BST 261: Data Science II
Lecture 4
Convolutional Neural Networks (CNNs)

Heather Mattie
Harvard T.H. Chan School of Public Health
Spring 2 2019

CNNs

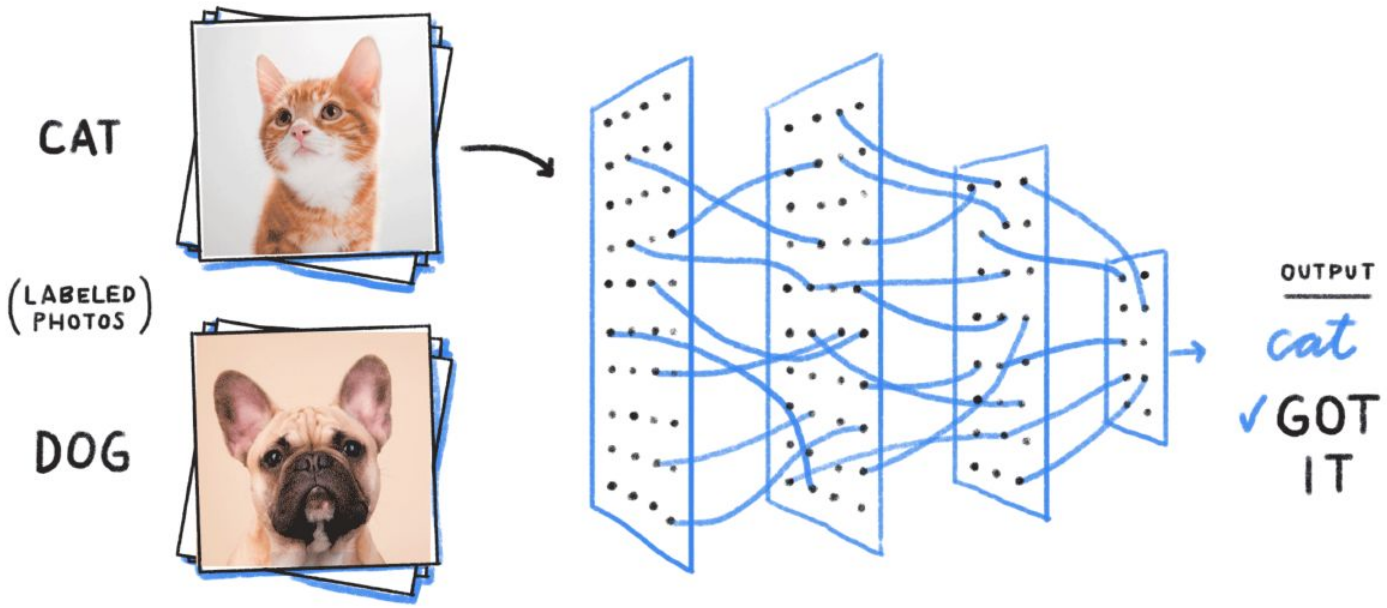
Convolutional neural networks = CNNs = convnets = computer vision

CNNs are at the heart of deep learning, emerging in recent years as the most prominent strain of neural networks in research. They have revolutionized computer vision, achieving state-of-the-art results in many fundamental tasks, as well as making strong progress in natural language processing, computer audition, reinforcement learning, and many other areas. CNNs have been widely deployed by tech companies for many of the new services and features we see today. They have numerous and diverse applications, including:

- Detecting and labeling objects, locations, and people in images
- Converting speech into text and synthesizing audio of natural sounds
- Describing images and videos with natural language
- Tracking roads and navigating around obstacles in autonomous vehicles
- Analyzing video game screens to guide autonomous agents playing them

