Convolutional Neural Networks

Convolutional and Pooling Layers

N. Rich Nguyen, PhD
SYS 6016

Computers vs. Trivial Perception Tasks

For a long time, computers were **unable** to reliably perform seemingly trivial tasks:

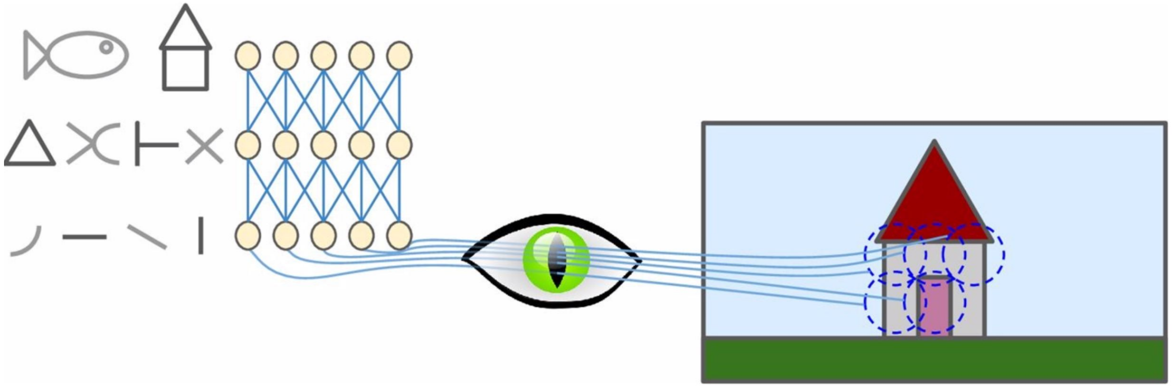
- Detecting a puppy in a picture
- Recognizing spoken words

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

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

Since the advent of **Convolutional Neural Networks,** computers have been able to tackle this challenge

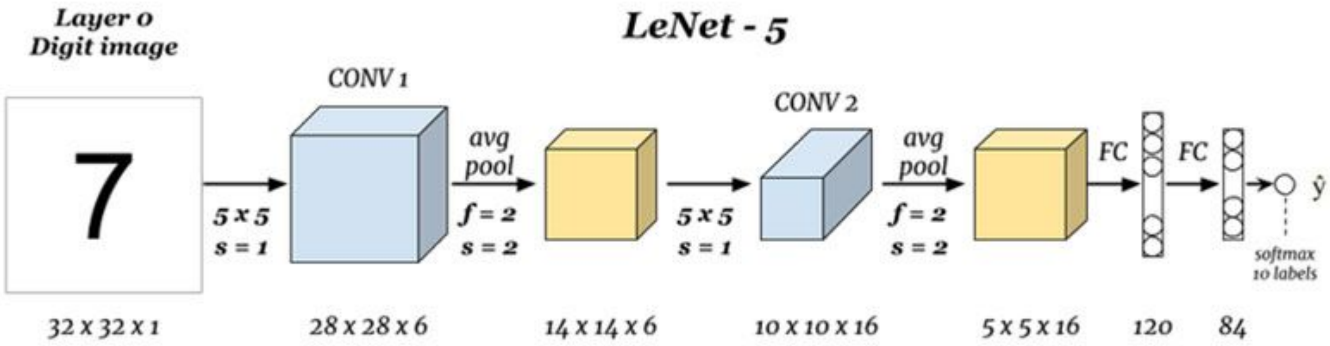
Inspiration: The Architecture of Visual Cortex



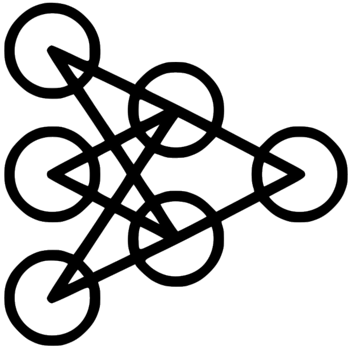
- In 1981, David Hubel and Torsten Wiesel showed that many neurons in the visual cortex has a small local **receptive field**. They received a **Nobel Prize** for this discovery.
- Some neurons have **a larger** receptive field, so they react to more complex patterns based on output of lower-level neuron → this powerful architecture enables humans to recognize all sorts of **complex patterns**.

Convolutional Neural Networks (CNNs)

- CNNs are inspired by the visual cortex architecture
- **Yann LeCun** in 1998 introduced the first CNN called **LeNet-5** architecture, widely use to recognize handwritten check numbers.
- The name "*convolutional*" indicates that the network employs a mathematical operation called **convolution,** a special kind of linear operation.
- Two building blocks: **convolutional layers** and **pooling layers**



Convolutional Layer



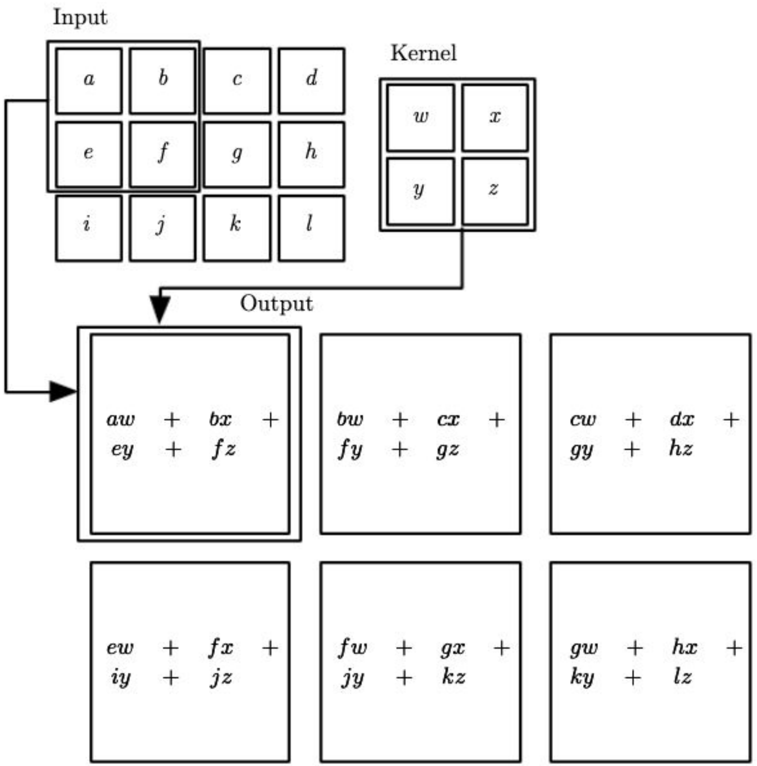
Convolution operation

Convolution is an operation on two multidimensional arrays of real-values:

- The **input** is an array of **data**
- The **kernel** is an array of **parameters**

For images, we use a 2D input `I`, and 2D kernel `K`:

1. `K` is put on top of `I`, and compute the **dot product** of corresponding pixels.
2. `K` is moved in a left-right and then up-down (at one pixel stride) of `I` and repeat step 1.
3. The resulting output is also a 2D image.



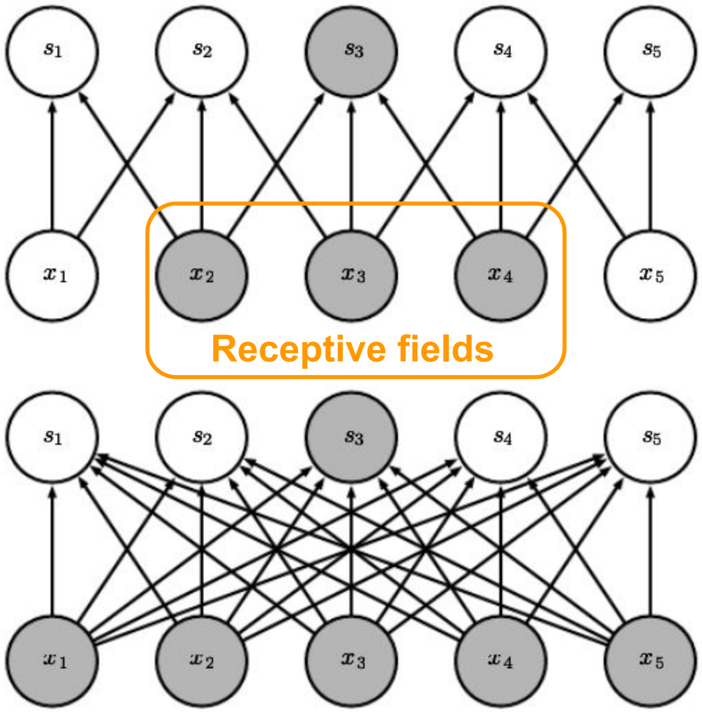
Why use convolution?

Convolution leverages **three important properties** that **improve** ML systems:

1. Sparse Interactions
2. Parameter Sharing
3. Equivariant Representations

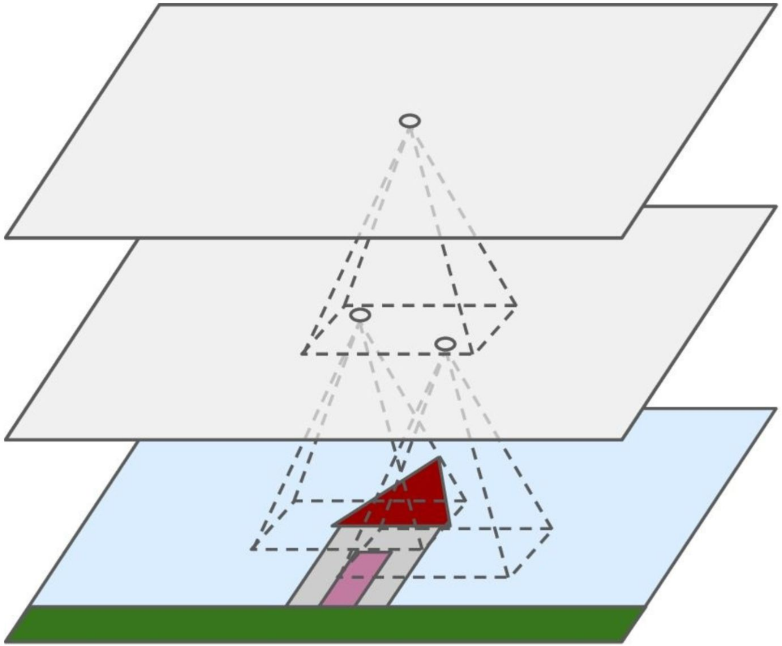
Property 1: Sparse Interactions

- Traditional networks have **dense** interactions: every output unit interacts with every input units.
- Convolutional networks have **sparse** interactions by making the kernel smaller than the input image (ie. tens of pixels vs. thousands of pixels)
- Store fewer parameters, less memory, improved statistical efficiency: $O(m \times n) \rightarrow O(k \times n)$ where k is several orders of magnitude smaller than m.



