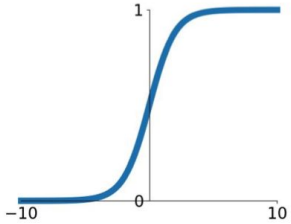
MXNet
(Amazon)

Developed by U Washington, CMU, MIT, Hong Kong U, etc but main framework of choice at AWS

Activation Functions

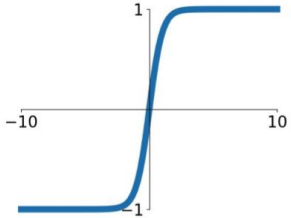
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



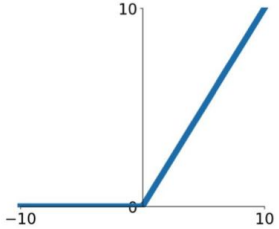
tanh

$$\tanh(x)$$



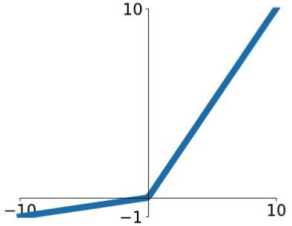
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

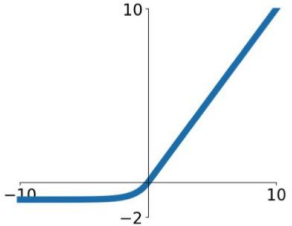


Maxout

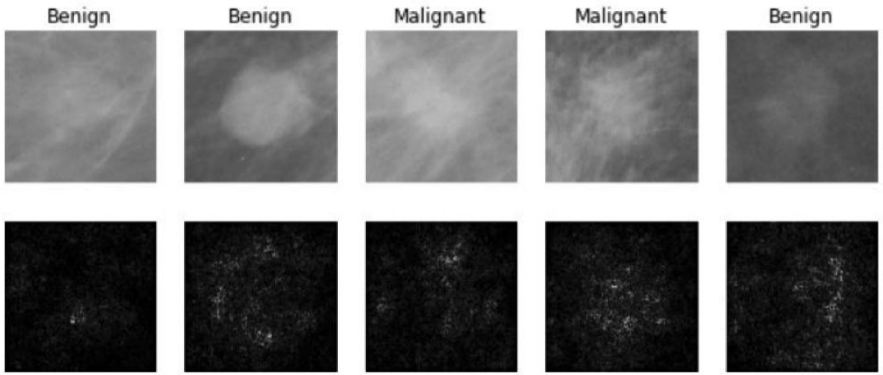
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



CNNs are everywhere these days

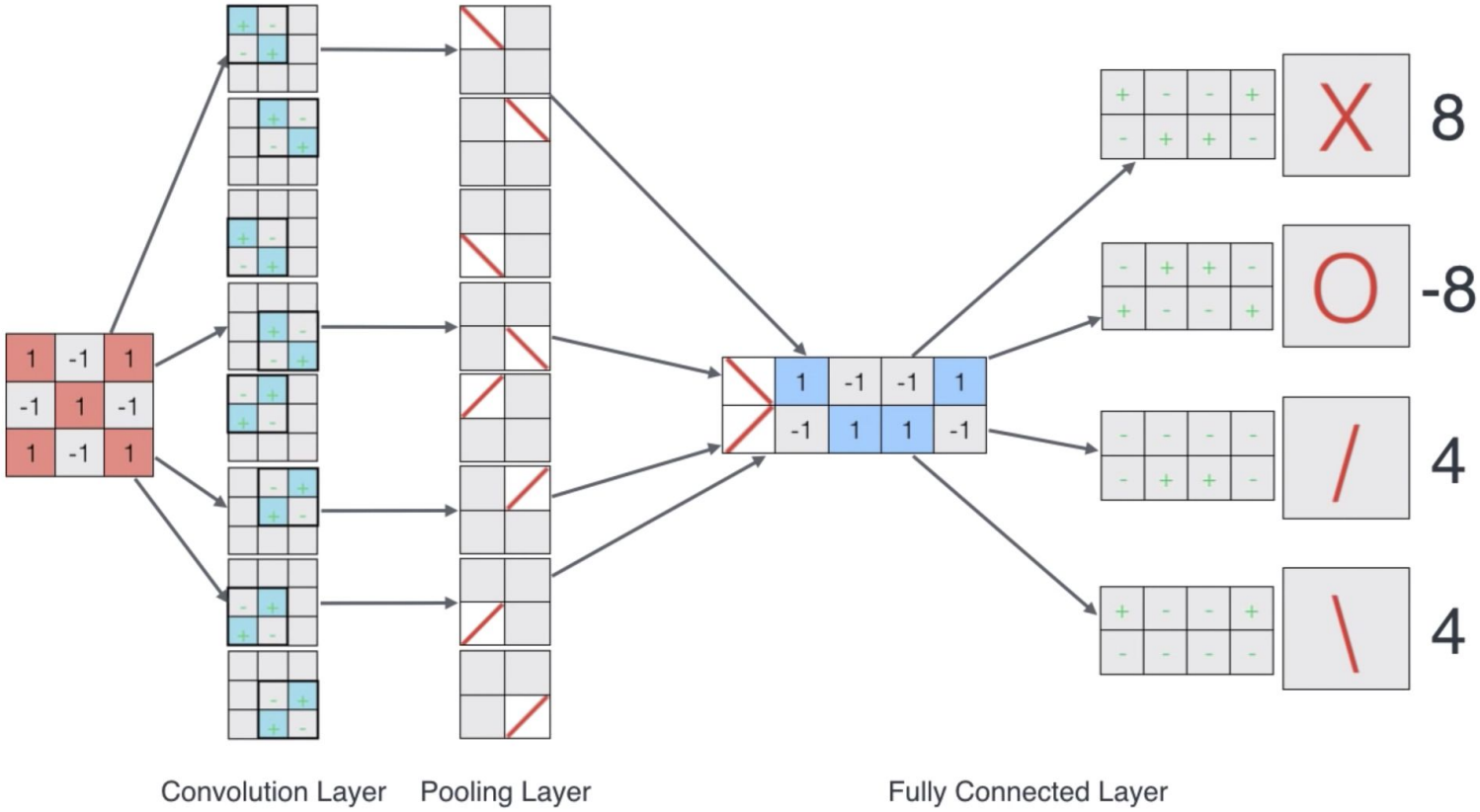


[Levy et al. 2016]

[Dieleman et al. 2014]

[Sermanet et al. 2011]
[Ciresan et al.]

Putting it together

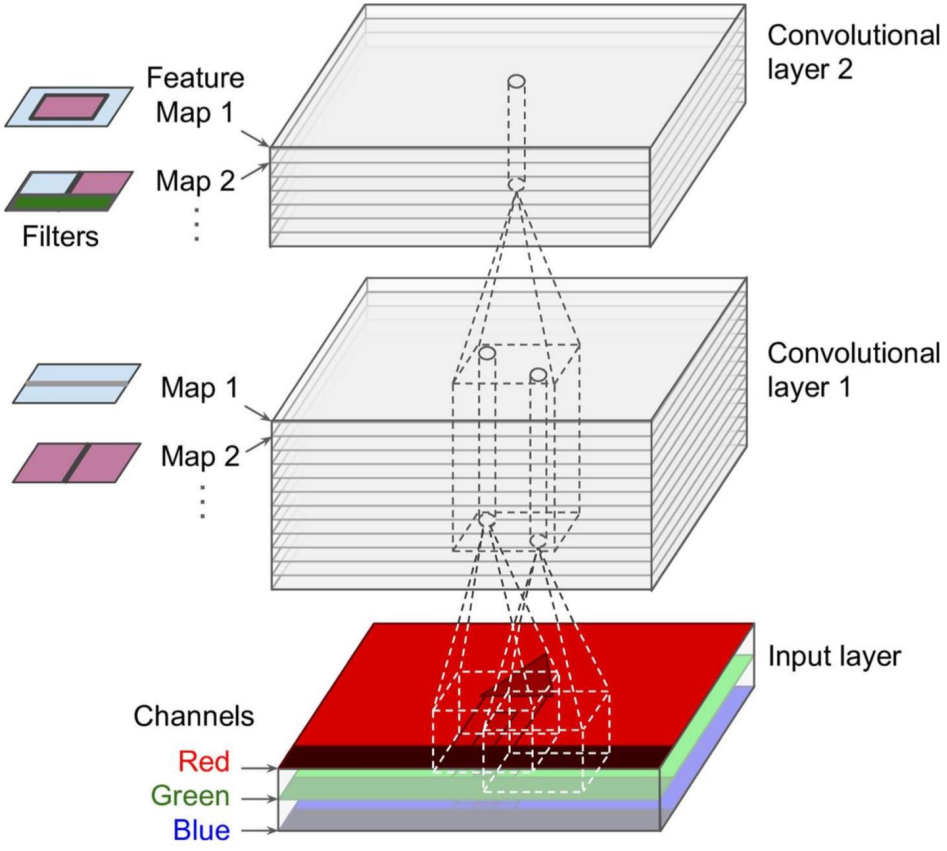


Stacking Multiple Feature Maps

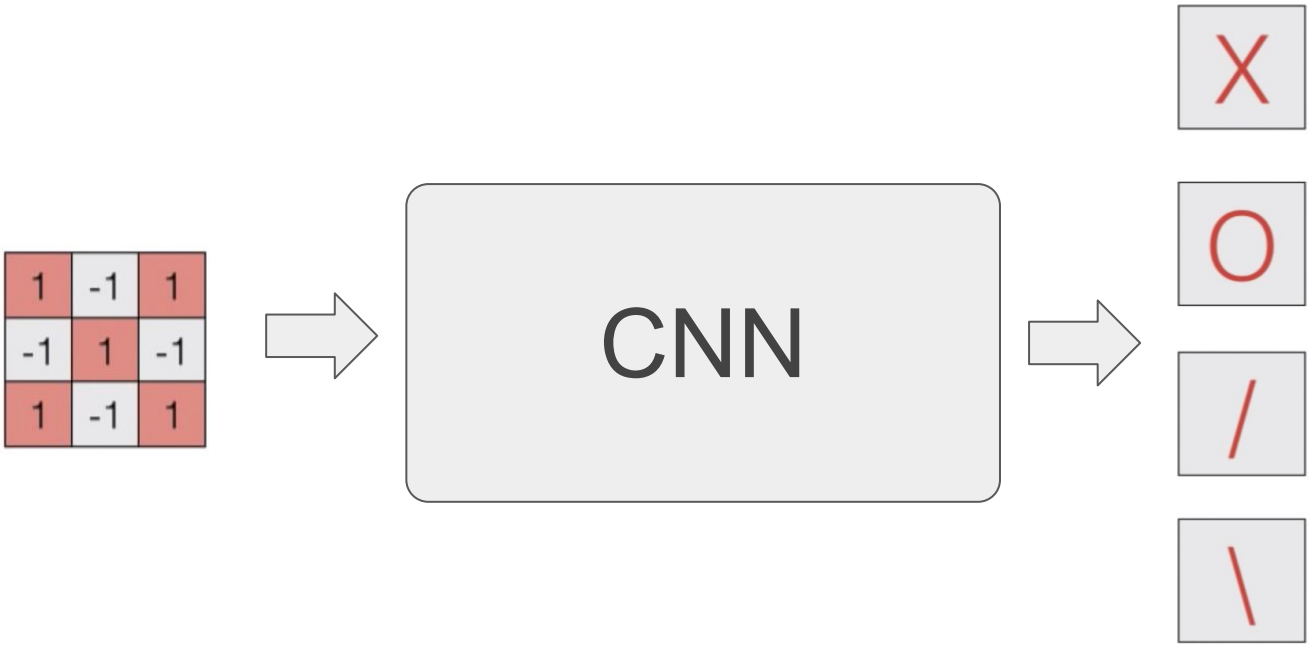
Each convolutional layer is composed several feature maps of equal sizes → more accurate to represent in 3D.