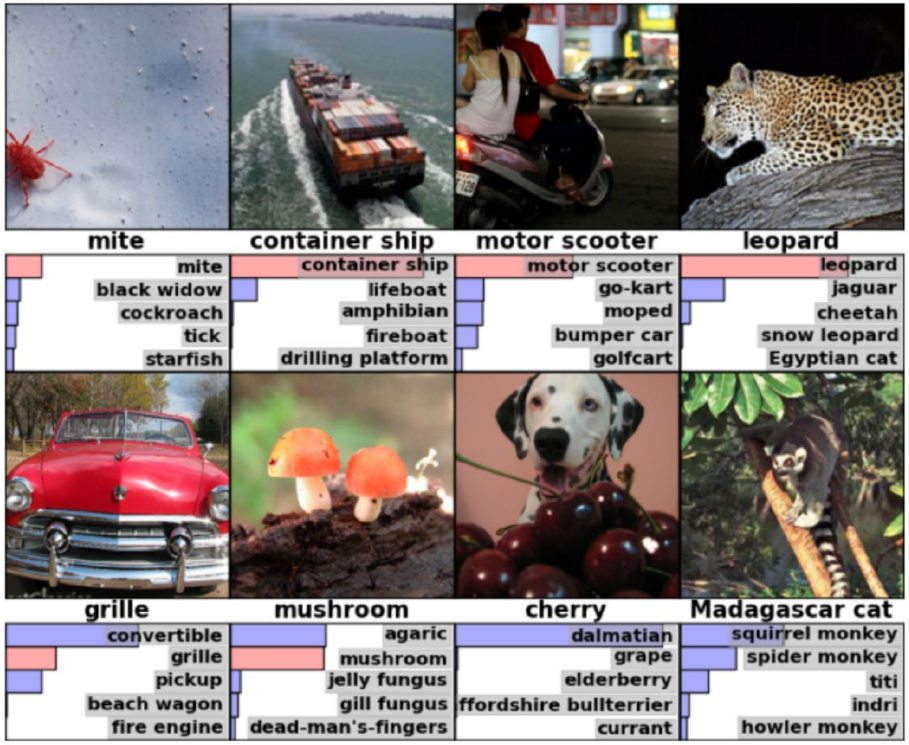
CNN Applications

Image classification



CNN Applications

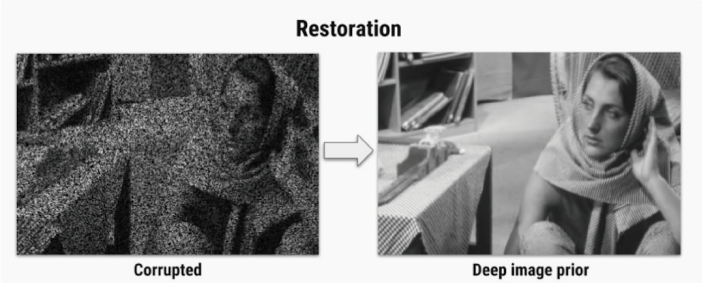
Computer vision

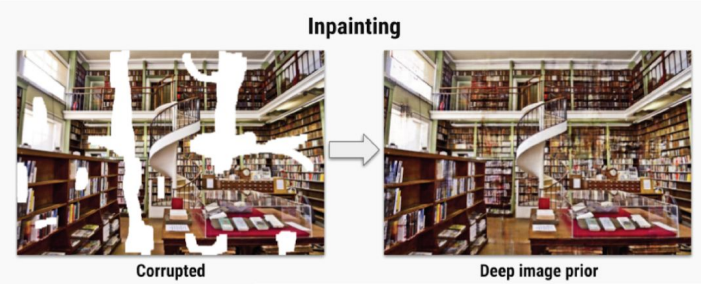


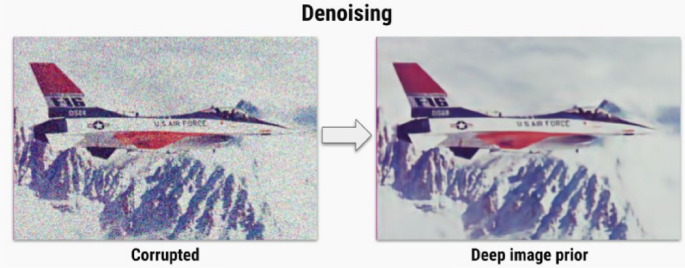
Top: 4 correctly classified examples. Bottom: 4 incorrectly classified examples. Each example has an image, followed by its label, followed by the top 5 guesses with probabilities. From Krizehvsky *et al.* (2012).

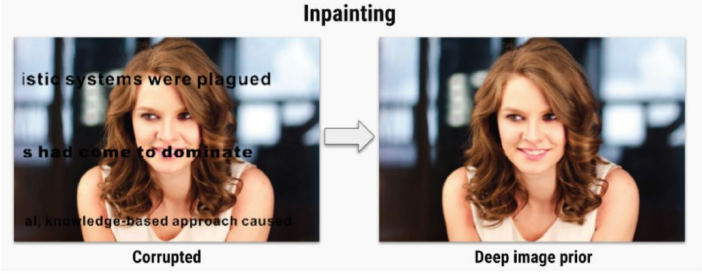
CNN Applications

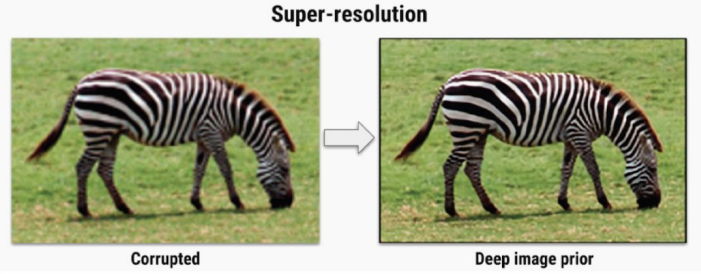
Restoration, Inpainting, Denoising and Super-resolution