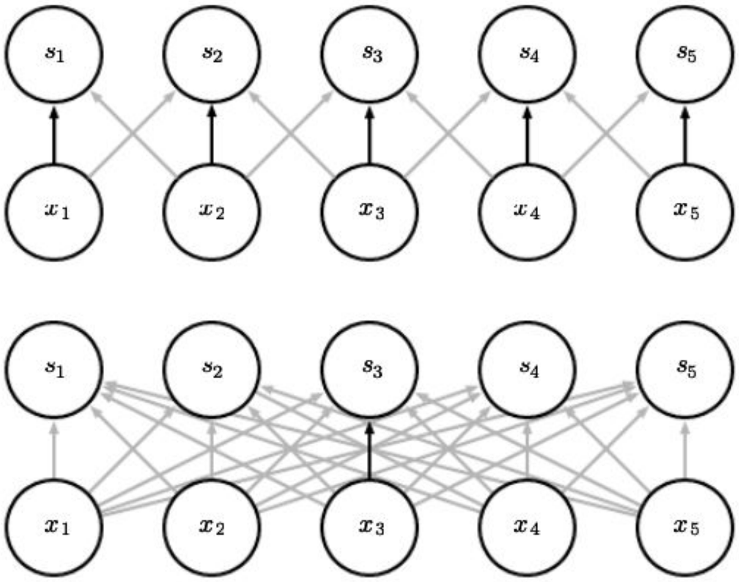
Inspiration from biological neuron

- Similar to their biological counterparts, neurons in the 1st convolutional layer are connected to only to input pixels in their **receptive field**
- Each neuron in the 2nd layer is connected only to 1st layer neurons located within its receptive field
- Concentrate on low-level **local features** in the early conv. layers**,** then assemble them into higher-level features, and so on → work well for **object recognition**



Property 2: Parameter Sharing

- Traditional networks use each element of the weight matrix **exactly once** when computing the output of a layer
- Convolutional networks use the same parameter for **more than one** function in a model: the kernel parameters are used at **every** position of the input
- Instead of learning a separate set of parameters at every location, we learn only one set → save on storage (`kxk` parameters instead of `mxn`)

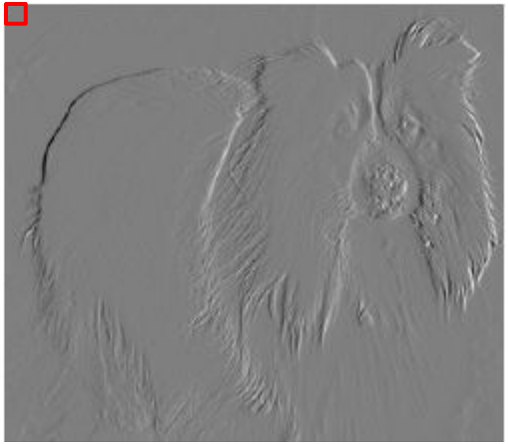


Black arrows indicate the connections that use the same parameter at an input location

Efficiency of edge detection

The right image shows the strength of **all vertical oriented edges** in an input image of a dog. It was formed by taking each pixel of a 320x280 input image and subtracting the value of its neighboring pixel on the left.

input	a	b
	$\otimes$	
kernel	-1	1
	=	
convolution	a(-1) + b(1) = b - a	

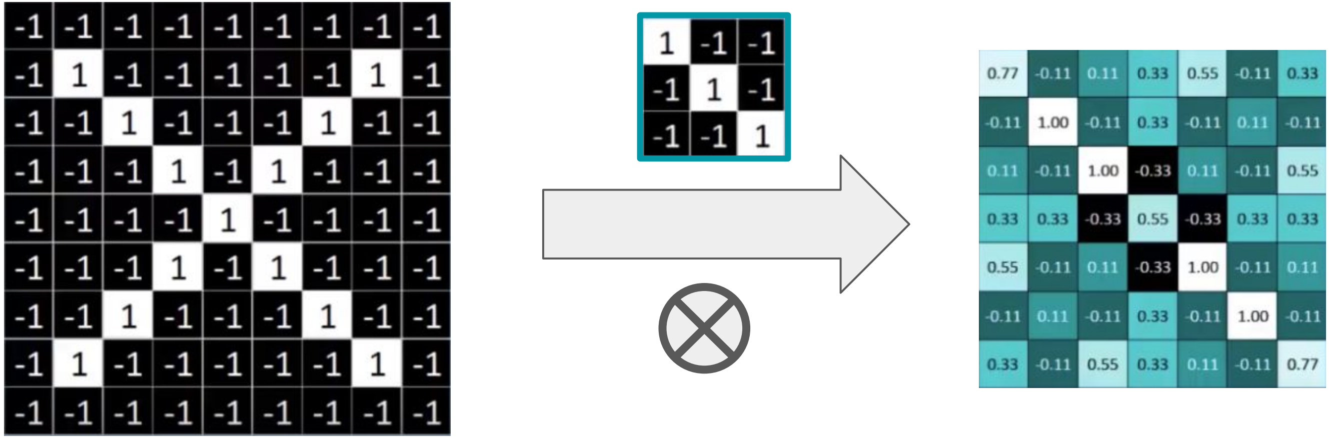


Convolution kernel containing two elements requires 319 x 280 x 3 = 267,960 ops

Using matrix multiplication would take 320 x 280 x 319 x 280 = ~8 billions ops

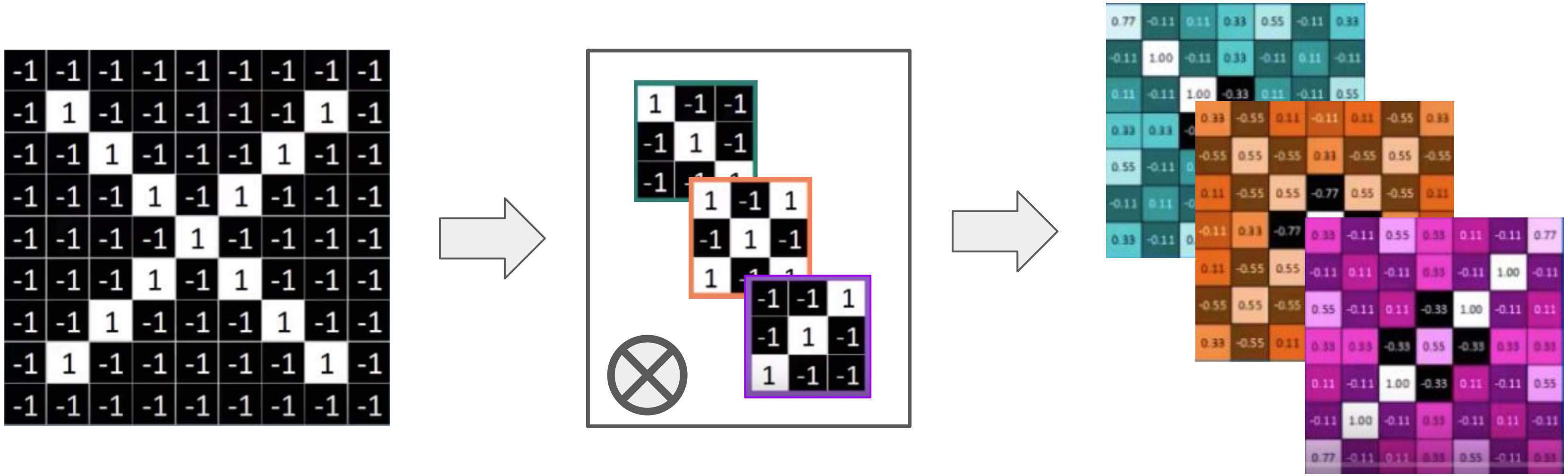
Property 3: Equivariance Representation

- Convolutional layer has a property called **equivariance to translation**.
- A function is **equivariance** when if its input changes, its output changes in the same way: ie. if function `g(I(x,y))=I(x-1,y)` meaning it shifts every pixel of `I` one unit to the left, then `convo(g(I))=g(convo(I))`.
- It means that convolution creates a 2D map of **where** a certain pattern appear in the input → great for detecting a certain local feature → a **feature map**!



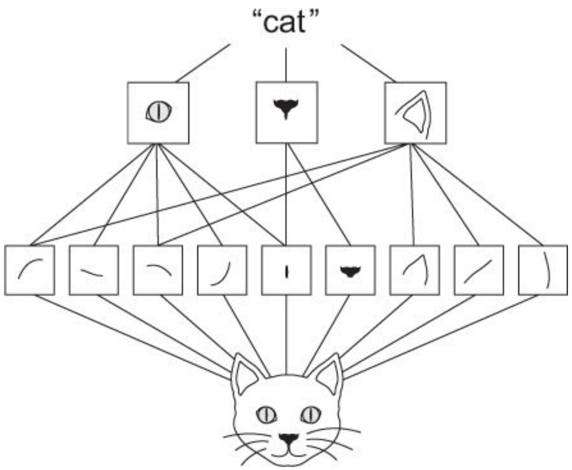
Stacking Multiple Feature Maps

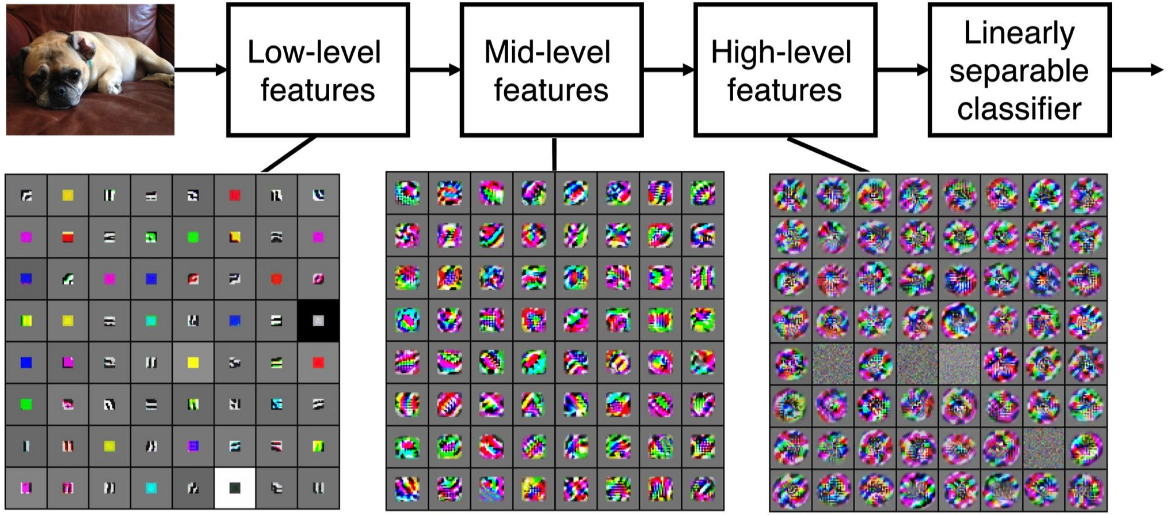
- A convolutional layer simultaneously applies multiple kernels to its input and produces **a stack of feature maps**, making it *capable of detecting multiple features anywhere in its inputs*
- Each convolutional layer is composed several 2D feature maps of equal sizes → more accurate to represent them in a 3D map