All neurons within one feature map share the same parameters, but different feature maps may have different parameters

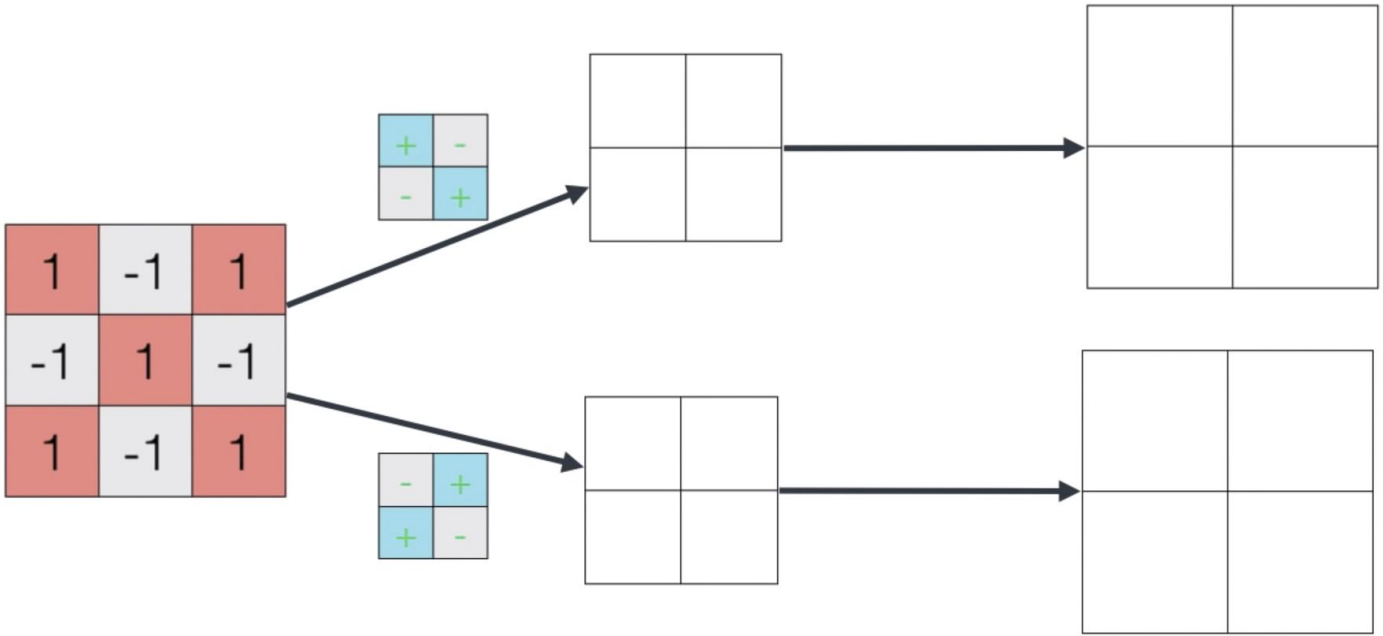
In short, a convolutional layer simultaneously applied multiple filters to its input, making it capable of detecting multiple features anywhere in its inputs.



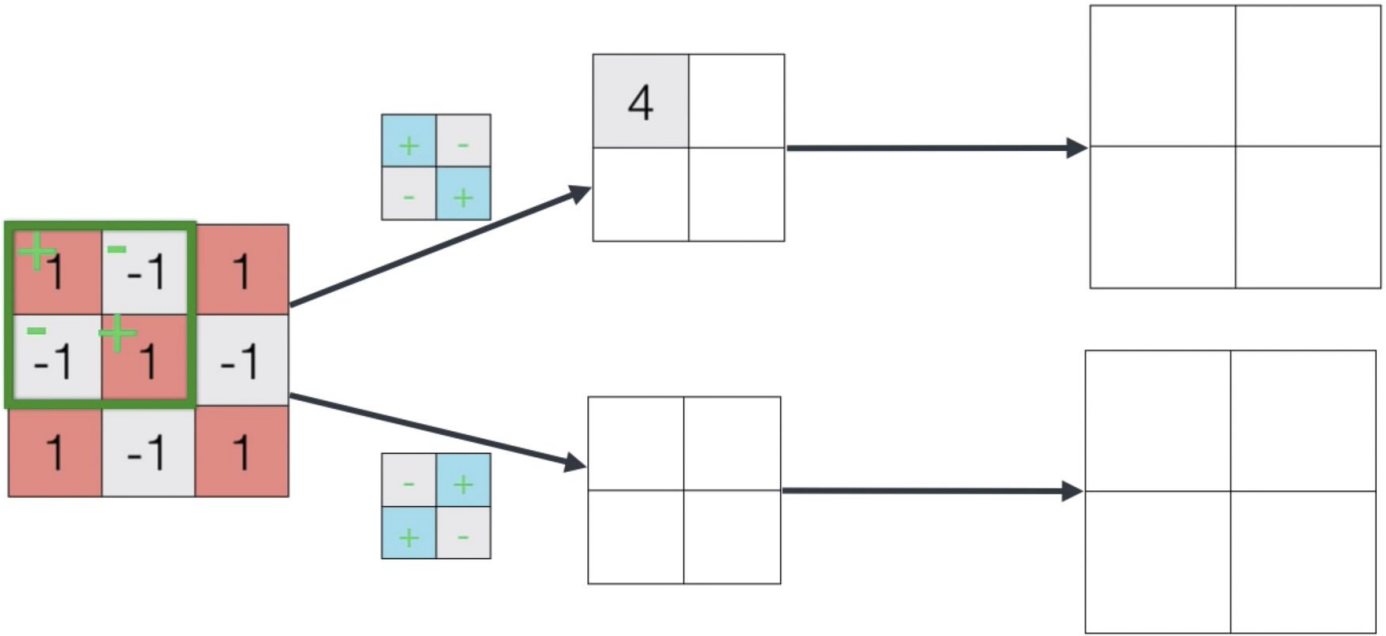
A simple example of classifying X,O, / , and \