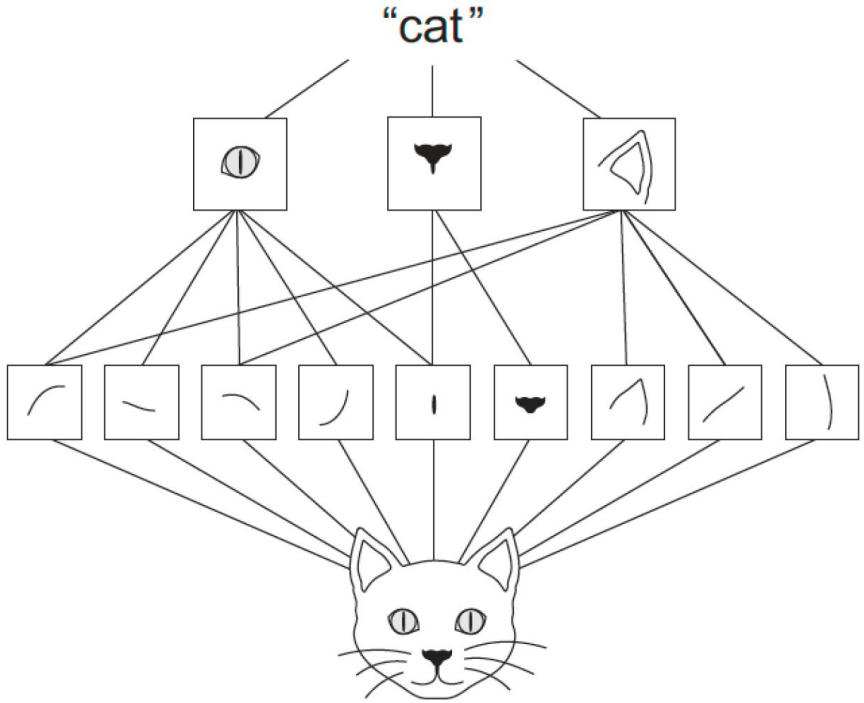
CNNs Applications

Self-driving cars

YouTube video: https://www.youtube.com/watch?v=bDOnn0-4Nq8

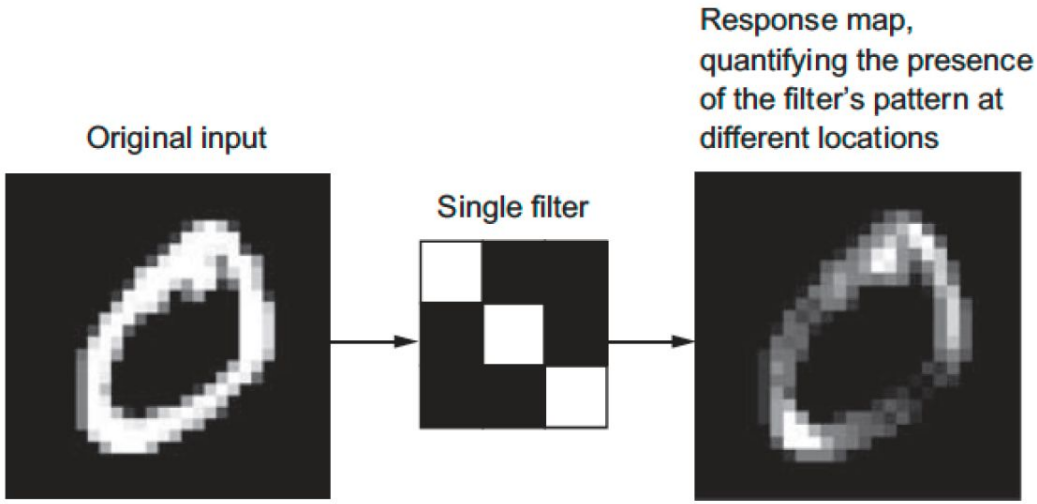
Dense vs Convolutional Layers

- Dense layers learn global patterns
- Convolutional layers learn local patterns
- CNNs learn spacial hierarchies of patterns: one convolutional layer will learn small patterns and the next larger patterns made of the features of the layer before, and so on