From low-level to high-level features

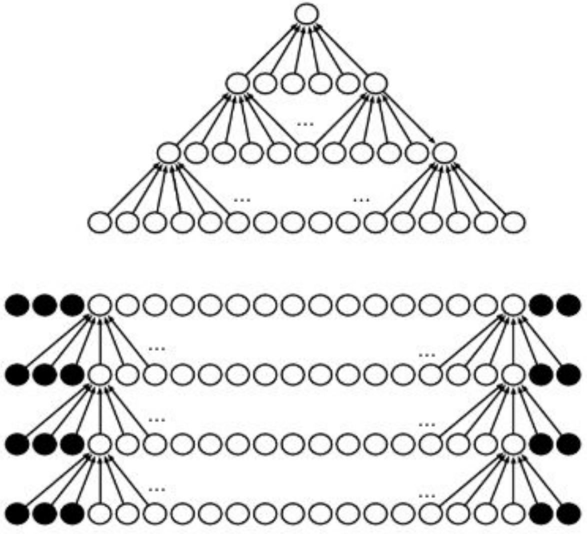
A CNN follows **the spatial hierarchy**: 1st conv layer learns **local patterns** (edges), 2nd conv layer learns **larger patterns** (eyes or ears) made of features from the 1st layer, and so on → multiple data representation in a hierarchy



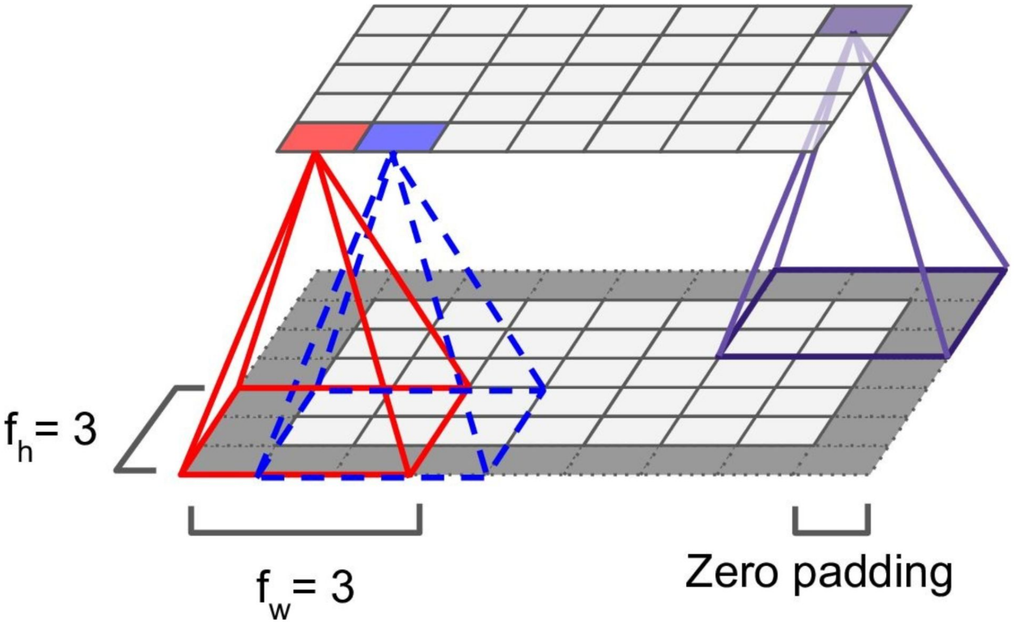


Hyperparameter: Same/Zero Padding

- Convolution operation makes the input image “shrink”
- Keep the output image the **same** size by padding the input image with zeros

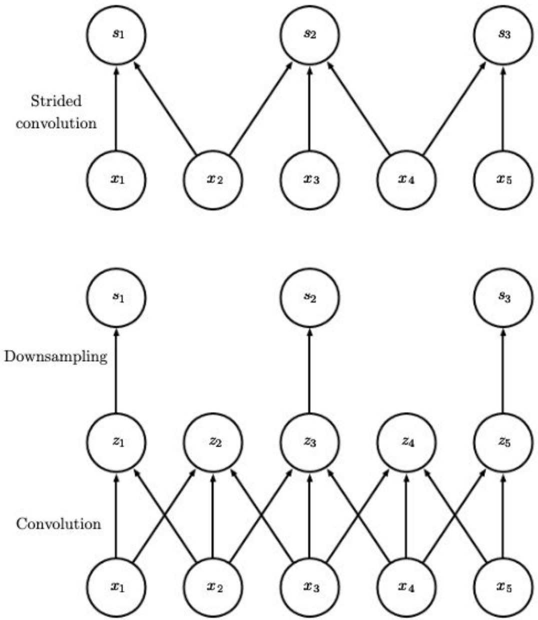


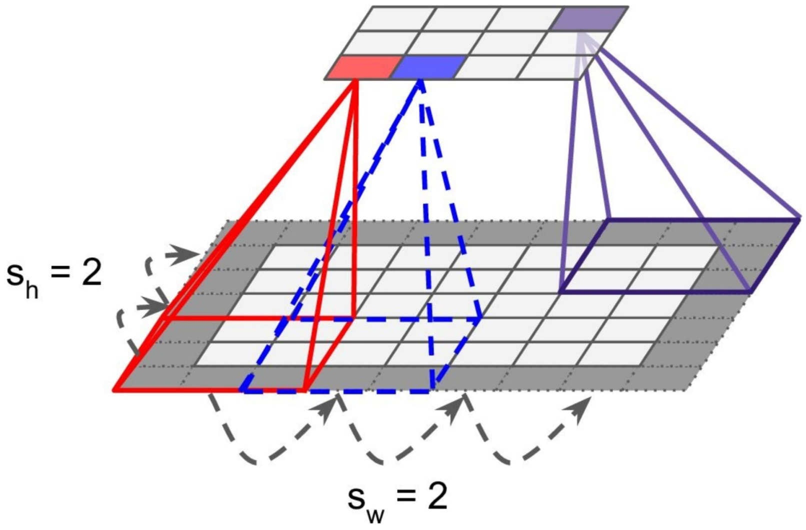
Black circles indicate same/zero padding



Hyperparameter: Stride

- Convolution operation makes the input image “shrink”
- Make the output image shrink faster with **a stride,** meaning to skip some input locations**,** equivalent to **downsampling** after the convolution

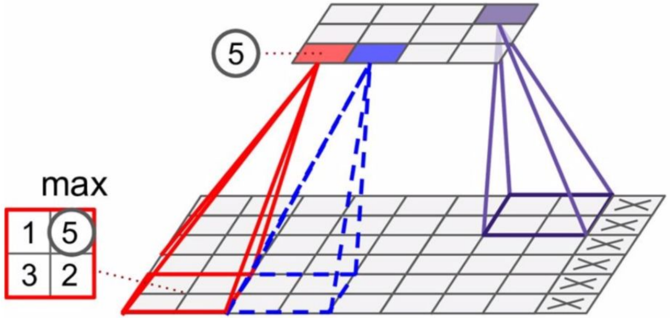


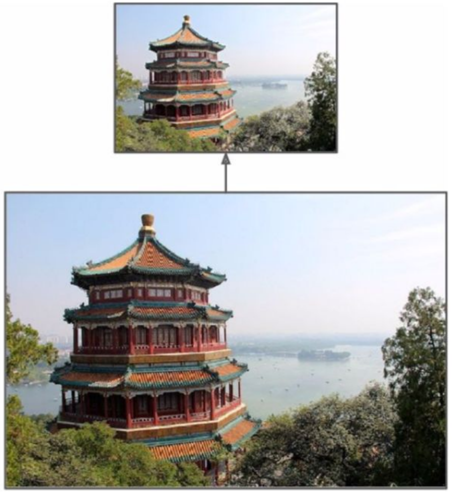


Pooling Layer

Pooling Layer

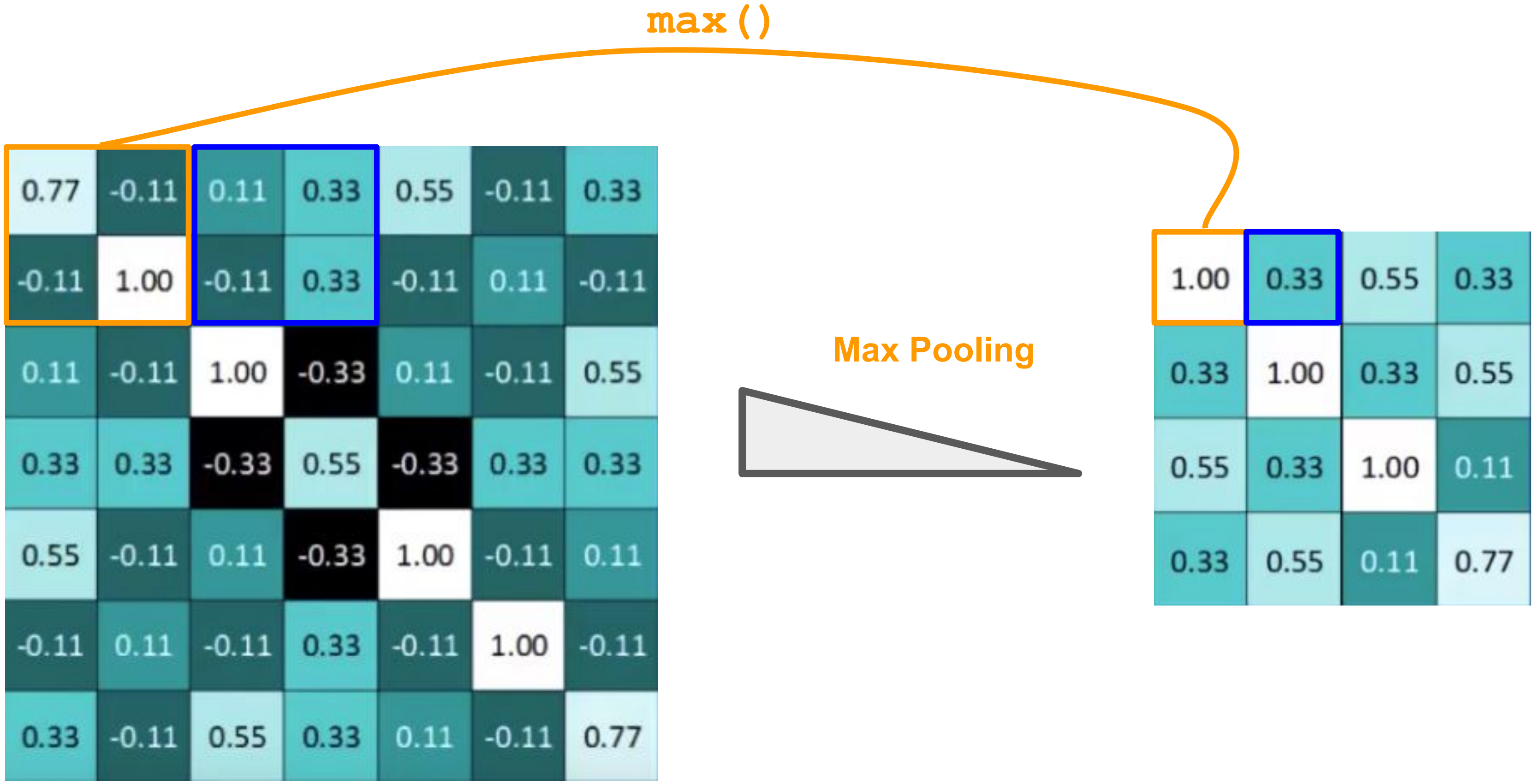
- The goal is to **subsample** the input image in order to reduce the computational load, memory usage, and number of parameters.
- Each neuron in a pooling layer is connected to the outputs of a limited number of neurons in the previous layer (in a receptive field)
- A pooling neuron has **no weights**, all it does is **aggregate the inputs** (using **max** or **average** function)