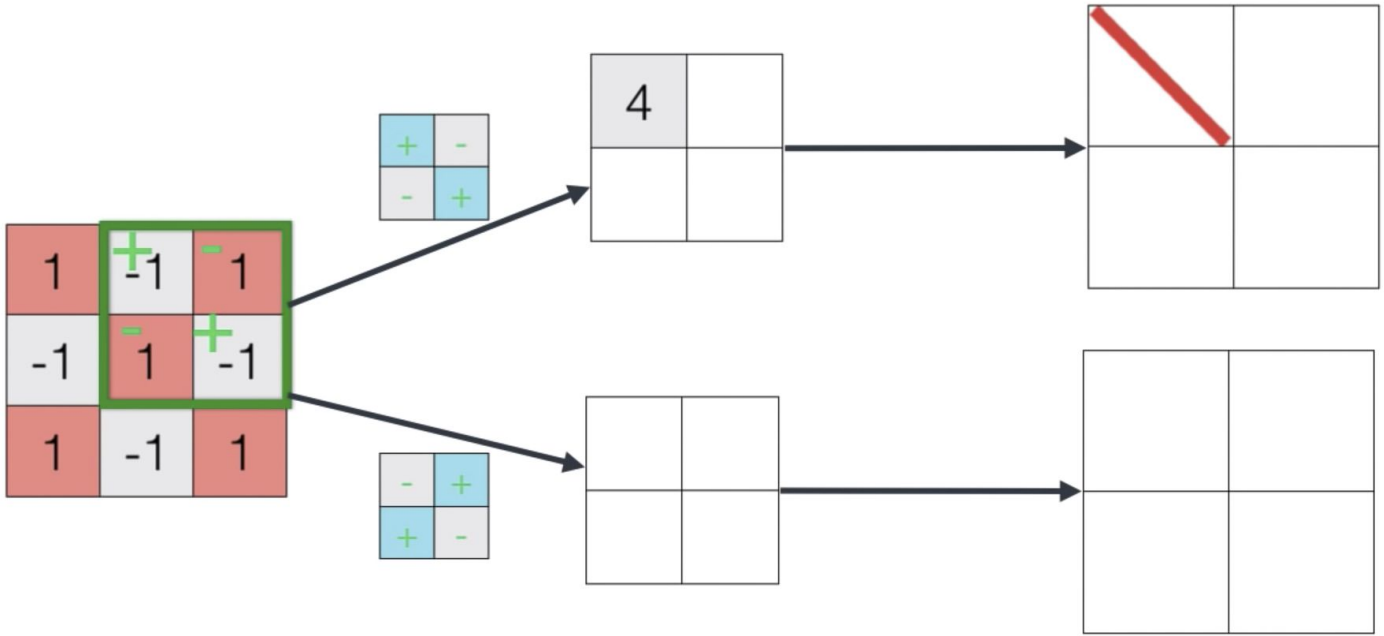
Example



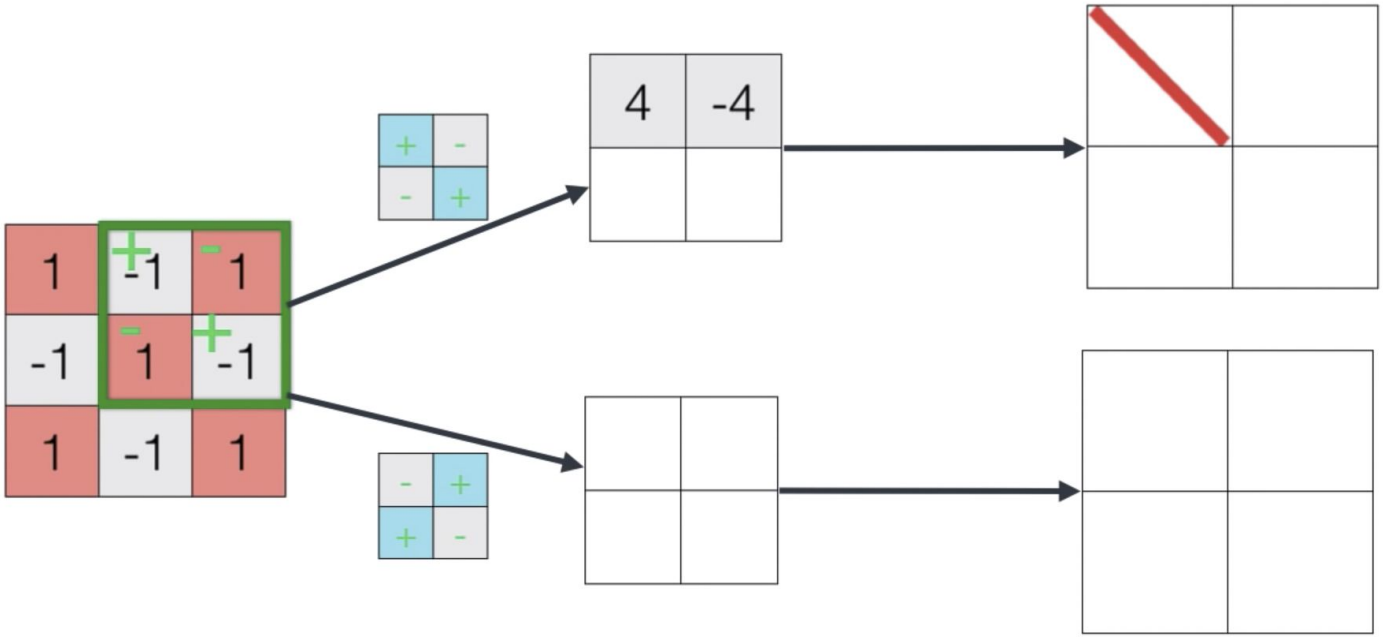
Example



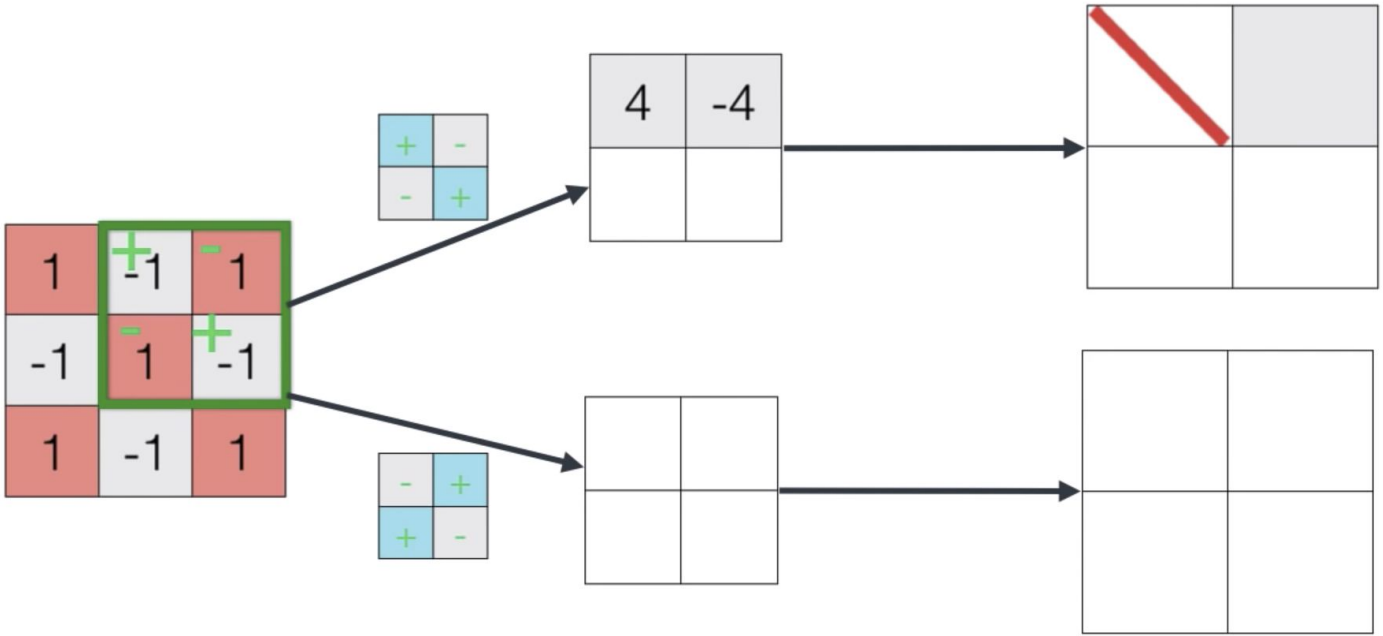
Example



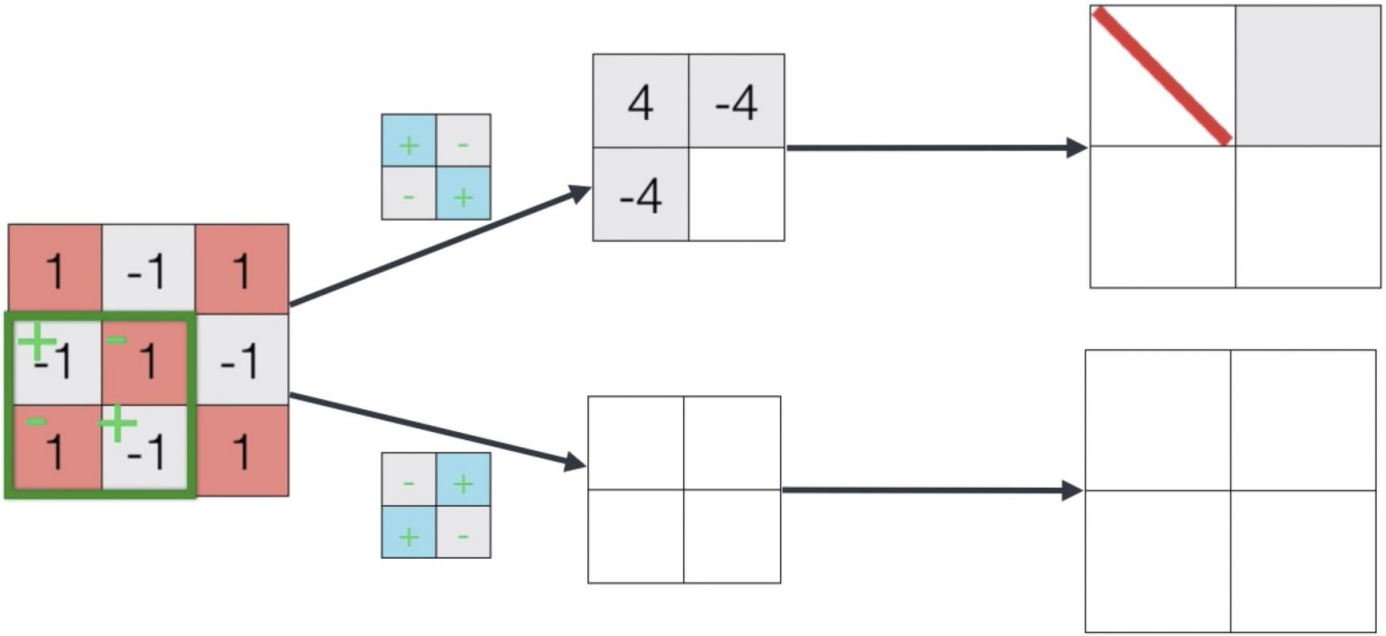
Example



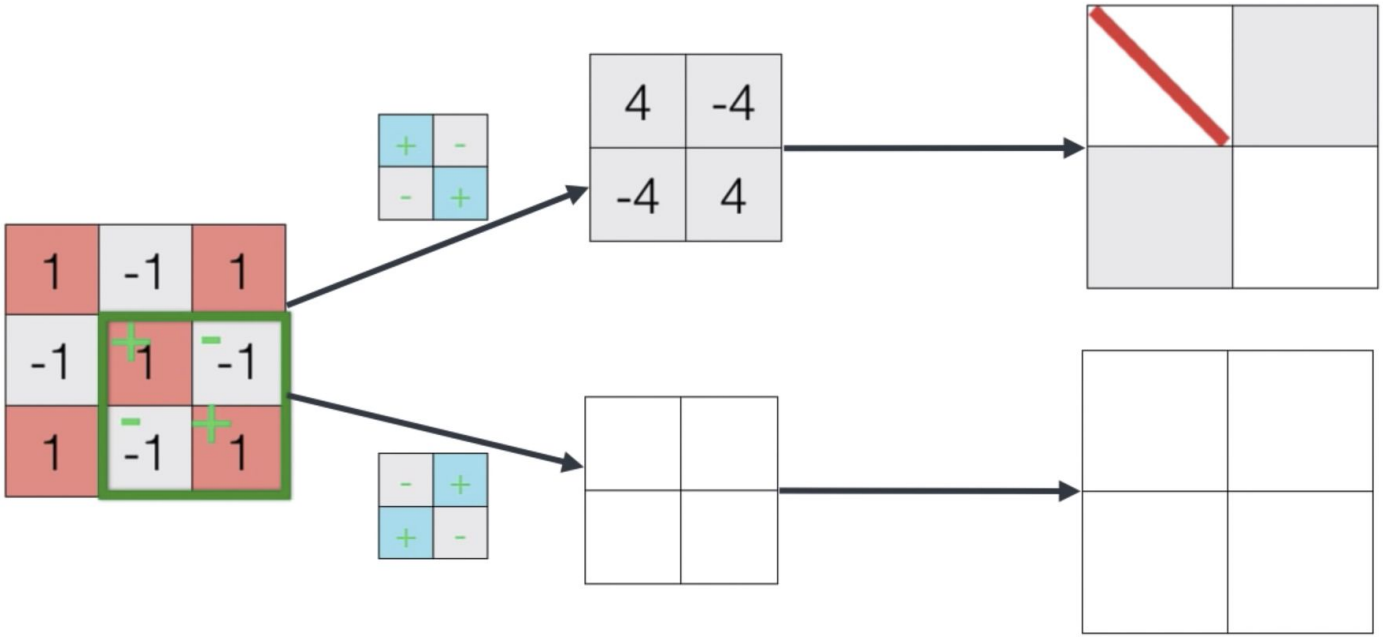
Example



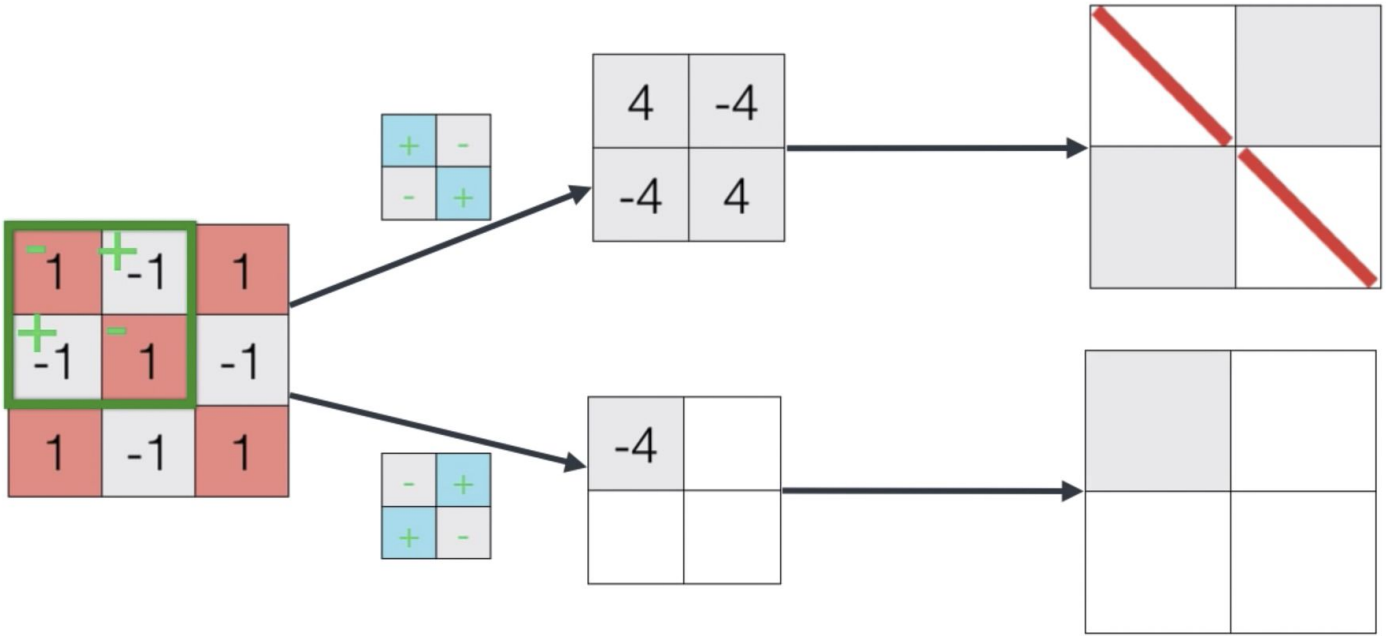
Example