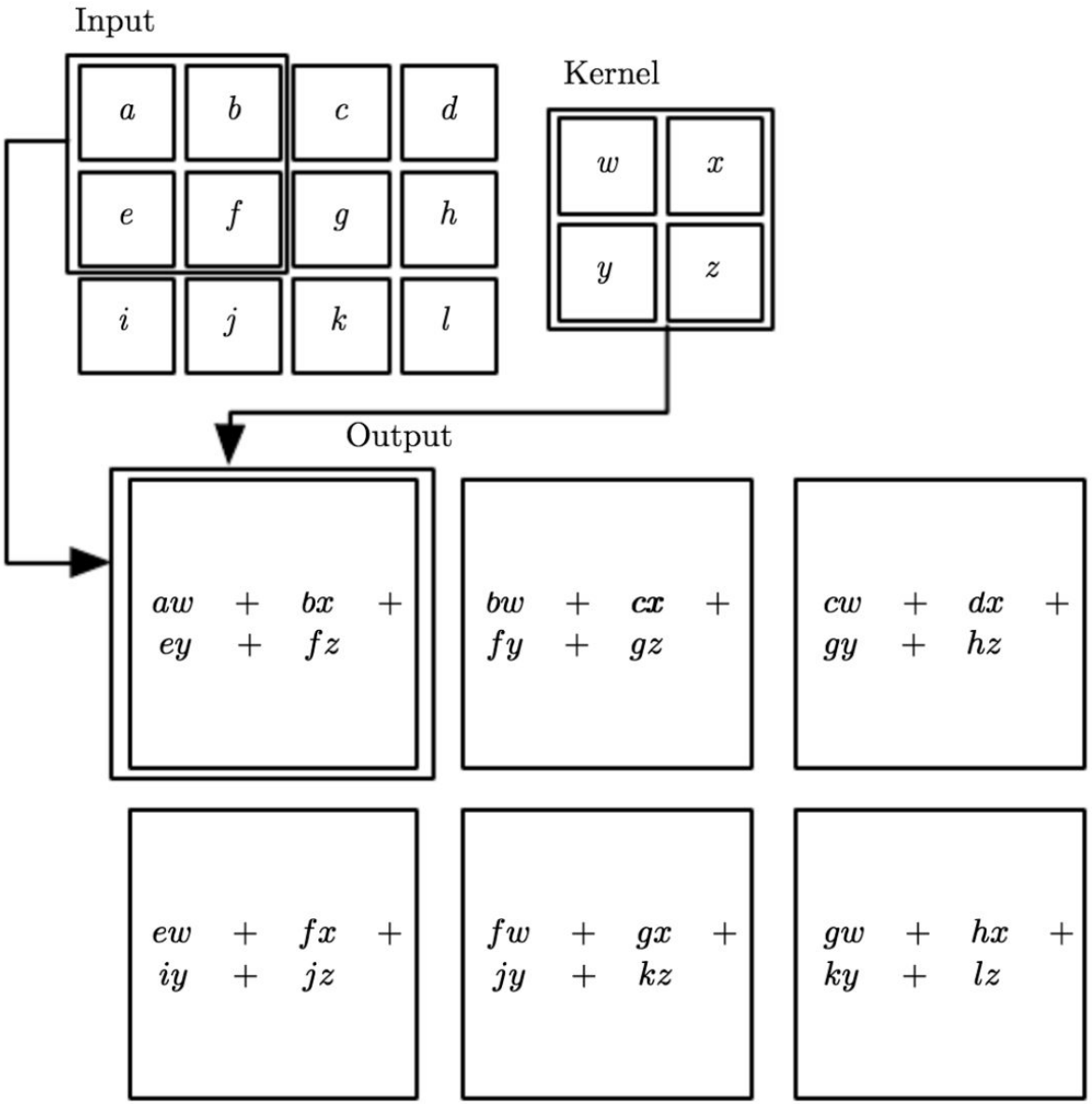


Convolution Layers

- The primary purpose of the convolution operation is to extract features from the input
- To do this, the input is split into several different areas, and the convolution operation applied to each area
- A summary value is then calculated and kept as part of the output
- Here, *I* is the input image (sometimes called **feature map**), *K* is a two-dimensional **kernel**, or more commonly, **filter**, and is a weighting function
- The output resulting from this operation is called the **response map**



The Convolution Operation



Filters

- The most common filter sizes are 3 x 3 and 5 x 5
- Sometimes 7 x 7 filters are also used
- A 1 x 1 filter is a special case that we will see later in the course
- Odd dimension filters are preferred for computer vision
  - Provide natural padding (we'll see this in the following slides)
  - Ensures a central position or pixel of the filter

Convolution

- Convolution leverages 3 important ideas that can help improve a machine learning system:
  - **Sparse interactions** / sparse connectivity / sparse weights
    - Filters are smaller than the input images and thus fewer parameters (weights) need to be stored
    - This reduces computational expense and improves statistical efficiency
  - **Parameter sharing**
    - The same parameter is used for more than one function in the model
    - In a CNN, each each element of the filter is applied to every position of the input

Convolution

- **Equivariant representations**
  - Parameter sharing causes equivariance to translation: if the input changes, the output changes in the same way
  - A function f(x) is invariant to a function g if f(g(x)) = g(f(x))
  - When processing time-series data, this means that convolution produces a sort of timeline that shows when different features appear in the input
  - Note that convolution is not equivariant to some other transformations, such as changes in the scale or rotation of an image

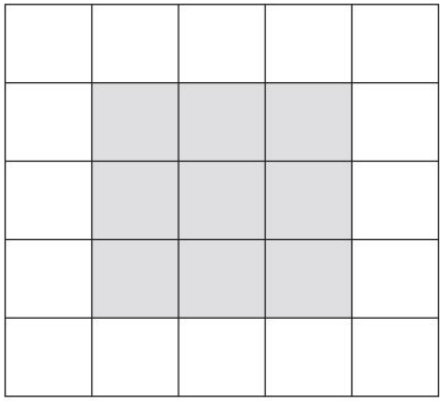
- **Translation invariant**: After learning a certain pattern from one part of an image, it can recognize it anywhere
- Convolution also provides a way to work with inputs of different size