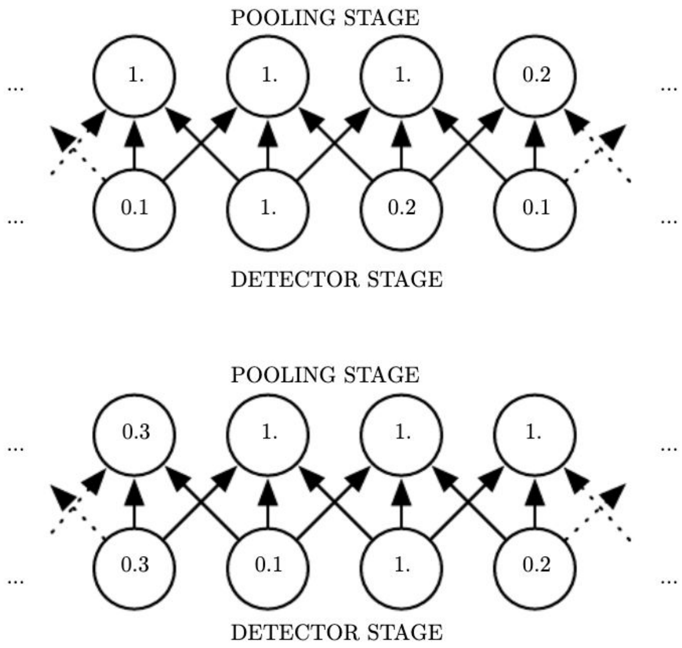


75% of the input values is dropped!

Max Pooling

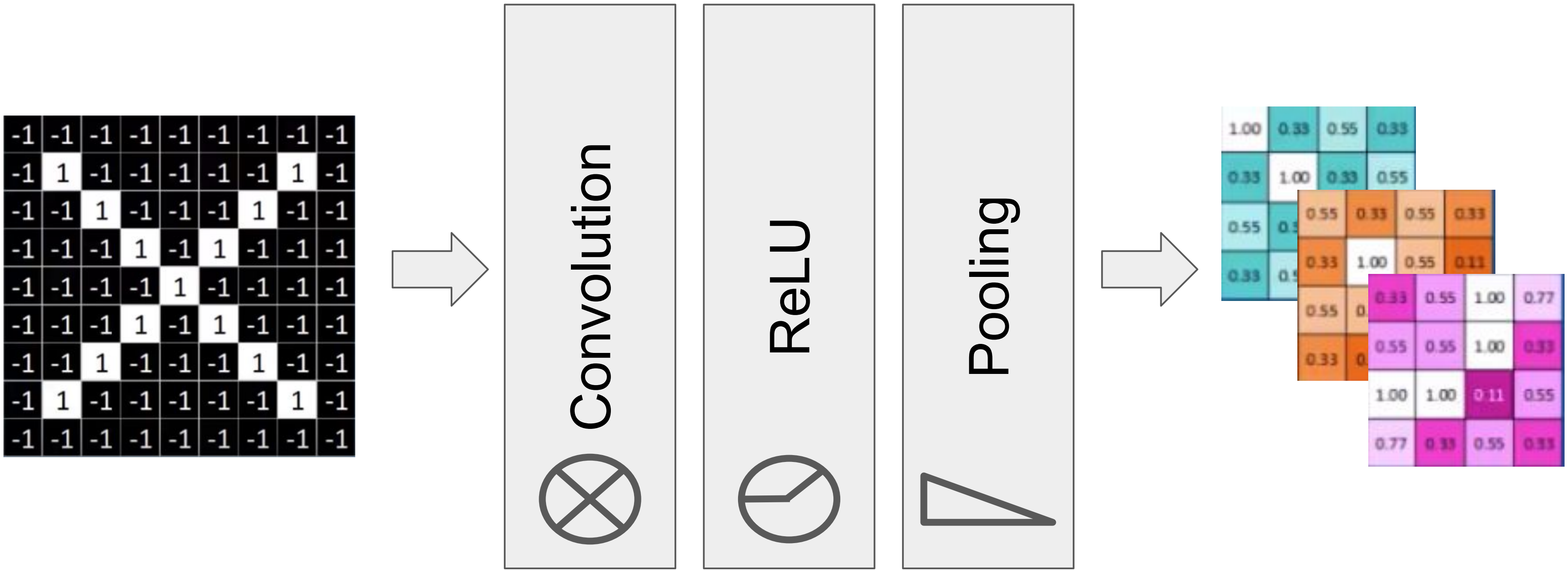

max ()
0.77 -0.11 0.11 0.33 0.55 -0.11 0.33
-0.11 1.00 -0.11 0.33 -0.11 0.11 -0.11
0.11 -0.11 1.00 -0.33 0.11 -0.11 0.55
0.33 0.33 -0.33 0.55 -0.33 0.33 0.33
0.55 -0.11 0.11 -0.33 1.00 -0.11 0.11
-0.11 0.11 -0.11 0.33 -0.11 1.00 -0.11
0.33 -0.11 0.55 0.33 0.11 -0.11 0.77
Max Pooling
1.00 0.33 0.55 0.33
0.33 1.00 0.33 0.55
0.55 0.33 1.00 0.11
0.33 0.55 0.11 0.77

Max pooling introduces invariance

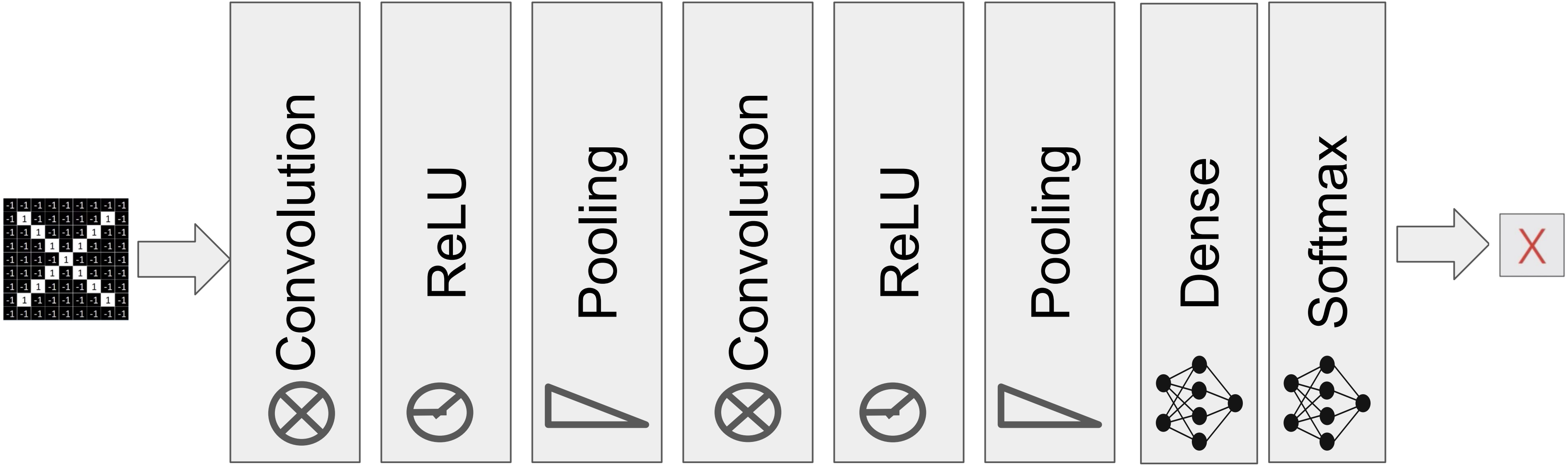


- The top figure shows the outputs of max pooling, with a stride of one pixel between pooling regions each has width of three pixels
- The bottom figure shows a view of the same network, after the **input has been shifted to the right** by one pixel. *Every value in the bottom row has changed, but **only half** of the values in the top row have changed*
- The max pooling units are sensitive only to the maximum value in the neighborhood, not its exact location → **invariant to small translations**

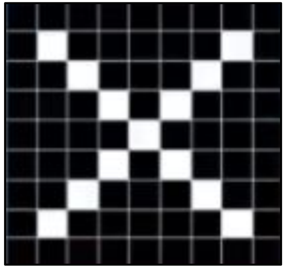
How the layers get stacked

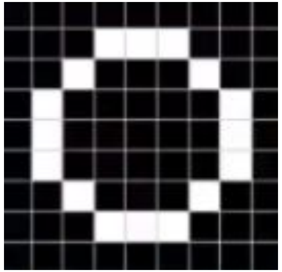


Putting the pipeline together in blocks



An toy example provided the bonus slides for you to try out!