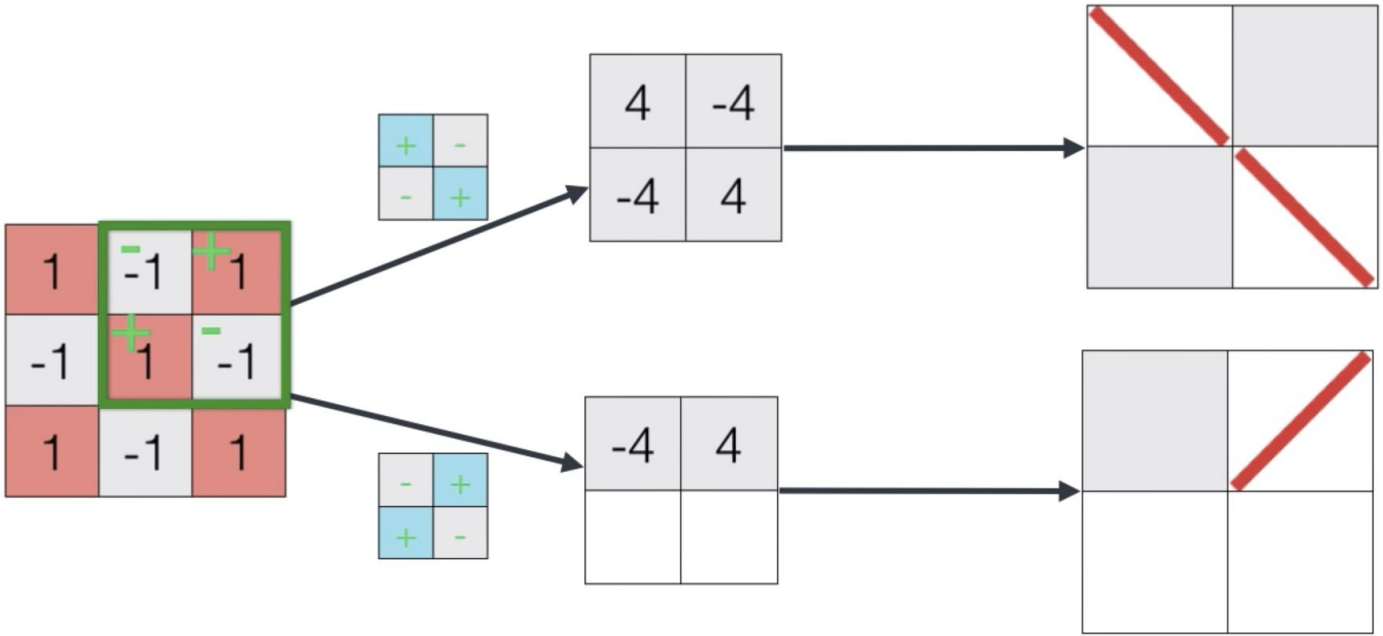
Example



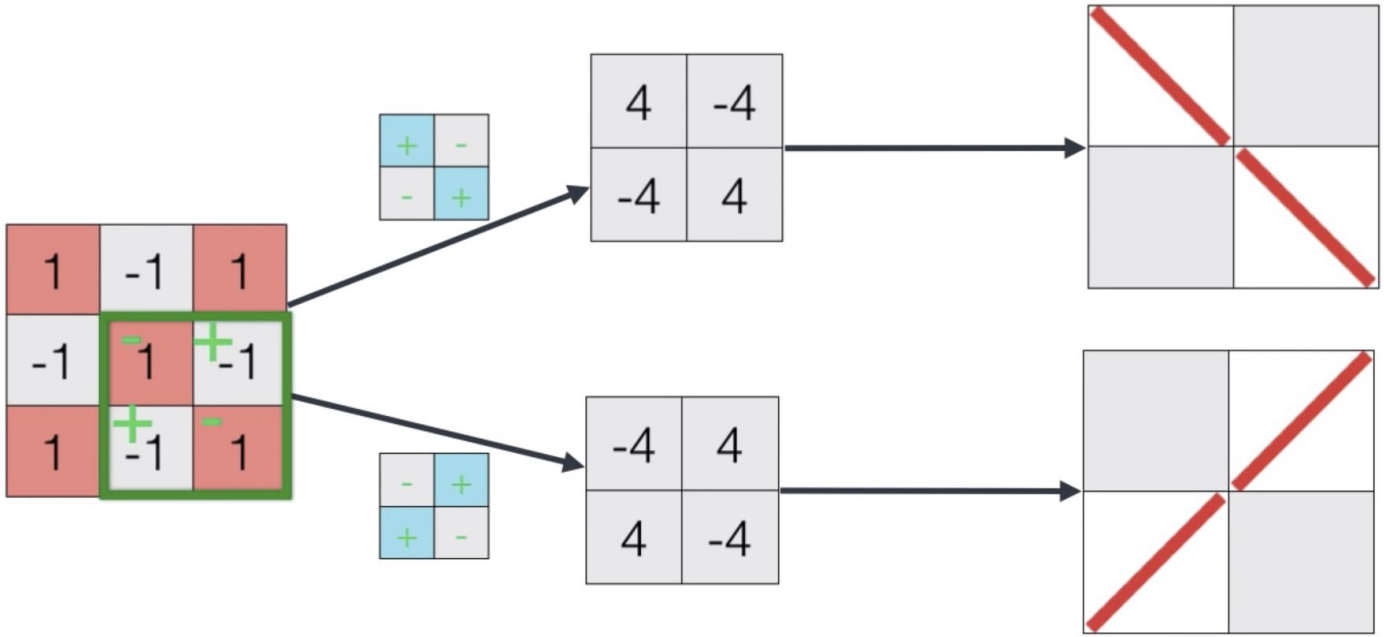
Example



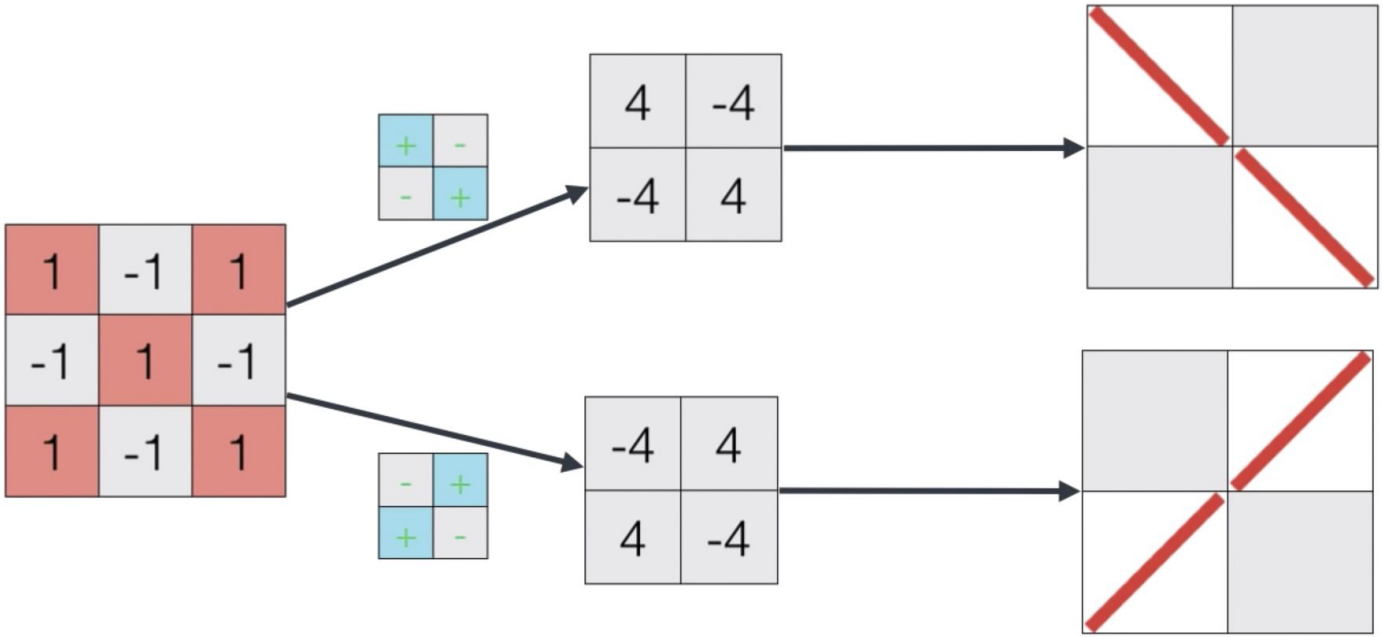
Example



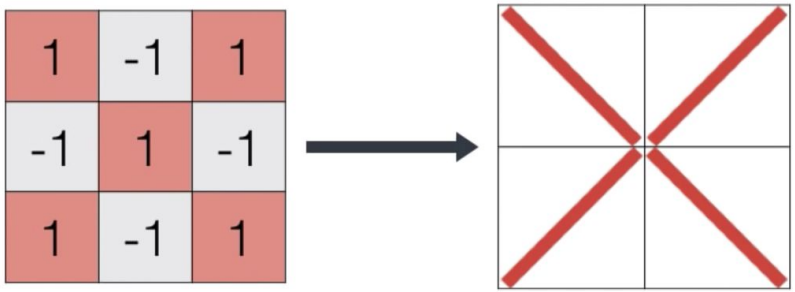
Example



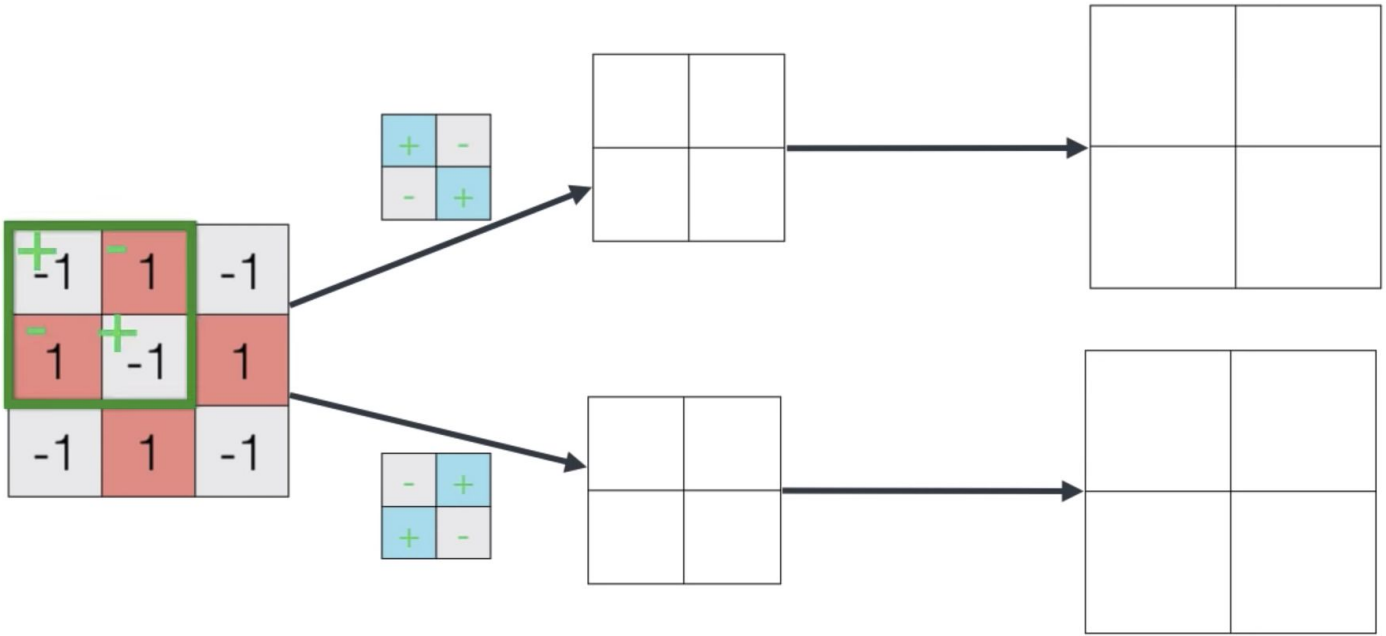
Example



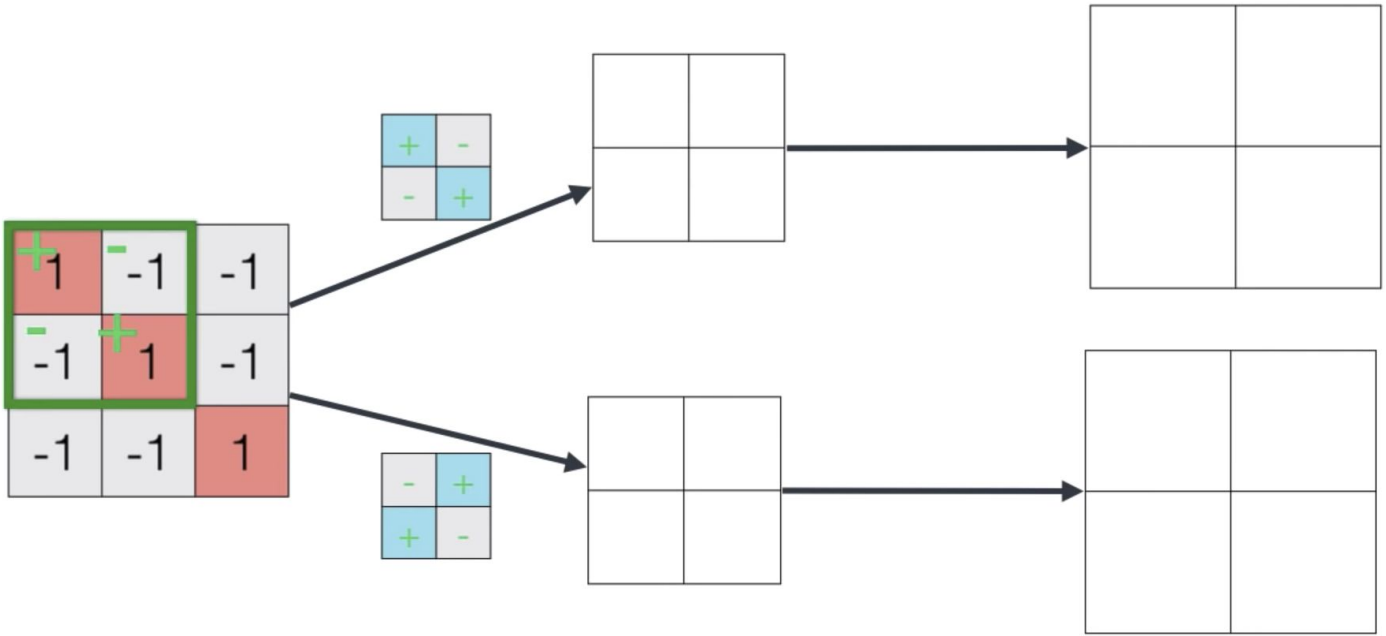
Example



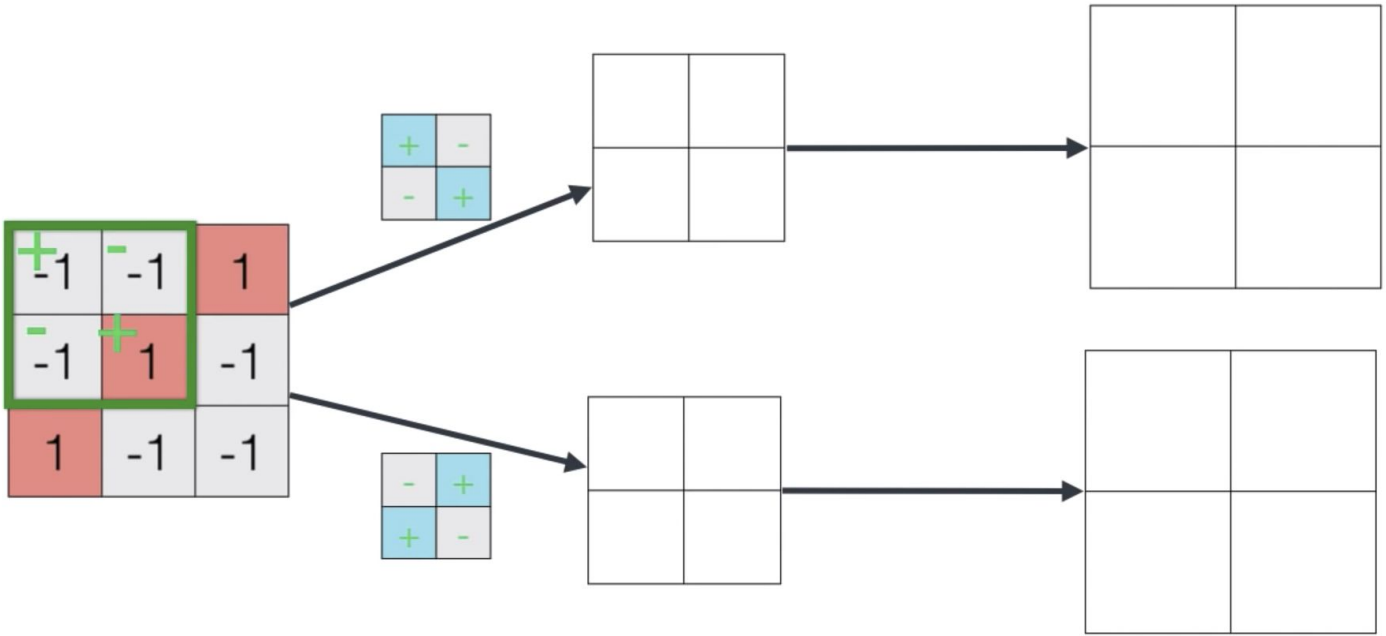