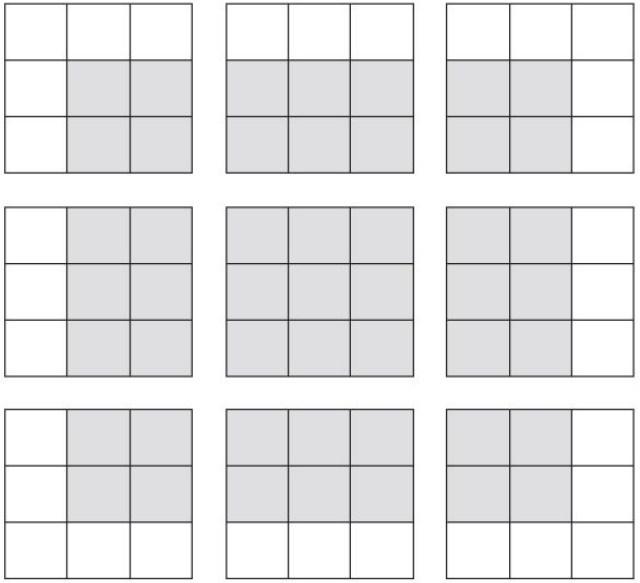
Padding

- Applying a filter to an input image shrinks it - the output dimensions are smaller than the input dimensions
- Additionally, when we perform the convolution operation on an input, the pixels on the edges aren't used as much as the pixels in the middle
- To make the amount of information used more equal across pixels, you can use **padding**
- Padding consists of adding an appropriate number of rows and columns to each side of the image (think a border of pixels)
- This enables an output with the same dimensions as the input

Valid Padding

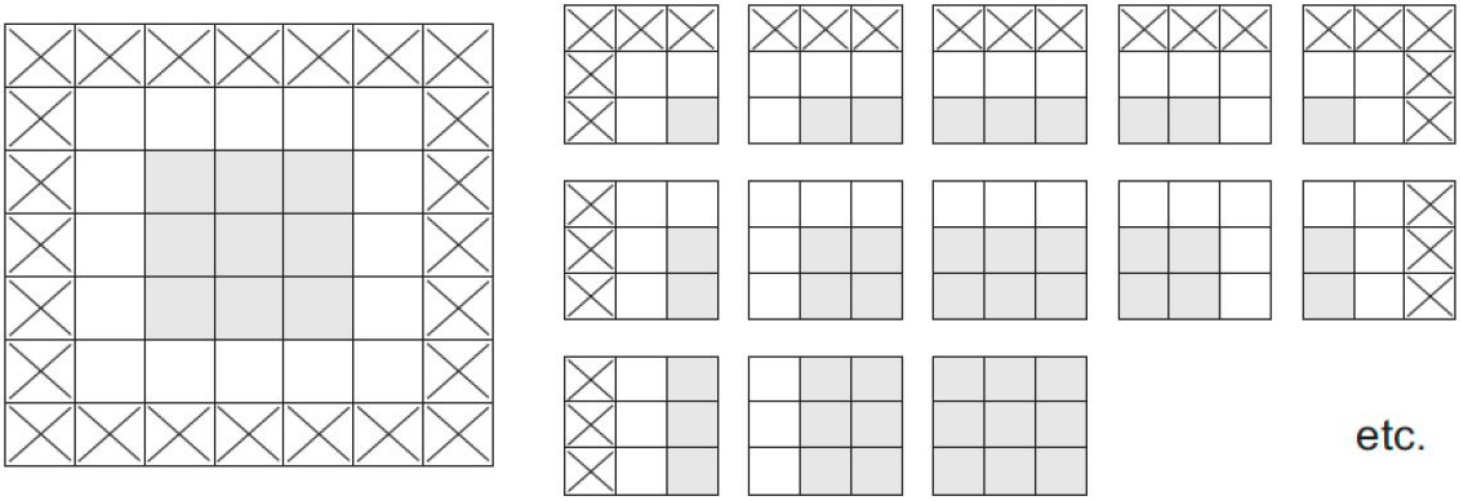
- In Keras, the default is no padding, or **“valid” padding**
- This means the output will not be the same dimension as the input, and will instead depend on the dimension of the input and the size of the filter
- Below is a 5 x 5 image
- If we apply a 3 x 3 filter, the output will also be 3 x3





Same Padding

- Below is the same 5 x 5 image, but with an added border
- If we use a 3 x 3 filter, we need to add $p = 1$ padding - here, p is the number of rows to add to the border of an image
- The output will then be 5 x 5
- Because the input and output have the same dimensions, this is called **"same" padding**