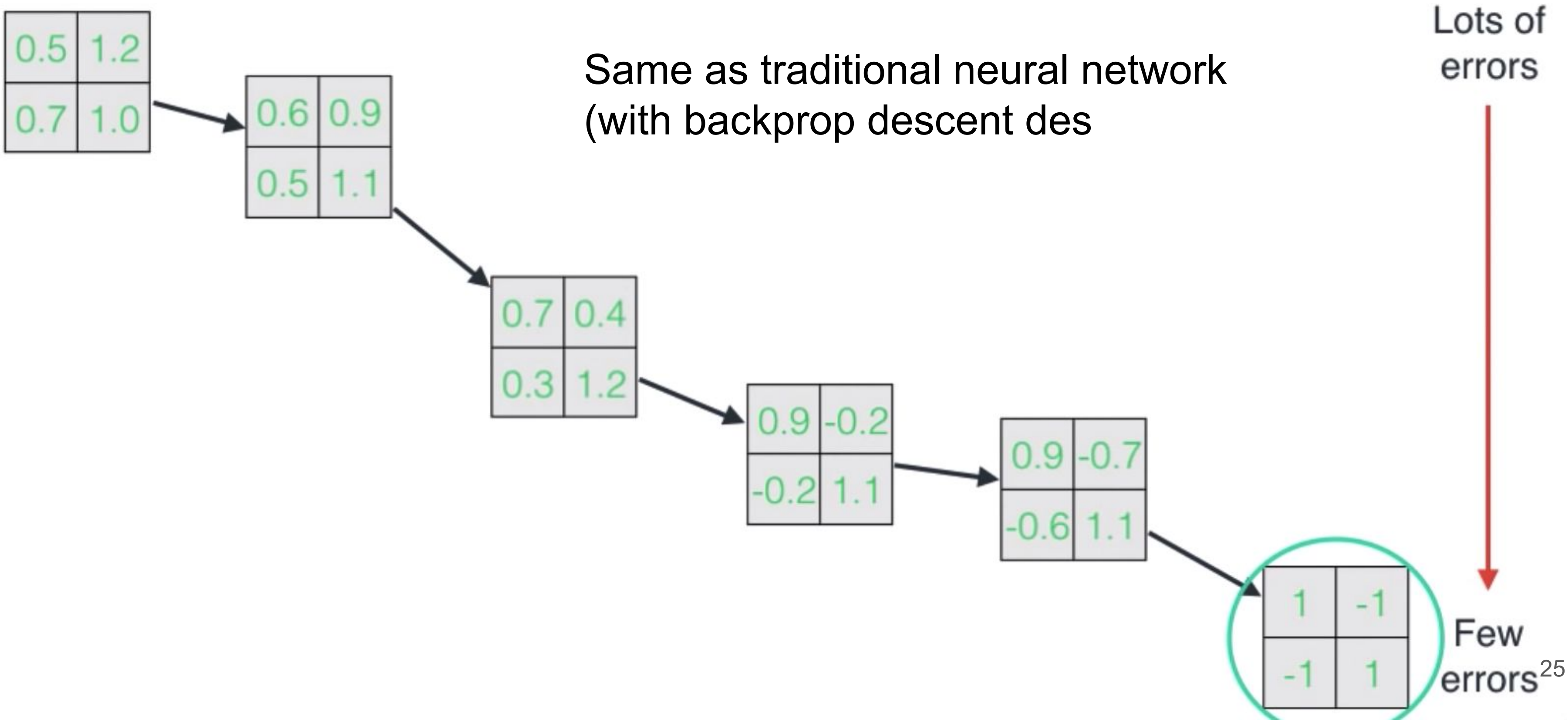


Implementing a CNN architecture

```
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")
])
```

Learning the parameters of the kernels

Same as traditional neural network
(with backprop descent des



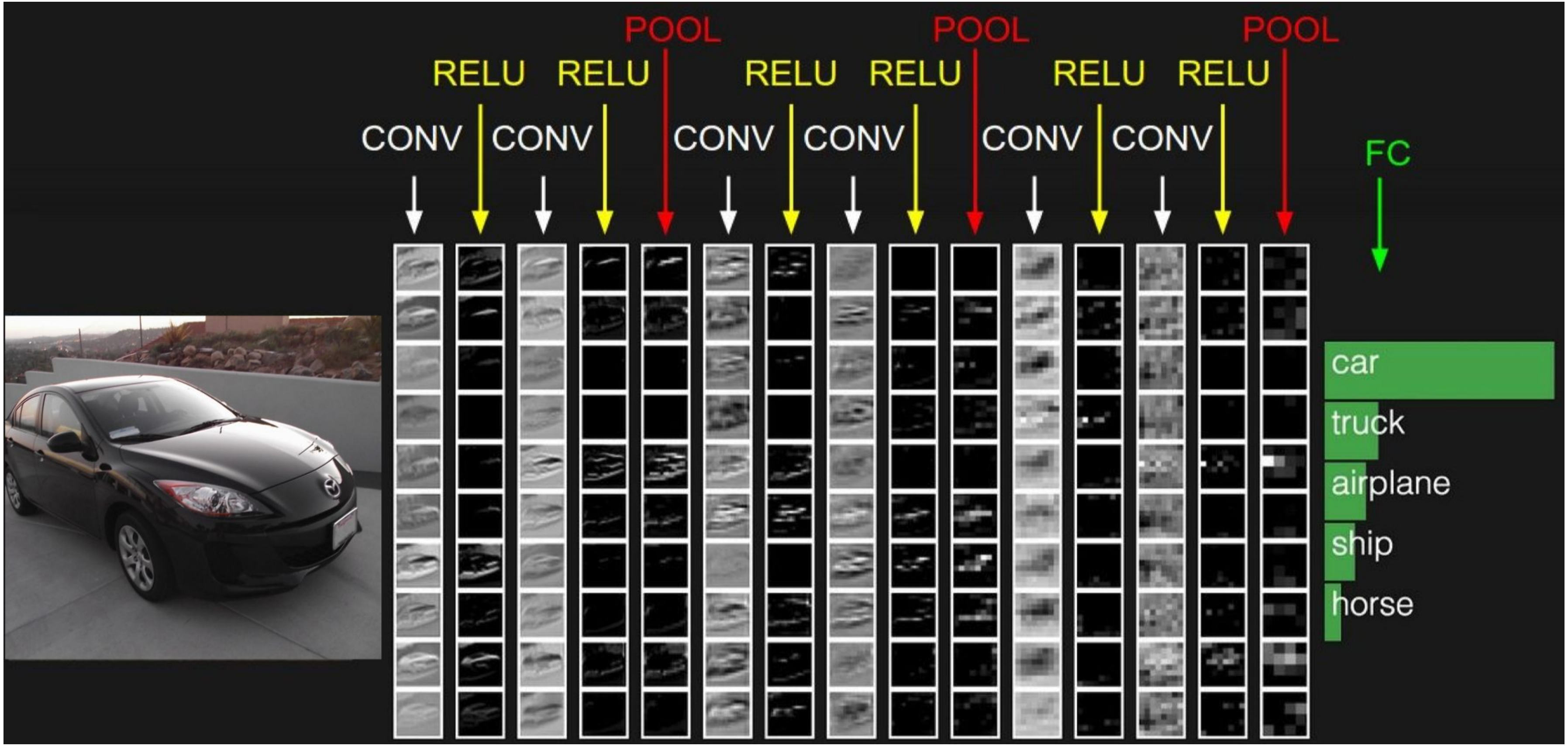
Train/test a CNN on Fashion-MNIST

```python
model.compile(loss="sparse_categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
history = model.fit(X_train, y_train, epochs=10, validation_data=[X_valid, y_valid])
score = model.evaluate(X_test, y_test)
X_new = X_test[:10] # pretend we have new images
y_pred = model.predict(X_new)
```

```
Train on 55000 samples, validate on 5000 samples
Epoch 1/10
55000/55000 [==============================] - 51s 923us/sample - loss: 0.7183 - accuracy: 0.7529 - val_loss: 0.4029 - val_accuracy: 0.8510
Epoch 2/10
55000/55000 [==============================] - 47s 863us/sample - loss: 0.4185 - accuracy: 0.8592 - val_loss: 0.3285 - val_accuracy: 0.8854
Epoch 3/10
55000/55000 [==============================] - 46s 836us/sample - loss: 0.3691 - accuracy: 0.8765 - val_loss: 0.2905 - val_accuracy: 0.8936
Epoch 4/10
55000/55000 [==============================] - 46s 832us/sample - loss: 0.3324 - accuracy: 0.8879 - val_loss: 0.2794 - val_accuracy: 0.8970
Epoch 5/10
55000/55000 [==============================] - 48s 880us/sample - loss: 0.3100 - accuracy: 0.8960 - val_loss: 0.2872 - val_accuracy: 0.8942
Epoch 6/10
55000/55000 [==============================] - 51s 921us/sample - loss: 0.2930 - accuracy: 0.9008 - val_loss: 0.2863 - val_accuracy: 0.8980
Epoch 7/10
55000/55000 [==============================] - 50s 918us/sample - loss: 0.2847 - accuracy: 0.9030 - val_loss: 0.2825 - val_accuracy: 0.8972
Epoch 8/10
55000/55000 [==============================] - 50s 915us/sample - loss: 0.2728 - accuracy: 0.9080 - val_loss: 0.2734 - val_accuracy: 0.8990
Epoch 9/10
55000/55000 [==============================] - 50s 913us/sample - loss: 0.2558 - accuracy: 0.9139 - val_loss: 0.2775 - val_accuracy: 0.9056
Epoch 10/10
55000/55000 [==============================] - 50s 911us/sample - loss: 0.2561 - accuracy: 0.9145 - val_loss: 0.2891 - val_accuracy: 0.9036
10000/10000 [==============================] - 2s 239us/sample - loss: 0.2972 - accuracy: 0.8983
```

This CNN achieves 90% accuracy (not the state-of-the-art, but pretty good!)