Quick try with O



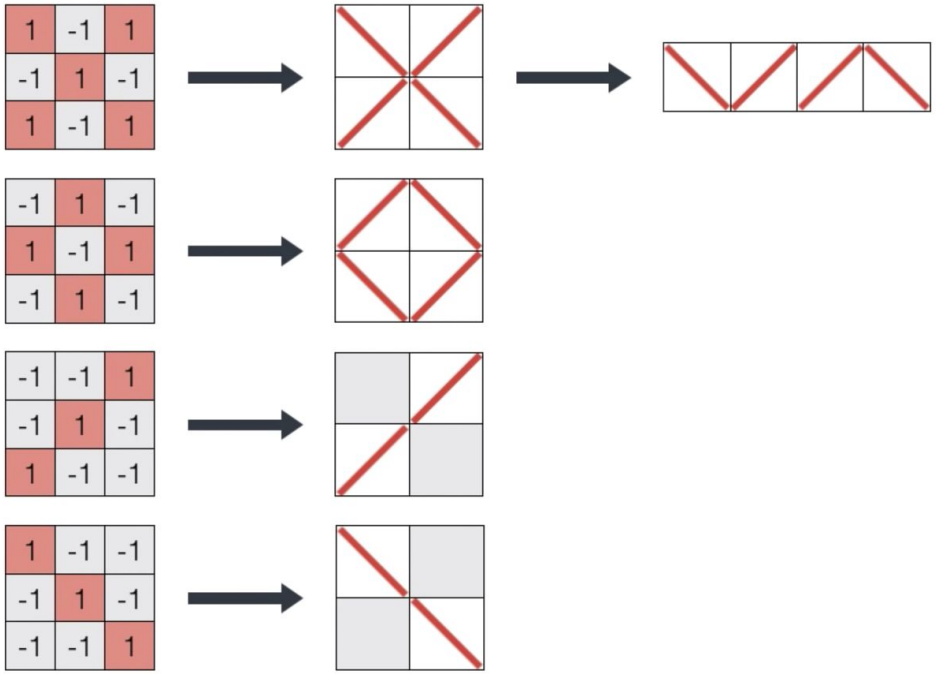
Quick try with \



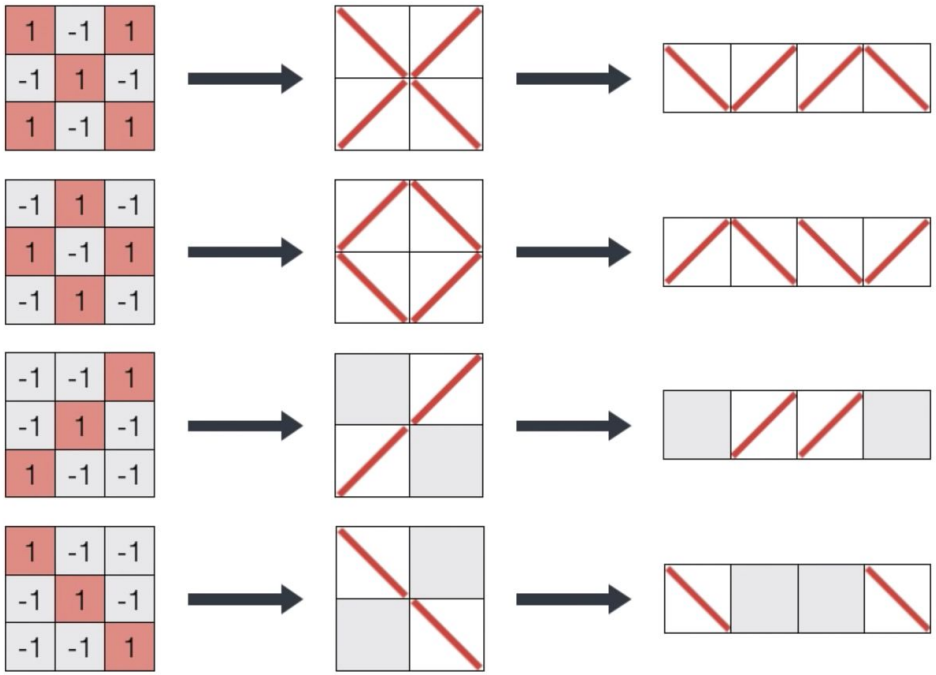
Quick try with /



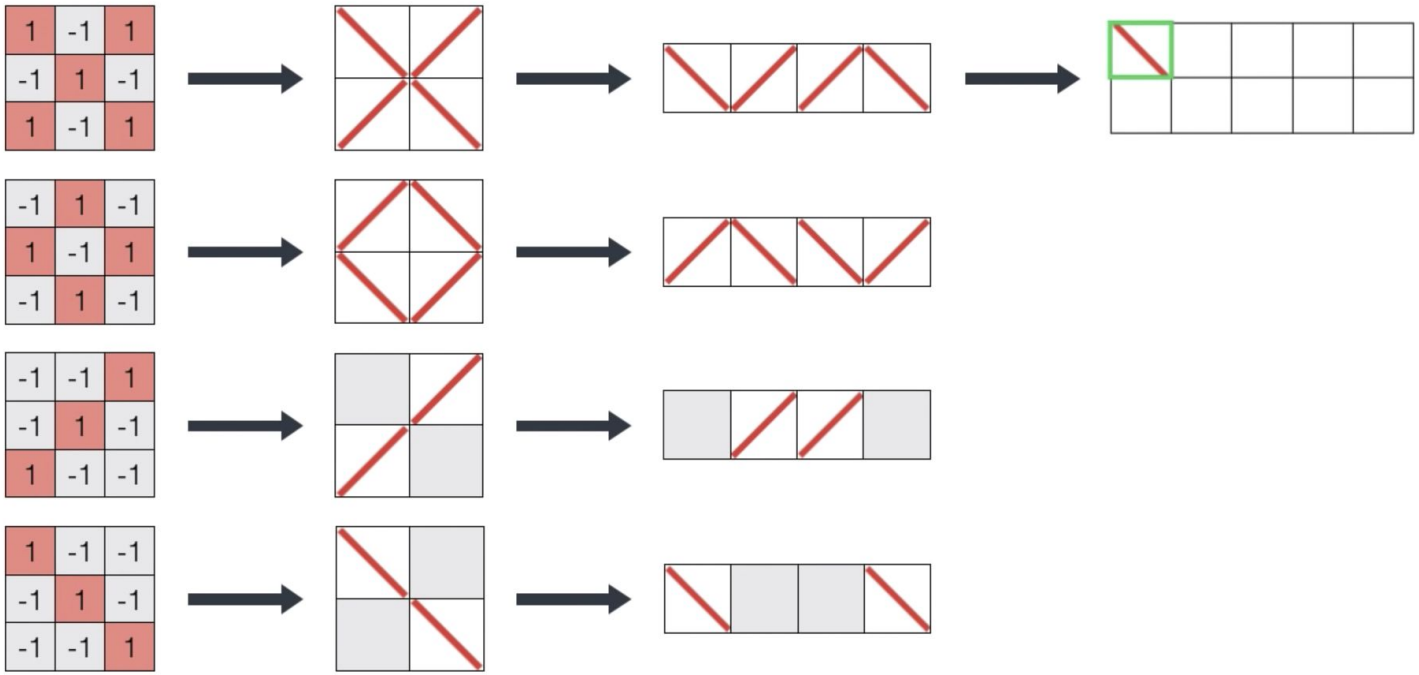
Encoding the filter



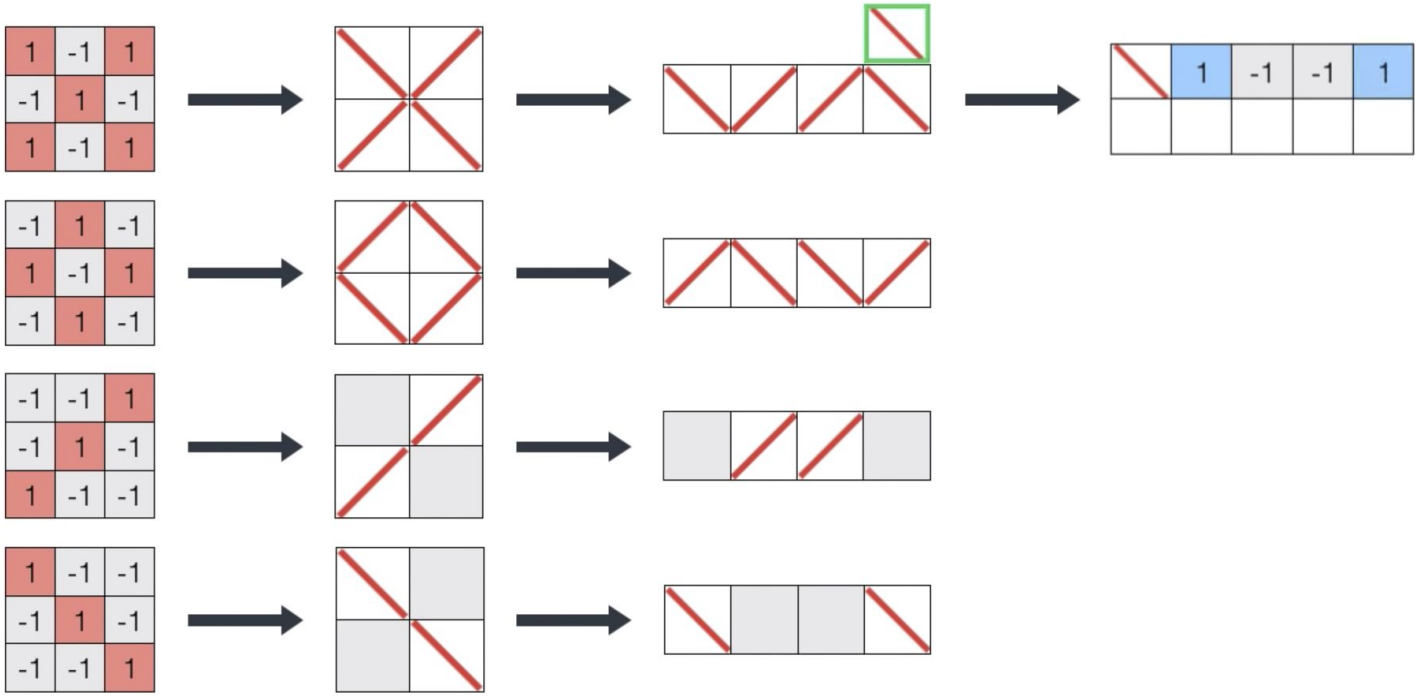
Encoding the filter



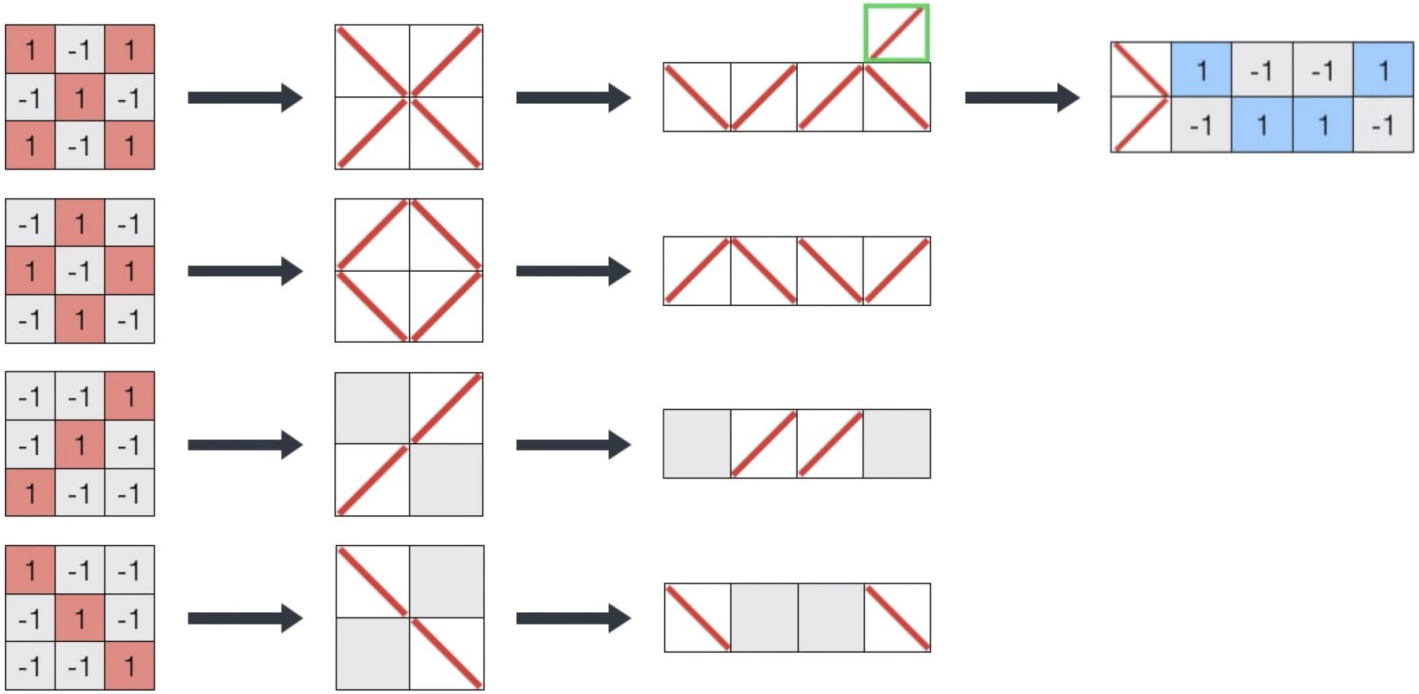
Encoding the filter