General Padding Formula

- There is a general formula that can help you decide how much padding you need or want
  - Let the input be *n* x *m* and the filter *f* x *f*
  - Without padding, the output would be:

$$(n - f + 1) \times (m - f + 1)$$

- For an output with the same dimension as the input, need *p* such that:

$$n \times m = (n + 2p - f + 1) \times (m + 2p - f + 1)$$

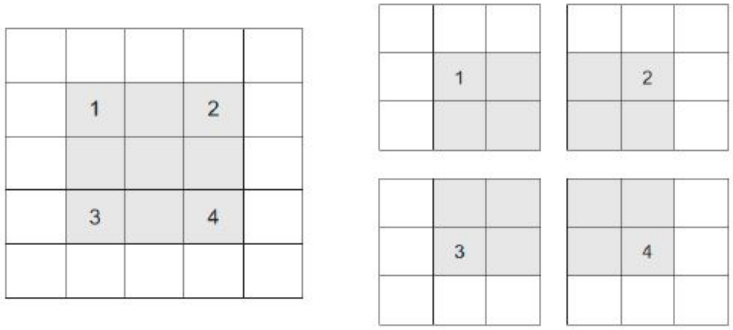
which boils down to

$$p = \frac{f - 1}{2}$$

Convolutional Strides

- Another factor that can influence output size is convolution **strides**
- So far we have assumed that we slide the filter over a single space to extract a new patch - but what if we wanted to slide over 2 spaces, 3 spaces, or more?
- Convolutional strides are convolutions with a stride greater than 1
- If your stride is set equal to 2, you will downsample the width and height of the input image by a factor of 2
- If *s* is the size of the stride, the output will have dimensions

$$\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{m+2p-f}{s} + 1 \right\rfloor$$



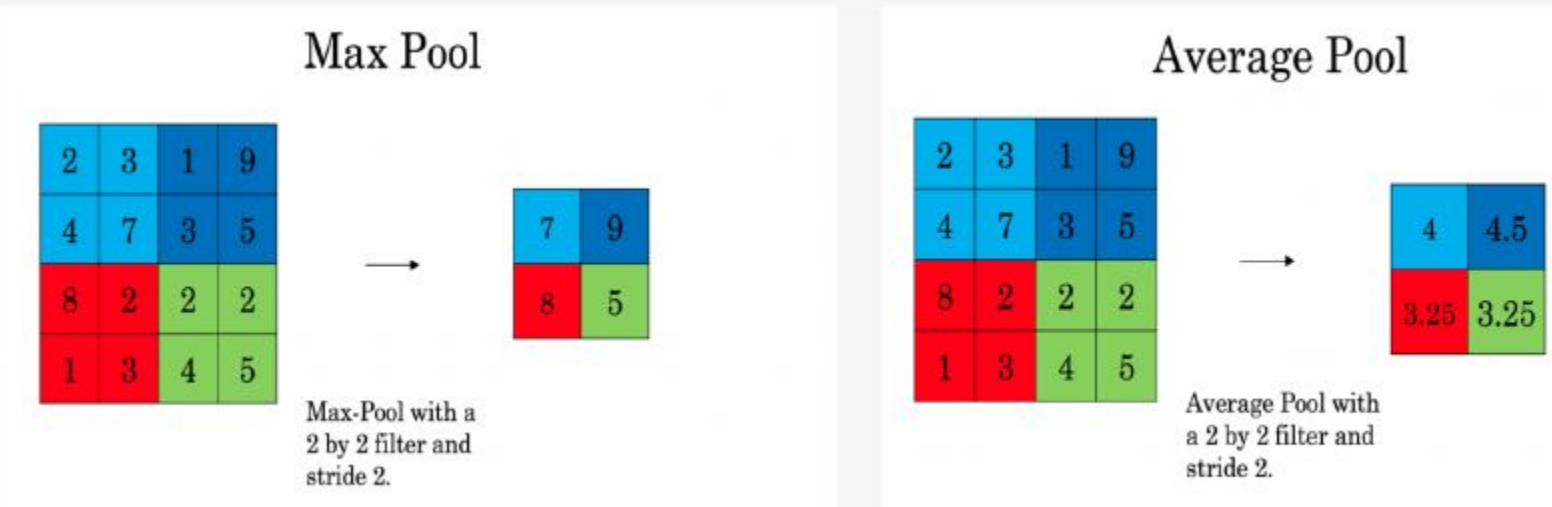
Pooling

- Strided convolutions are rarely used in practice - **max-pooling** is more common
- A typical layer of a CNN consists of 3 stages:
  - Convolution stage - linear transformation of the input
  - Detector stage - Nonlinear activation (ex: relu activation function is applied)
  - Pooling stage - further modification (downsizing) of the output
- A pooling function replaces the output at a certain location with a summary statistic of the nearby outputs
- Pooling greatly reduces the computational expense of the network by decreasing the number of parameters to be learned