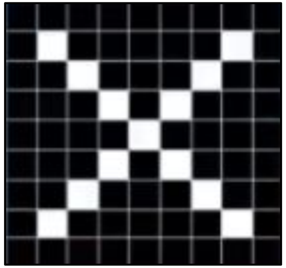
Example of a CNN classifier

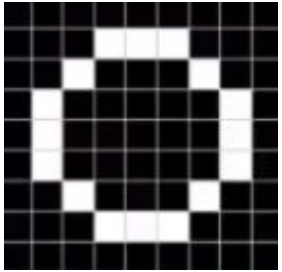


Source: Fei-fei Li's group at Stanford

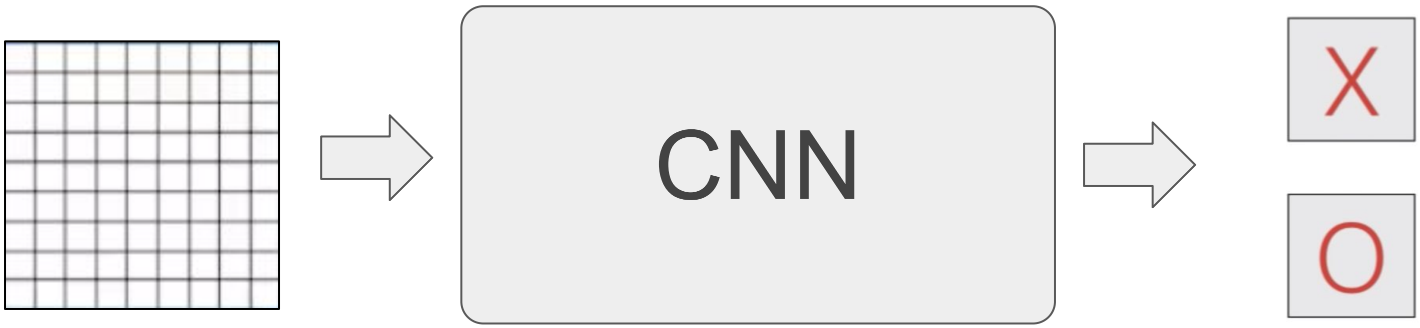
Bonus Content

A toy example on the slides for you to try out!

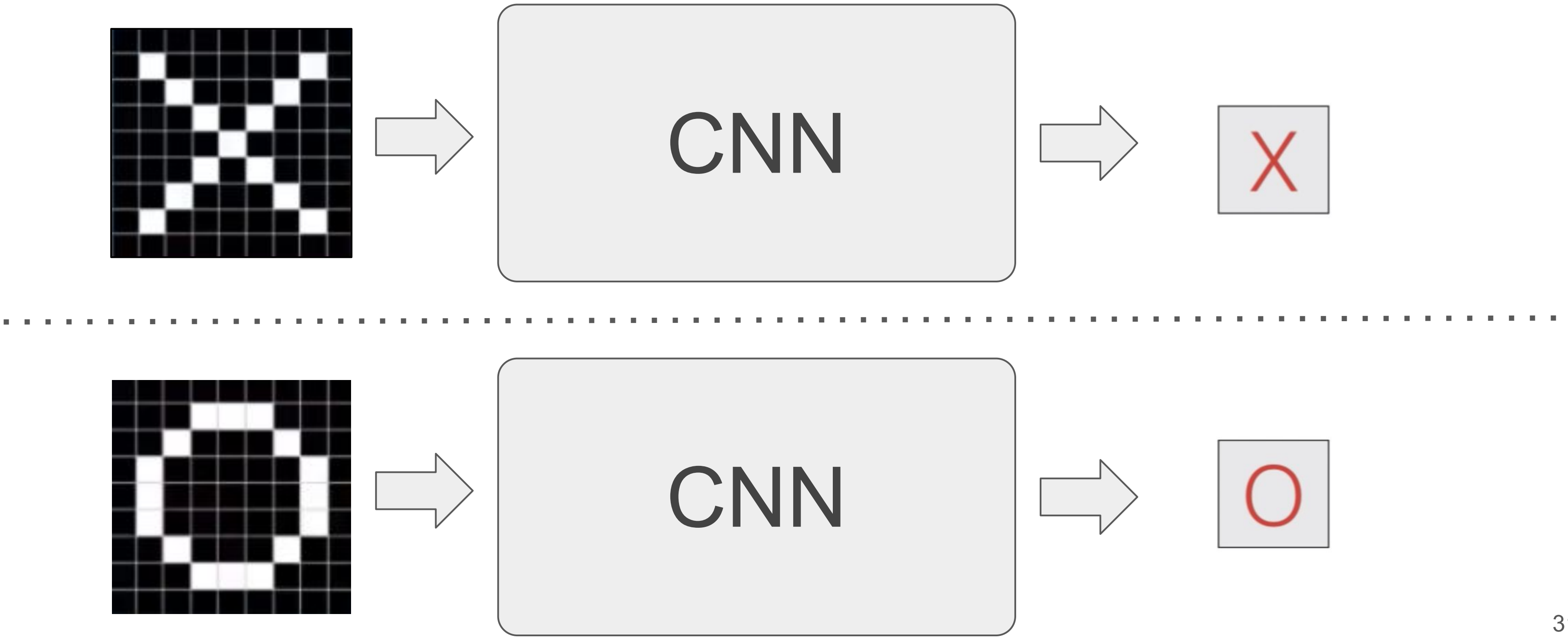




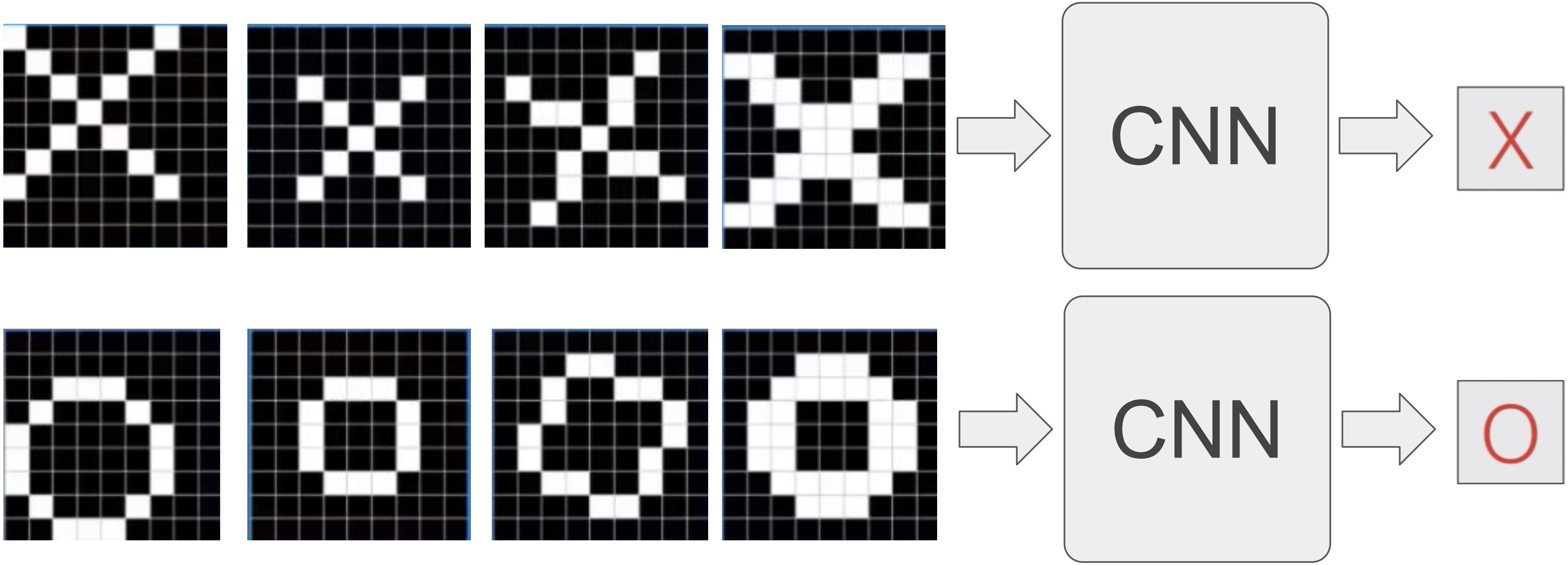
A simple example of classifying X or O



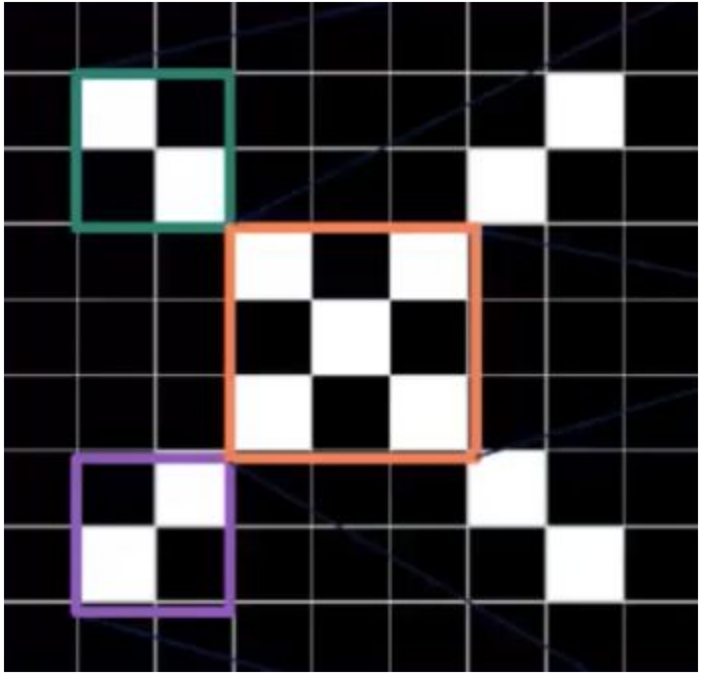
A simple example of classifying X or O

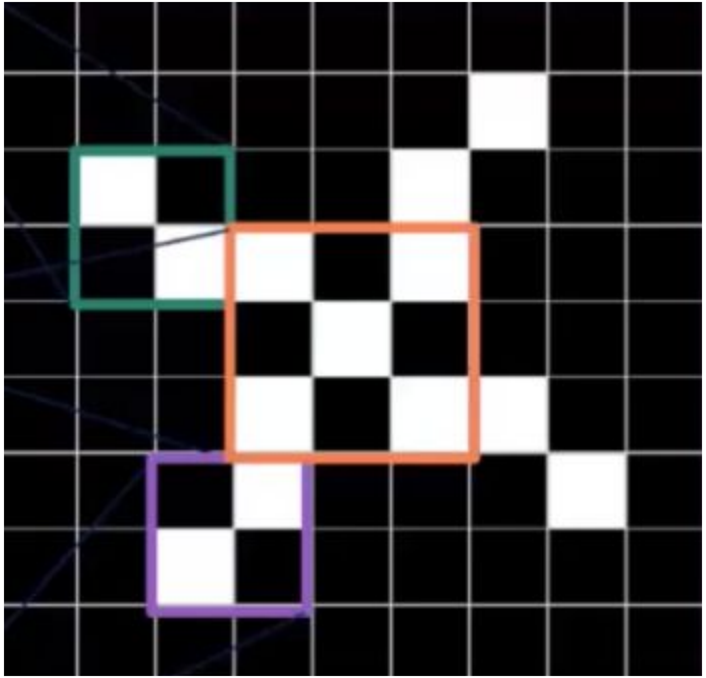


Tricky cases



CNNs match pieces of the image





Kernel match pieces of the image

1	-1	-1
-1	1	-1
-1	-1	1

1	-1	1
-1	1	-1
1	-1	1

-1	-1	1
-1	1	-1
1	-1	-1

Convolution: Trying a match at every location

-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	-1	-1	1	-1	-1	-1	-1
-1	-1	-1	1	-1	1	-1	-1	-1
-1	-1	1	-1	-1	-1	1	-1	-1
-1	1	-1	-1	-1	-1	-1	1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1