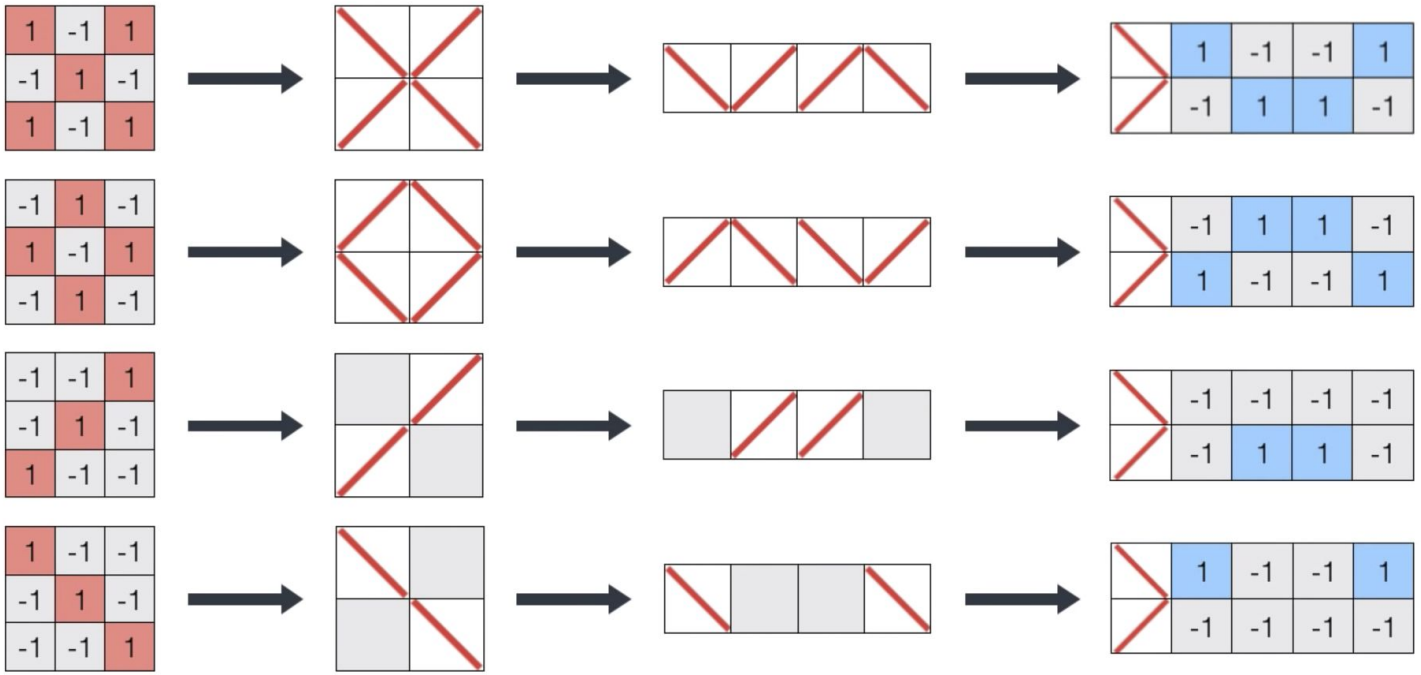
Encoding the filter



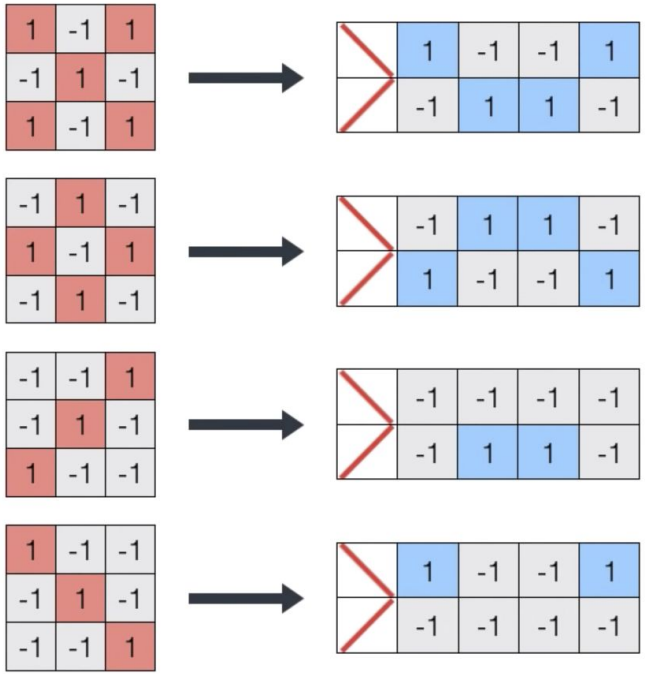
Encoding the filter



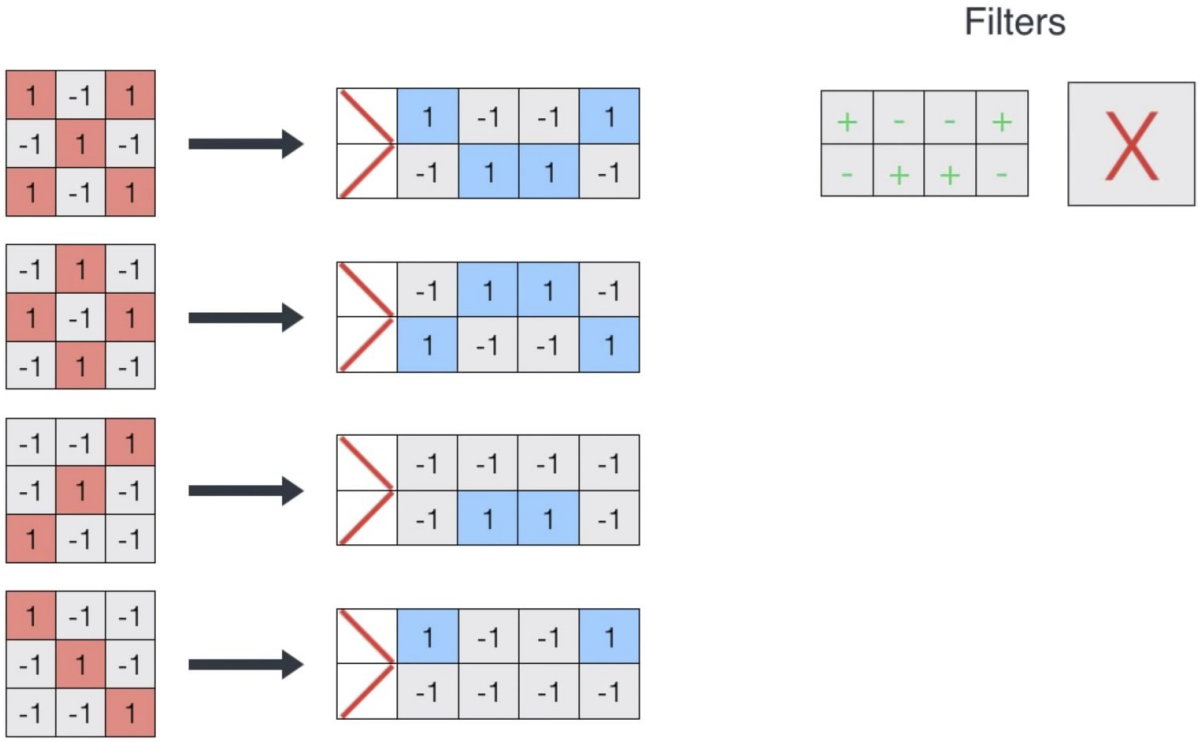
Encoding the filter



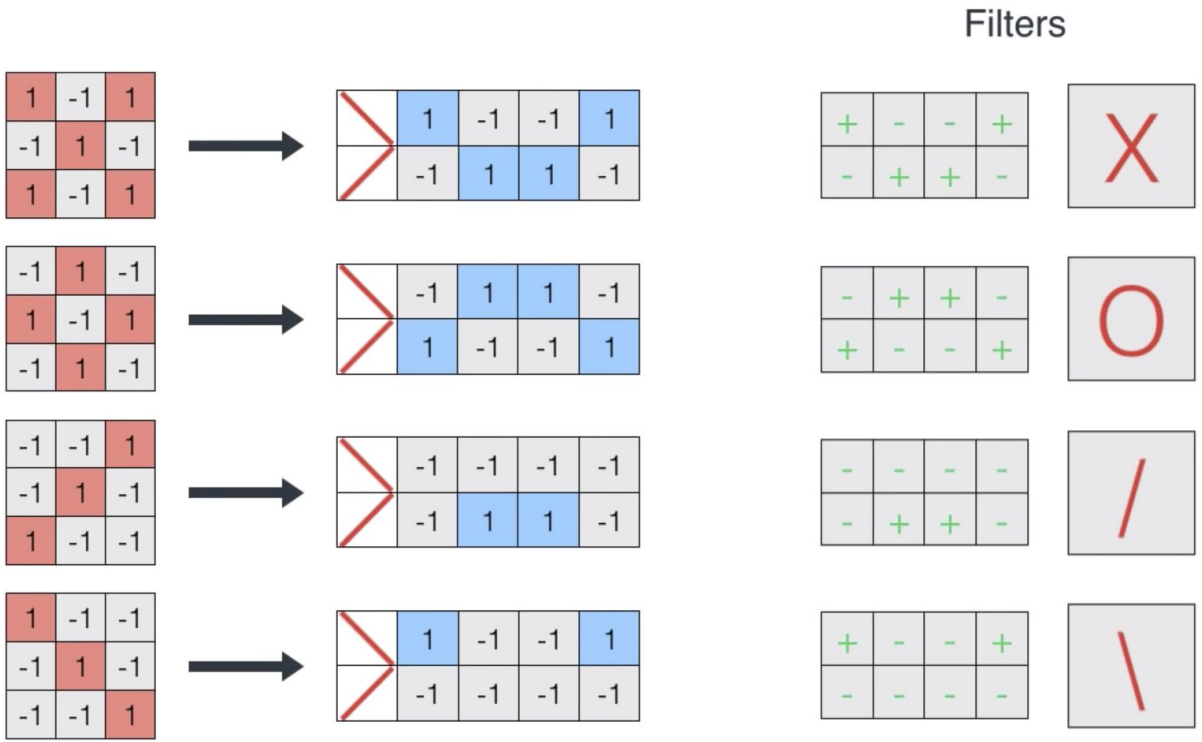
Building the filter



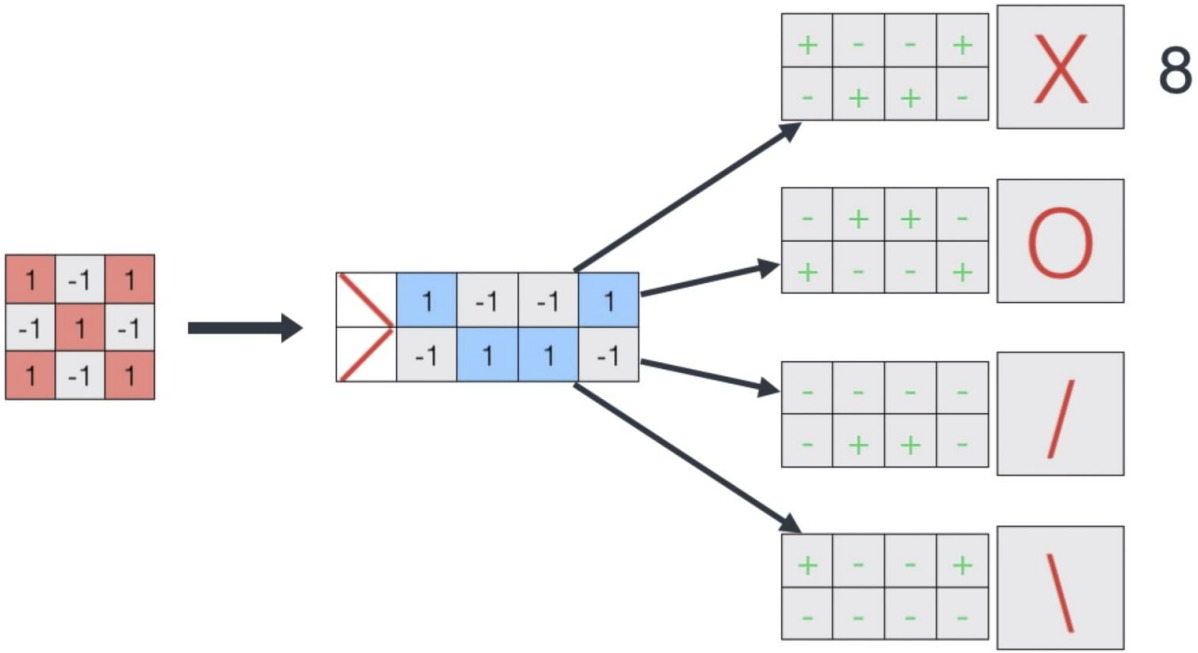
Building the filter