Pooling

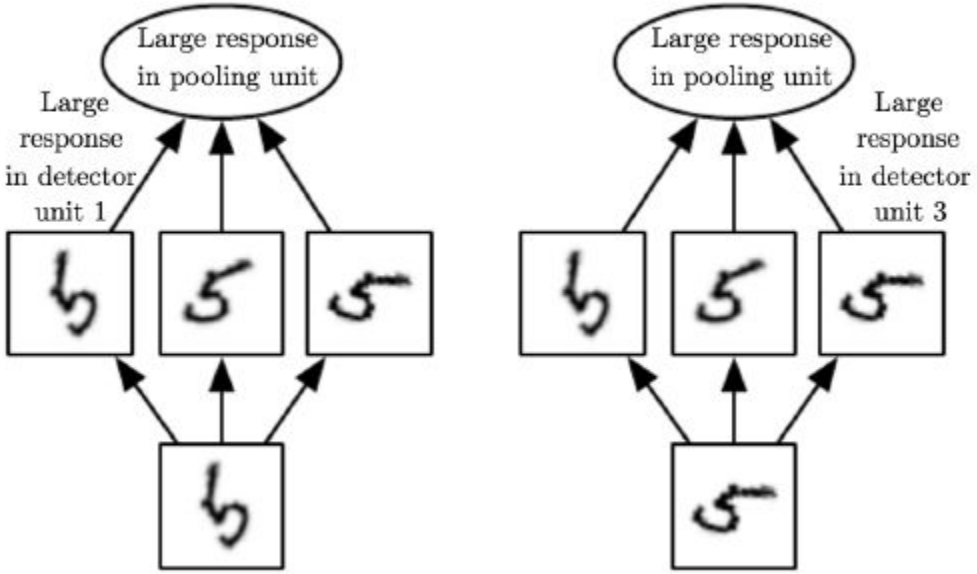
- There are different types of pooling
  - Max Pooling
    - Outputs the maximum value from a patch for each channel
    - Similar to convolution, but instead of transforming patches via a learned linear transformation, they're transformed via a hardcoded max tensor operation
    - Very common
    - Usually done with 2 x 2 windows
  - Average Pooling
  - Weighted Pooling
    - Based on the distance from the central pixel
- Max pooling tends to work better than other pooling functions and convolutional strides

Pooling


Max Pool
2 3 1 9
4 7 3 5
8 2 2 2
1 3 4 5
7 9
8 5
Max-Pool with a 2 by 2 filter and stride 2.
Average Pool
2 3 1 9
4 7 3 5
8 2 2 2
1 3 4 5
4 4.5
3.25 3.25
Average Pool with a 2 by 2 filter and stride 2.

Pooling and Invariance

- A pooling unit that pools over multiple features that are learned with separate parameters can learn to be invariant to transformations of the input
- Here, a set of 3 learned filters and a max pooling unit can learn to become invariant to rotation