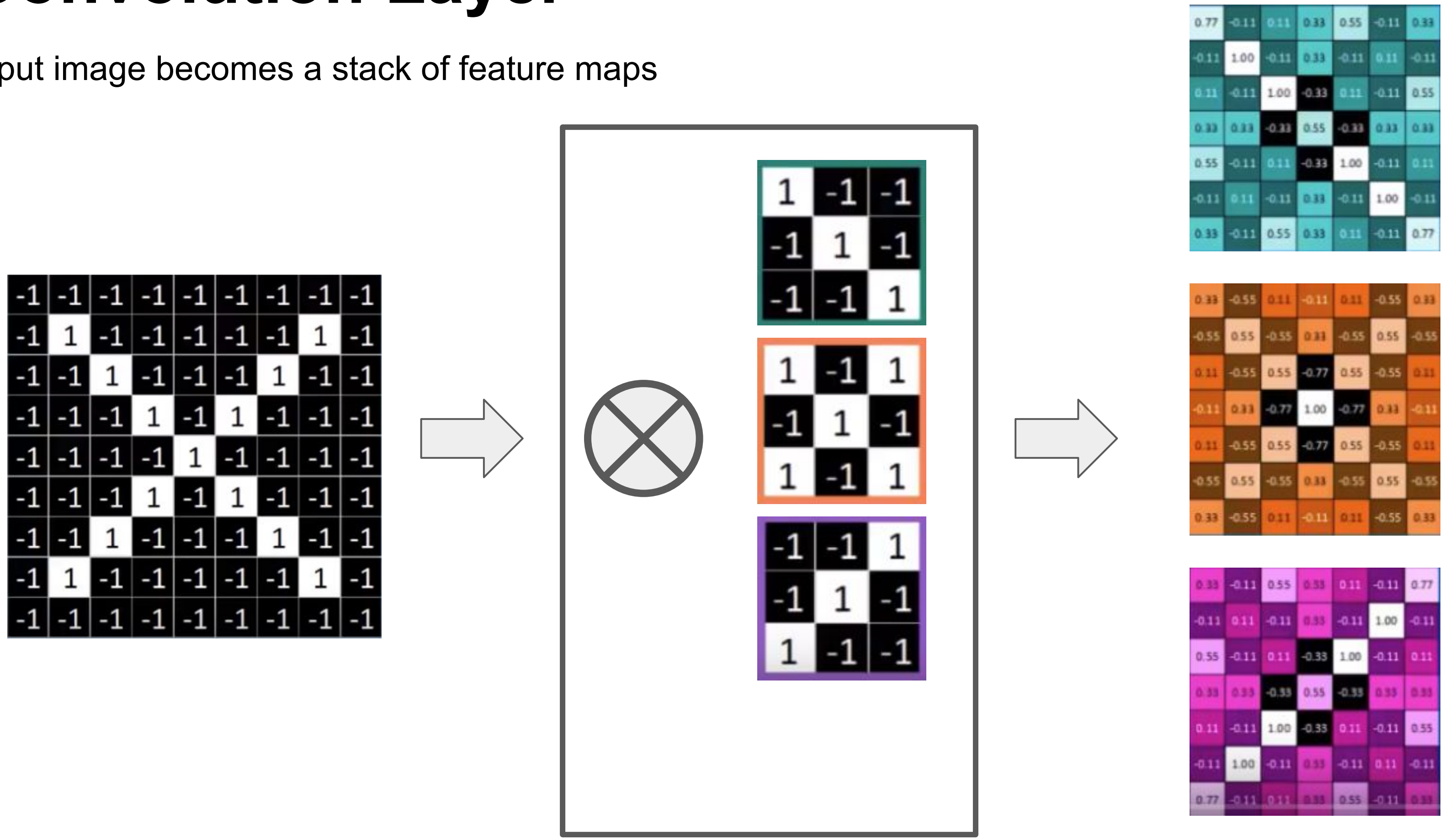
$\otimes$

1	-1	-1
-1	1	-1
-1	-1	1

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

Convolution Layer

Input image becomes a stack of feature maps

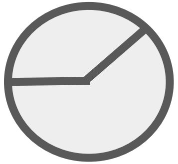


ReLU

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

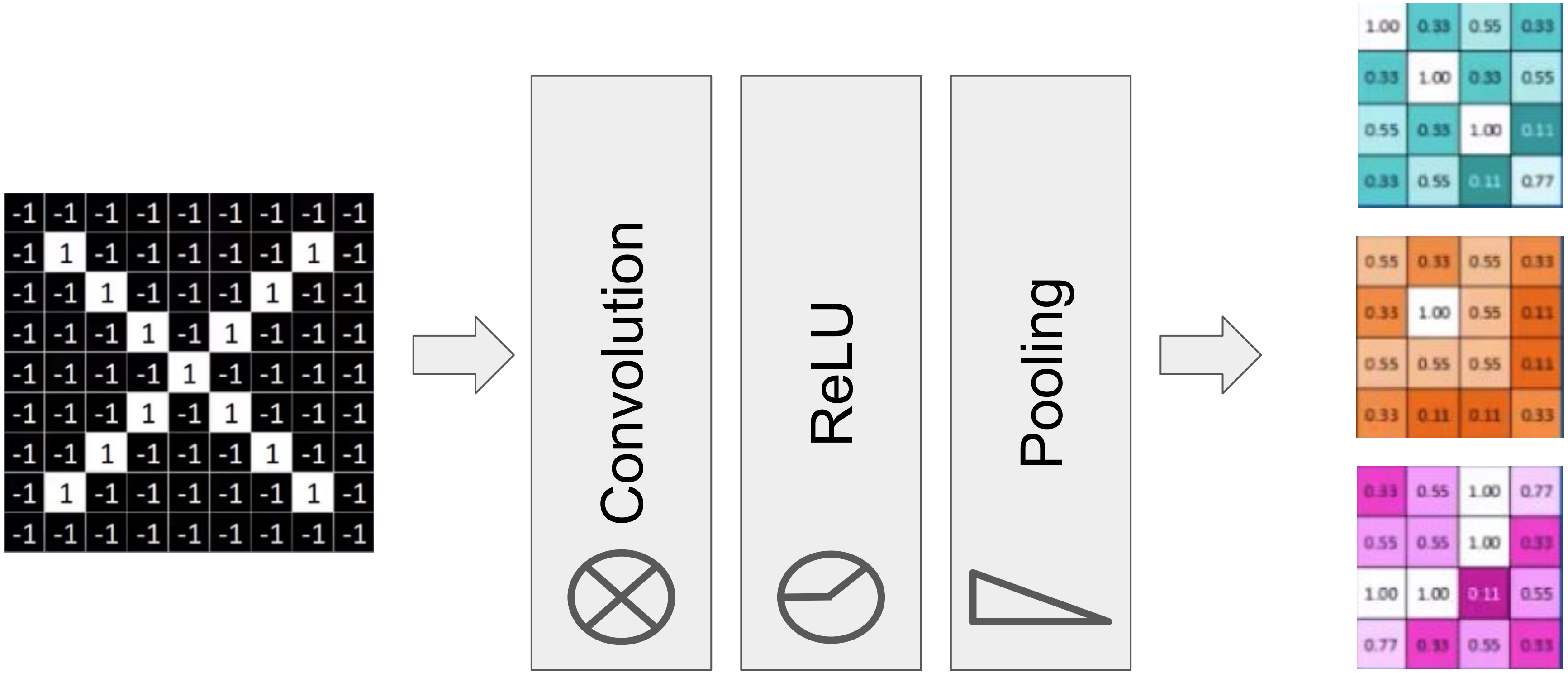
Pooling

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

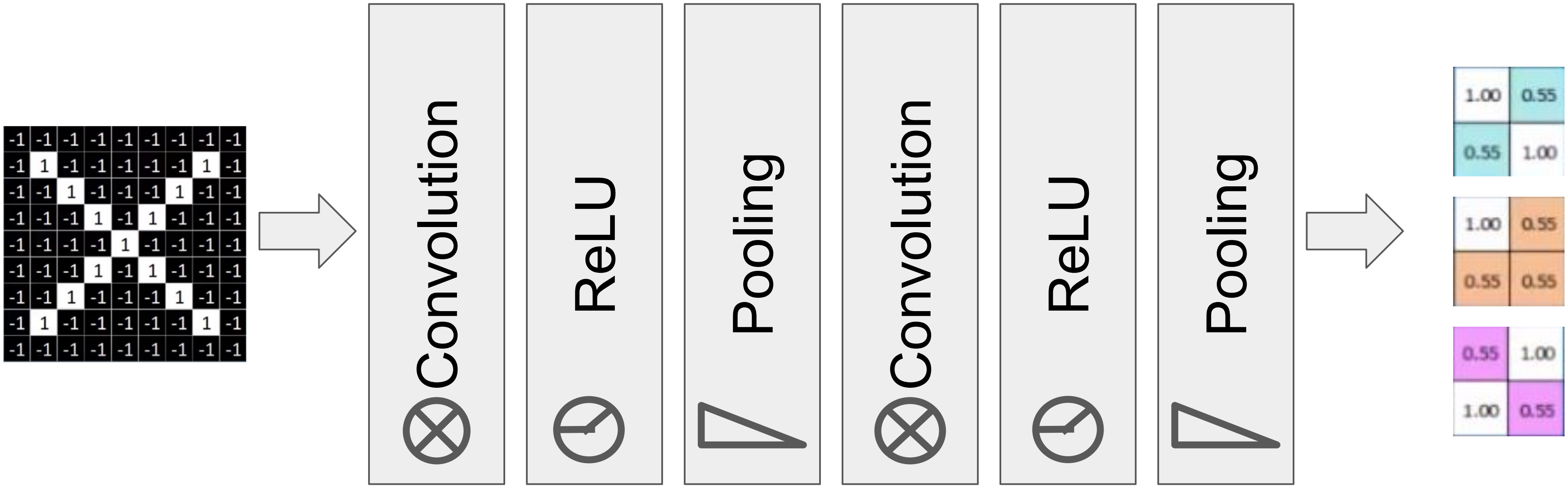
1.00	0.33	0.55	0.33
0.33	1.00	0.33	0.55
0.55	0.33	1.00	0.11
0.33	0.55	0.11	0.77