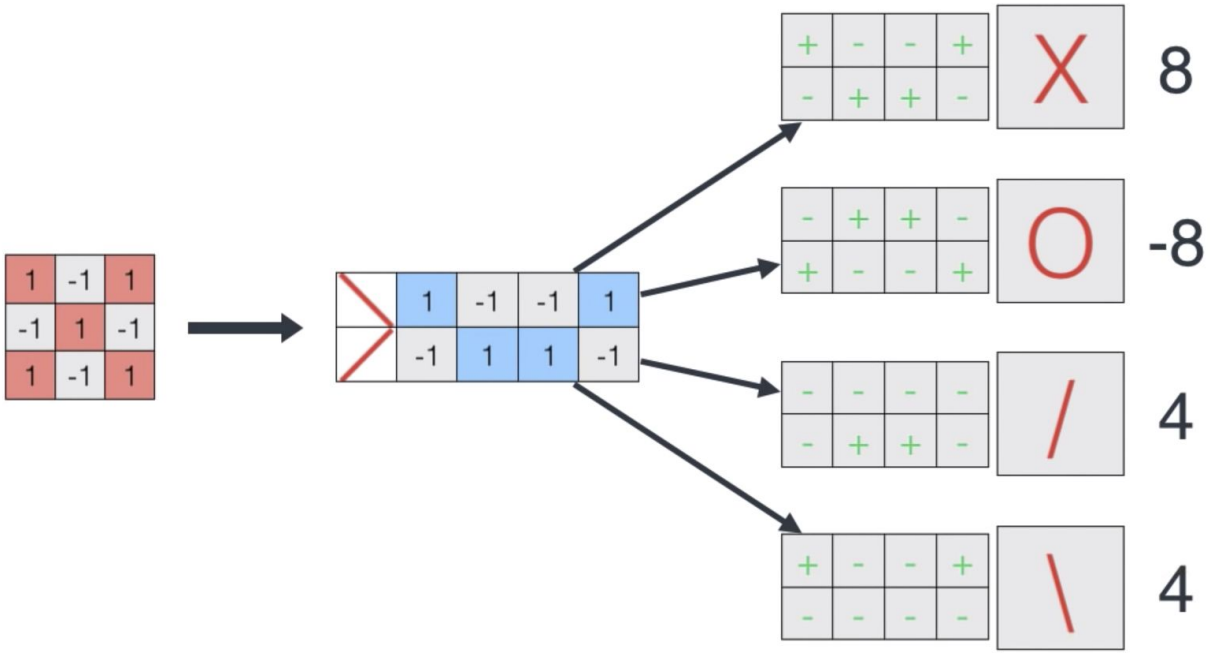
Building the filter



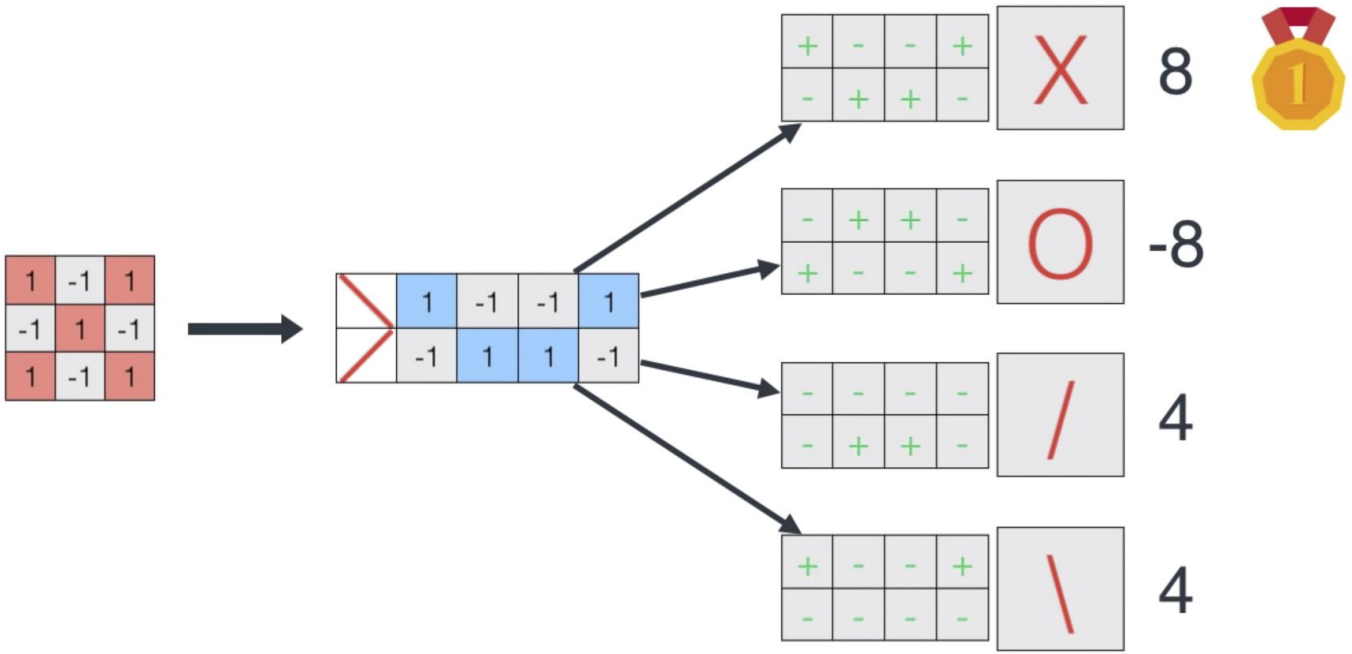
Computing the score



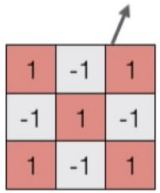
Computing the score



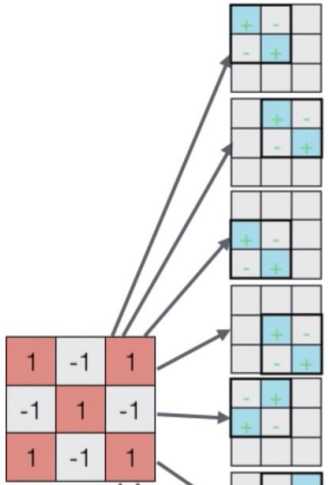
Computing the score



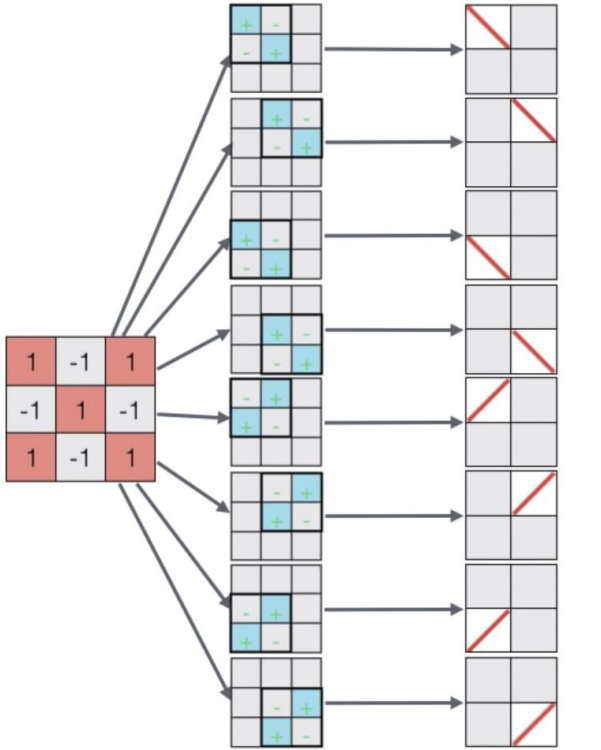
Putting it together



Putting it together