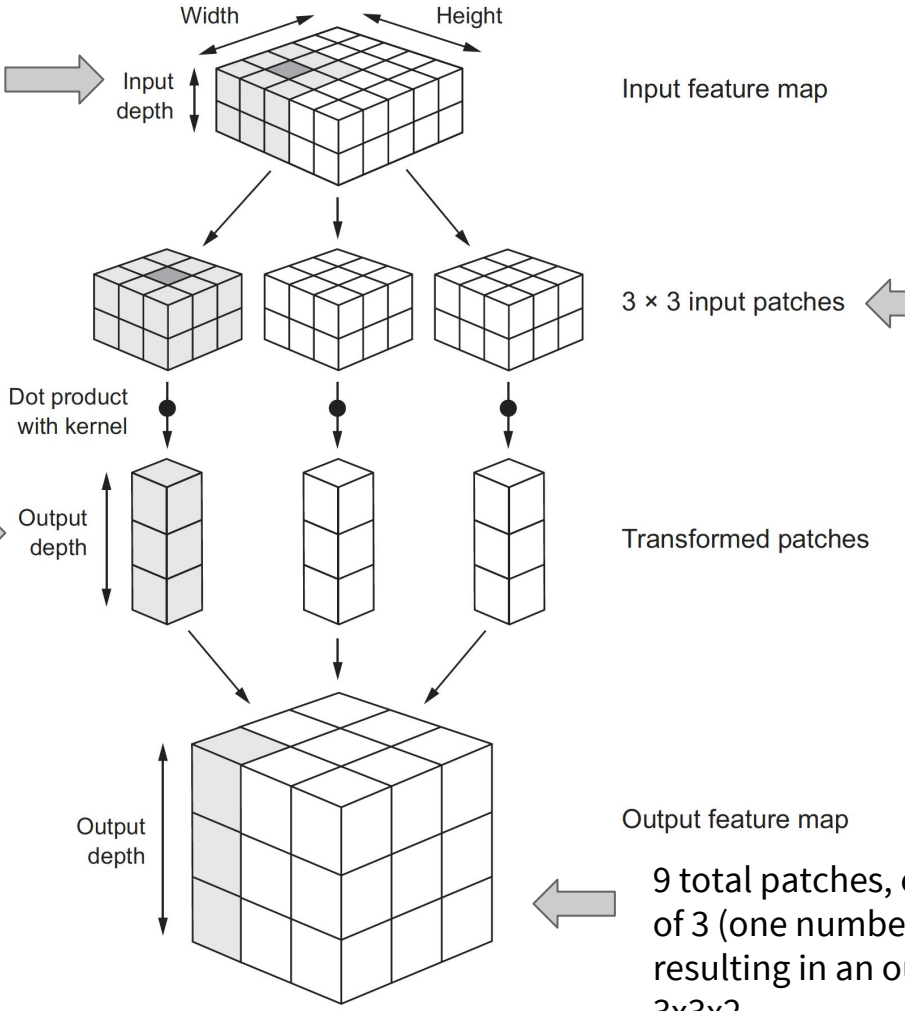


Terminology

- Convolutions operate over 3D tensors with two spatial axes, **height** and **width**, as well as a **depth** axis (or **channels** axis)
- For a color (RGB) image, the depth is equal to 3
- For black and white (grayscale) images, the depth is equal to 1
- The convolution operation extracts different **patches** from the input image and applies the same transformation to to each of them, resulting in a response map that is also a 3D tensor
- The response map has a width, height, and depth, all of which depend on the input image, filter, padding and stride
- Convolutions are defined by 2 key parameters:
  - Size of the filter
  - Depth of the output response map, i.e., how many filters are applied to the input
- Convolution works by sliding the filter over the 3D input image, stopping at every possible location, and extracting the 3D patch at each location
  - Each 3D patch is transformed into a 1D vector
  - All 1D vectors are then reassembled into a 3D output map

Convolution Schematic

Input image has shape 5x5x2

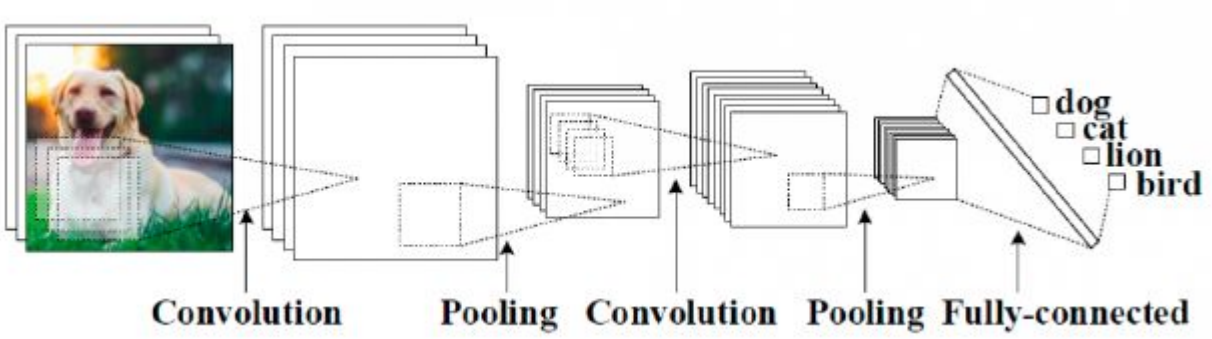


Here, each filter has a shape of 3x3x2, which results in a total of 9 patches from the original input. Only 3 of those patches are shown here.

There are 3 filters (not shown) which results in a depth of 3 for each patch.

9 total patches, each with a depth of 3 (one number for each filter), resulting in an output shape of 3x3x2.

CNN Schematic



MNIST Example

Loading the data and data manipulation don't change:

```
import keras
from keras import layers
from keras import models
from keras.datasets import mnist
from keras.utils import to_categorical
```

```
Using TensorFlow backend.
```

```
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

MNIST Example

```
# Still want a sequential, stack of layers
model = models.Sequential()
# Convolutional layer with 32 filters that are 3x3, relu activation function
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
# Pooling layer with 2x2 windows
model.add(layers.MaxPooling2D((2, 2)))
# Convolutional layer with 64 filters that are 3x3, relu activation function
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
# Pooling layer with 2x2 windows
model.add(layers.MaxPooling2D((2, 2)))
# Convolutional layer with 64 filters that are 3x3, relu activation function
model.add(layers.Conv2D(64, (3, 3), activation='relu'))

# Collapse the 3D tensor
model.add(layers.Flatten())
# Fully connected layer with 64 hidden units, relu activation function
model.add(layers.Dense(64, activation='relu'))
# Softmax output function with 10 classes
model.add(layers.Dense(10, activation='softmax'))

# Same optimizer, loss function and performance measure as before
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=5, batch_size=64)
```

MNIST Example

```
Epoch 1/5
60000/60000 [==============================] - 29s 488us/step - loss: 0.1787 - acc: 0.9436
Epoch 2/5
60000/60000 [==============================] - 30s 493us/step - loss: 0.0483 - acc: 0.9853
Epoch 3/5
60000/60000 [==============================] - 28s 459us/step - loss: 0.0339 - acc: 0.9897
Epoch 4/5
60000/60000 [==============================] - 29s 475us/step - loss: 0.0247 - acc: 0.9927
Epoch 5/5
60000/60000 [==============================] - 28s 474us/step - loss: 0.0192 - acc: 0.9942
```

```
model.evaluate(test_images, test_labels)
```

```
10000/10000 [==============================] - 2s 203us/step
[0.029071162088401797, 0.9921]
```

MNIST Example