Layers get stacked



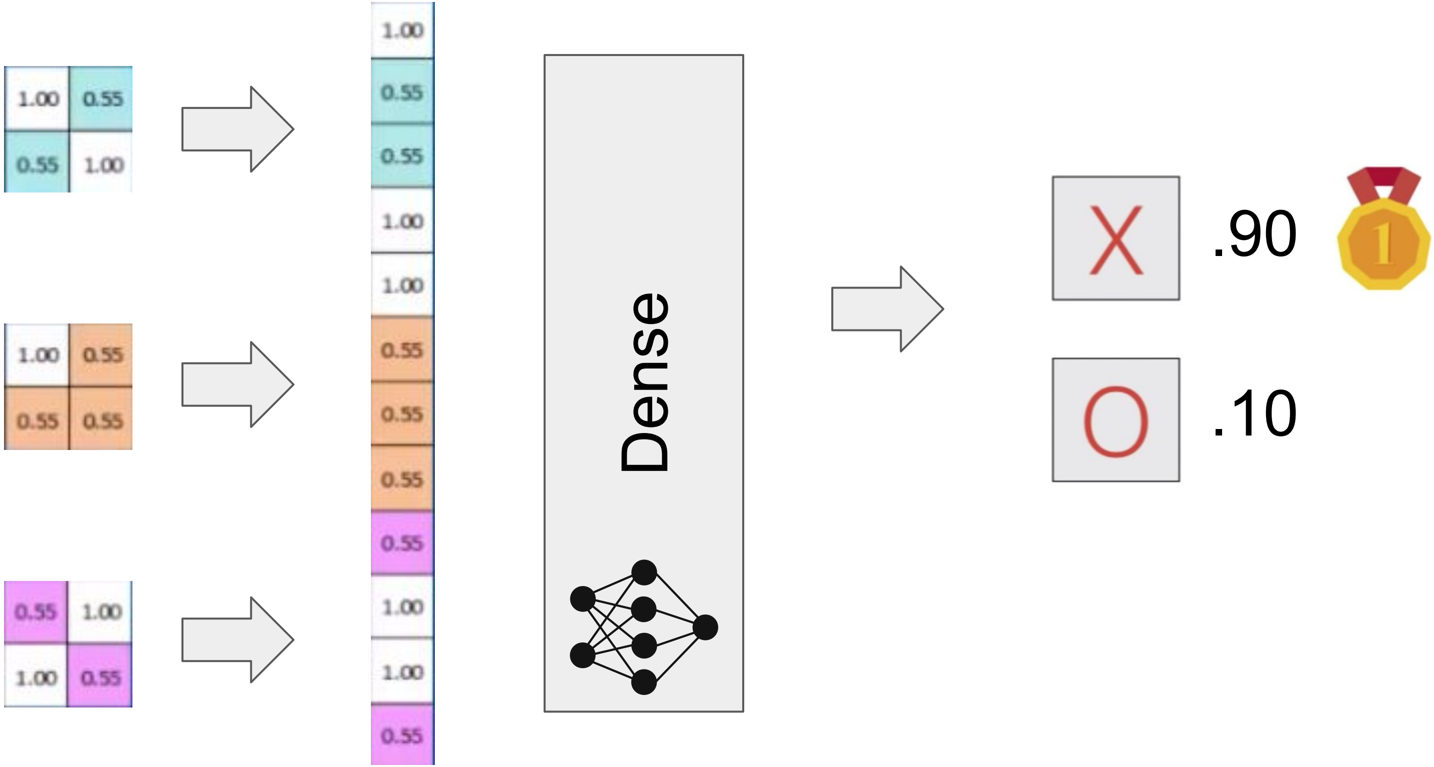
Deep Stacking

Layer operations can be repeated several times



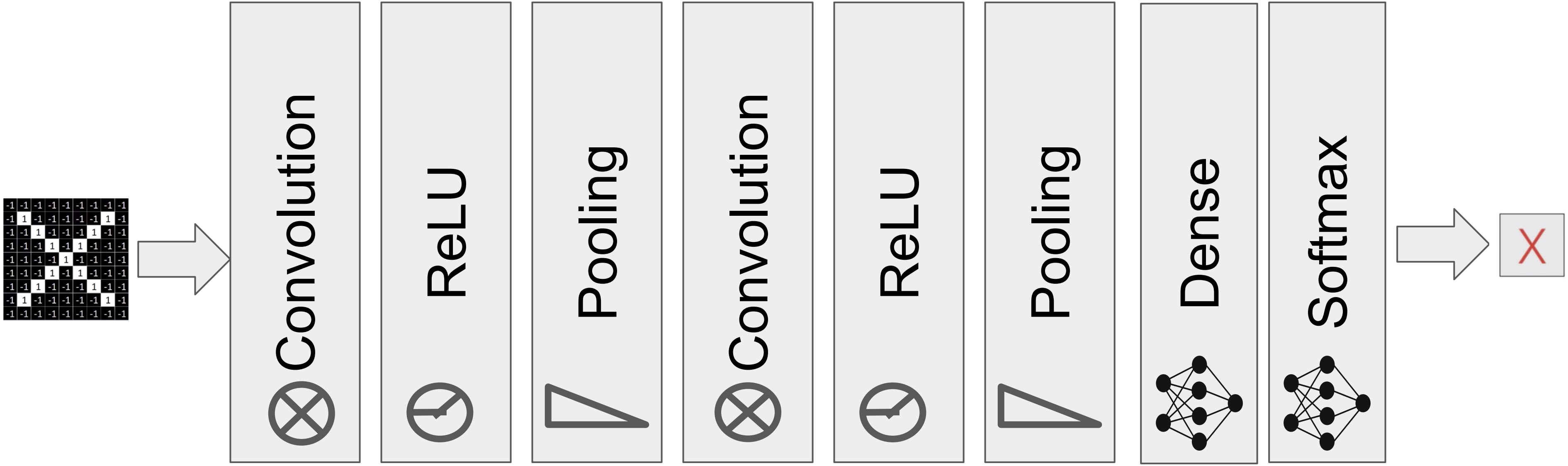
Fully Connected Layer

Every value gets a vote



Putting together in blocks

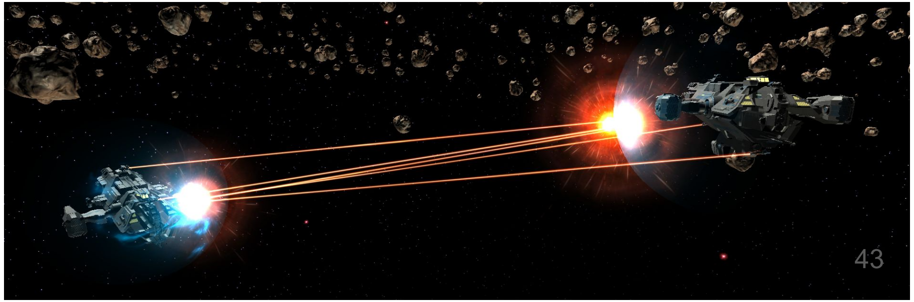
A set of pixels becomes a set of votes



The convolution operation

- Convolution is a operation on two functions of a real-valued argument.
- Suppose we are tracking the location of a **spaceship** at time `t` with a **laser** sensor producing a single measurement `x(t)`.
- Since the laser sensor is noisy, we do a weighted average `w(a)` with more weight on recent measurements and `a` is the age of a measurement:

$$\underbrace{\boxed{s(t)}}_{\text{(weighted) sum}} = \sum_{a=-\infty}^{\infty} \underbrace{\boxed{x(a)}}_{\text{input}}\underbrace{\boxed{w(t-a)}}_{\text{kernel}} = \underbrace{\boxed{(x*w)}}_{\text{convolution}}(t)$$



Convolution operation in ML

The **input** is usually a multidimensional array of **data**

The **kernel** is usually a multidimensional array of **parameters**

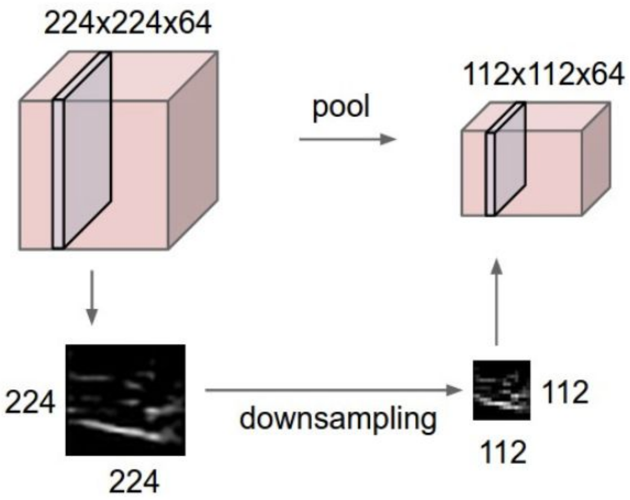
For image data, we use a 2D input image `I`, and 2D kernel `K`:

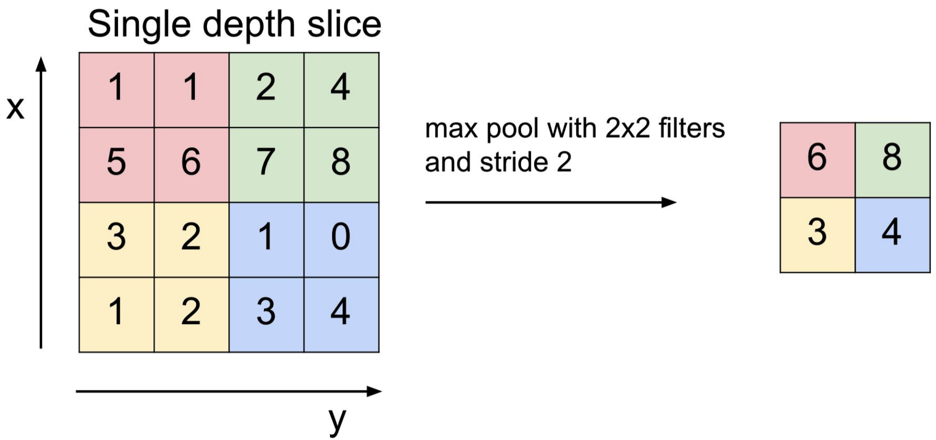
$$S(i,j) = \sum_m \sum_n I(m,n) K(i-m, j-n) = (I * K)(i,j)$$

Many ML libraries implement an *equivalent* function called **cross-correlation**:

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

Max Pooling


224x224x64
pool
112x112x64
224
224
downsampling
112
112


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