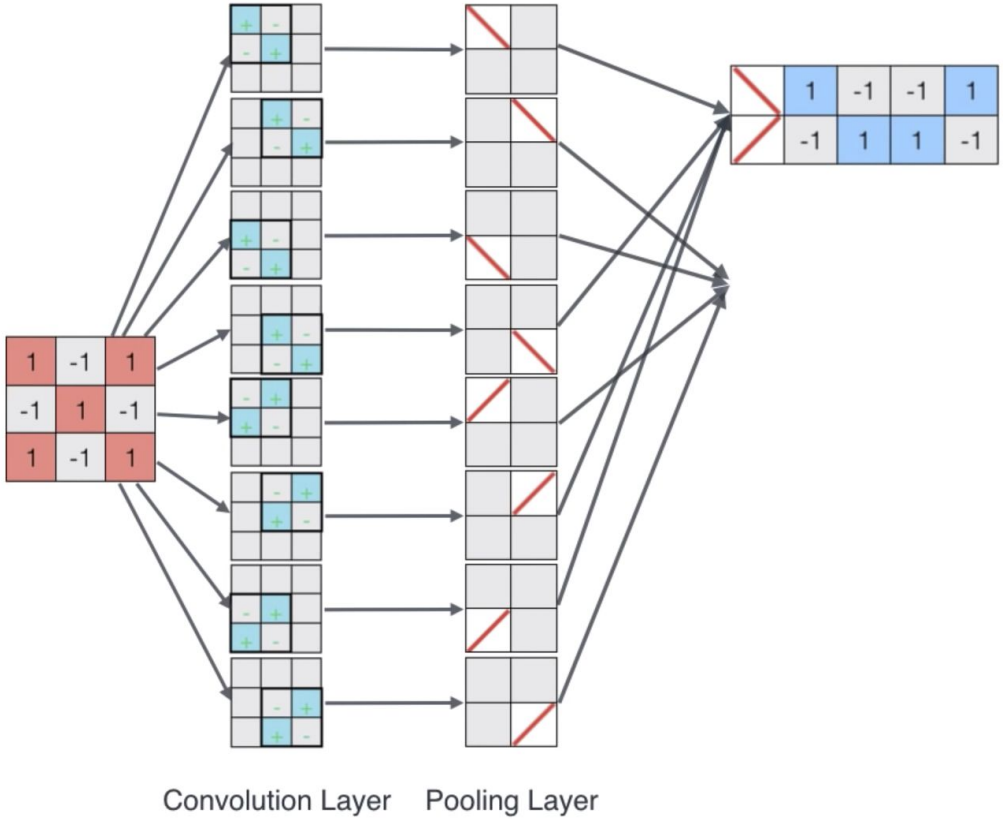


Putting it together

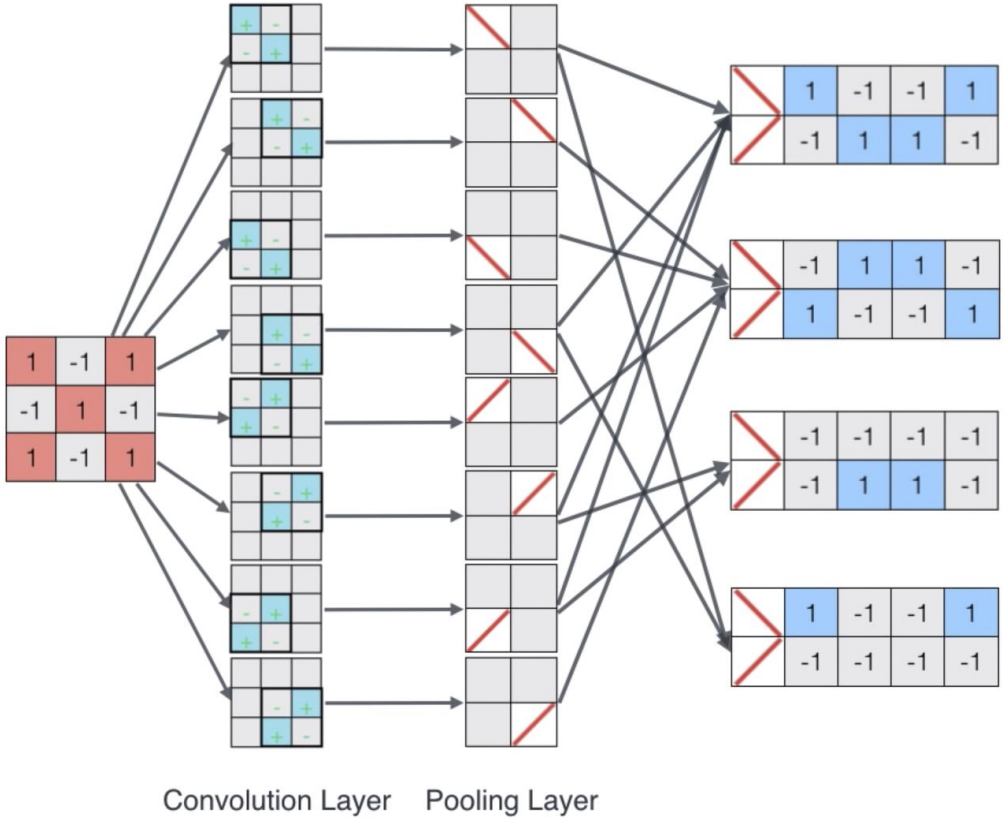


Convolution Layer Pooling Layer

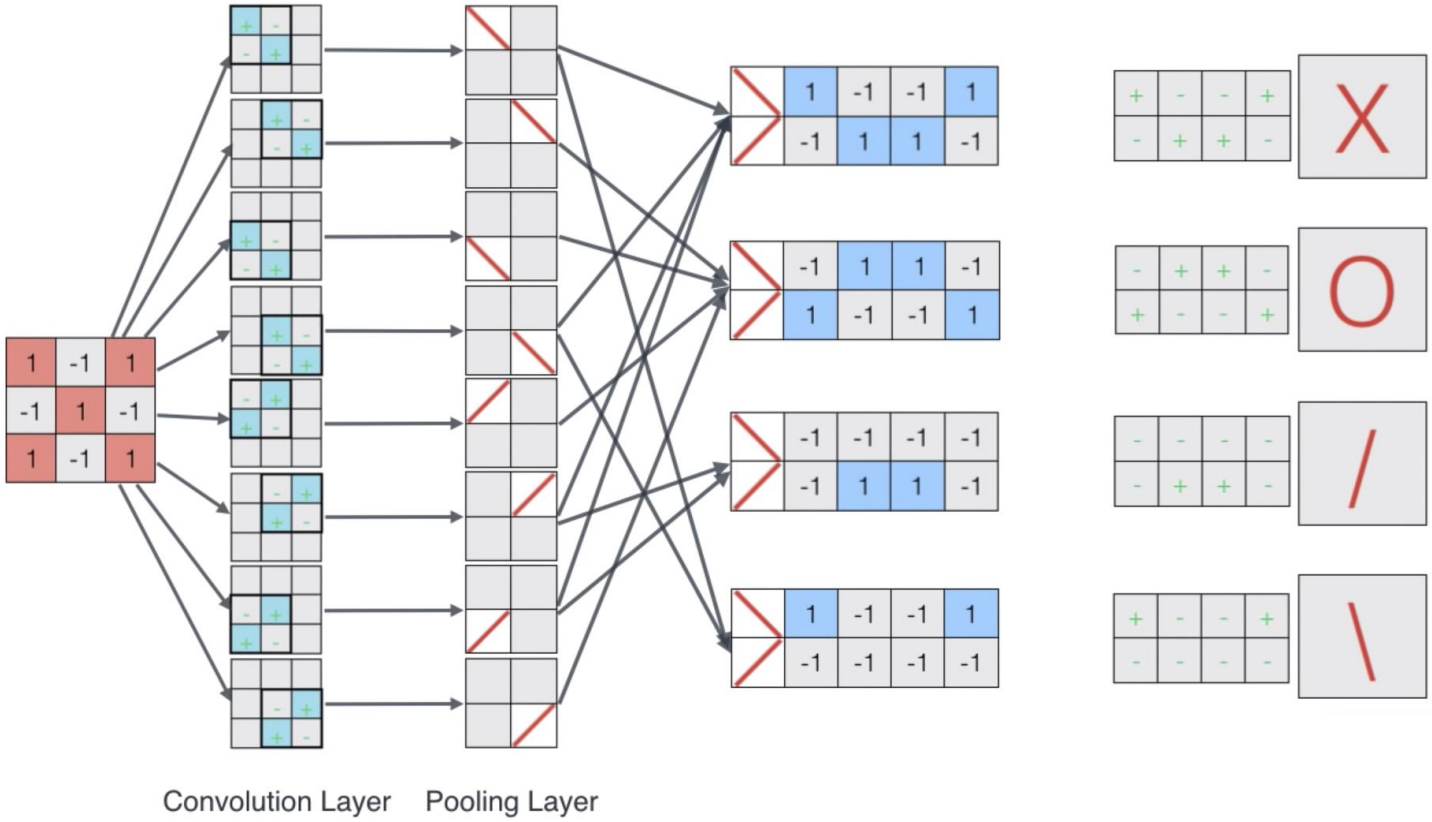
Putting it together



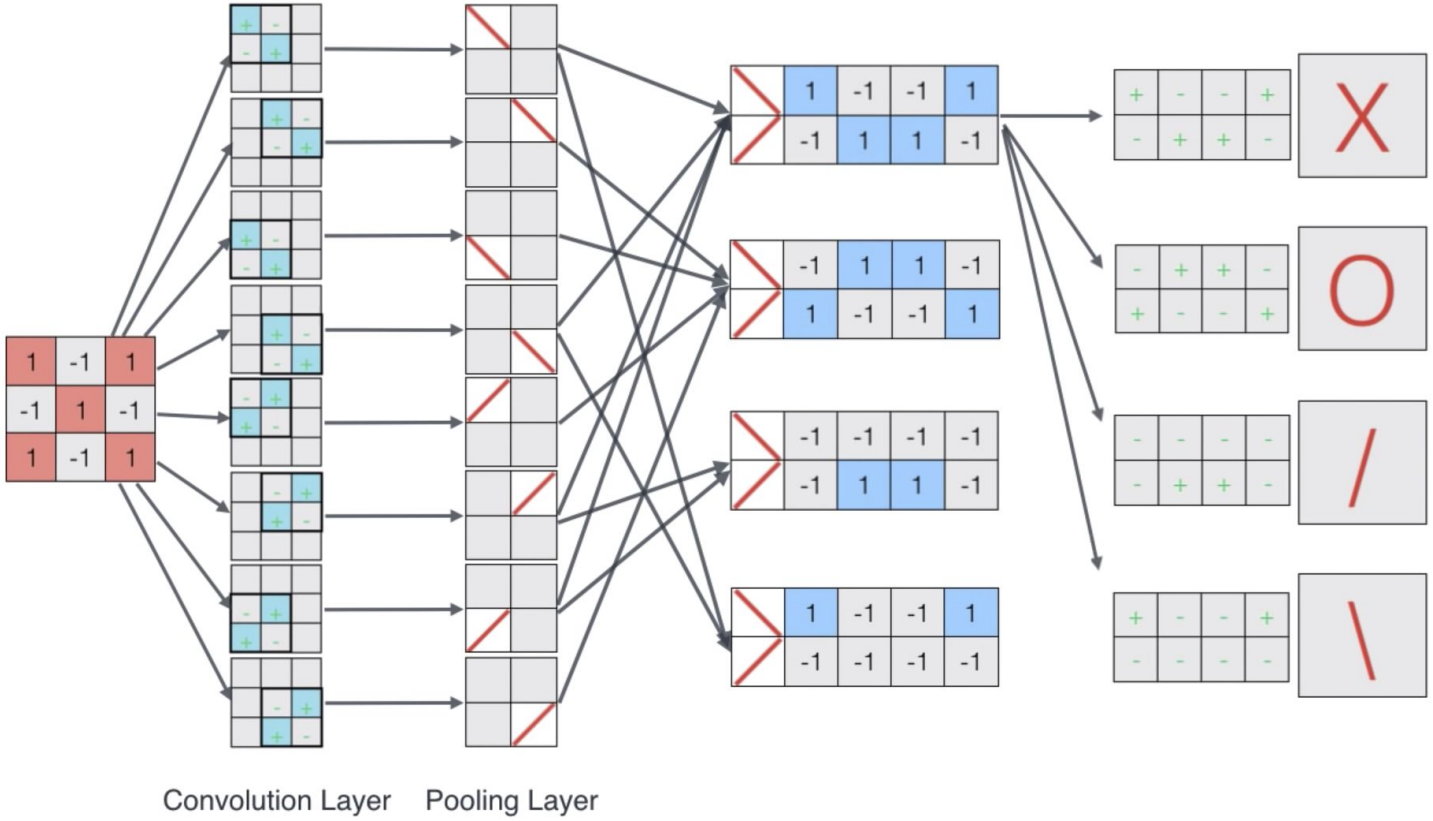
Putting it together



Putting it together



Putting it together



Putting it together