How does computer vision work?

Introduction (5 mts)

Business Context. Image data is everywhere. Everyone is carrying high-resolution cameras everywhere they go, taking pictures and video and posting them to the internet. Images and video are a major element of medical diagnoses, surveillance and security, self-driving vehicles, and the internet generally. The world produces astronomically more image data than could ever be seen or analyzed by people. We have only started making sense of that data once we developed tools for machine vision.

Business Problem. Many social networks manage pictures of their users, and want to identify said users in order to classify pictures and better understand the social graph.

Analytical Context. Machine vision is surprisingly easy these days. About 5 years ago, it was a classical talking point to say that computers have a really hard time telling apart something as simple as cats and dogs. This field has moved so fast that even telling apart dozens of breeds of dogs and cats is trivial. In fact, we will show just how trivial it is to tell apart even individual humans. We will train a **convolutional neural network (CNN)** to classify pictures of three Colombian celebrities. Afterwards, we will look into the structure of CNNs to know how we did what we did.

Structure of CNNs (20 mts)

CNNs are one of the most popular neural network architectures. They form the foundation of computer vision, though they can also be used in NLP, recommender systems, and more.

In CNNs, alternating layers of convolution and pooling serve to extract features from an image, starting with basic shapes like lines in particular orientations and soon expanding to complicated

combinations of the basic shapes. In the end, fully-connected layers allow the net to decide a classification category based on the features that have been extracted by earlier layers.

Diagrams of CNNs will show how the original image gets progressively narrower and deeper, as one layer gets convoluted by several kernels (expanding the depth) and then compressed by a pooling layer (reducing the width). What starts out as a wide image with all the original pixels becomes a deep layer of features with less spatial information:

Image source: Applied Deep Learning - Part 4: Convolutional Neural Networks

Here is the actual shape of one famous CNN architecture called VGG 19:

Image source: What exactly does CNN see?

Biological vision (10 mts)

Have you ever wondered how your own vision works? Let's look at the mammalian eye. It is built out of several layers of neurons, ordered backwards because mammalian evolution got stuck in a suboptimal design. The back layers contain receptors, which send their signals to four other layers of neurons in the retina. The last layer collects its signals and sends them to the optical cortex through the optic nerve:

Image source: Computer vision : Human Vision

But these cells are not just transmitting raw light patterns to the brain. The retina is processing what it sees. We first figured this out using cats:

Image source: Computer vision : Human Vision

In [1]:
```
from IPython.display import YouTubeVideo
YouTubeVideo('IOHayh06LJ4', width=800, height=500)
```

Out[1]:

Receptors are detecting photons themselves. The ganglion cells which are a few layers deeper, though, are detecting edges, shadows, and colors. The visual cortex, later on, is also organized

in layers and contains the cells that actually respond to lines of a particular orientation.

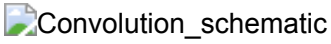
Rather than going into more detail on how the retina does this, however, let's see how we can make a computer do similar processing.

Convolutions (30 mts)

Convolutions are an operation that finds patterns in an image (we also use the word to talk about the result of such an operation).

An image can be represented as a large matrix of numbers (for example, in a black-and-white image, white can be 1 and black can be 0). A convolution consists of scanning it with a smaller matrix called a convolution filter (or **kernel**). As we scan the image with the kernel, we get a single number for each location and put that number into a new matrix called the feature map. Each pixel in the kernel gets multiplied by the corresponding pixel in the image, and the results are summed together. This is all that we need to start detecting edges, or even lines of particular orientations:

Image source: Understanding Convolutional Neural Networks for NLP



Exercise 1: (5 mts)

In the image above, the kernel can't scan all the way to the edge, so each new image is smaller than the previous. What could we do to conserve our image size?

Answer. We can add padding to the edges, extra pixels so that the kernel can slide all the way there. They can be empty, filled with random numbers, or represent a "mirrored" image of the nearest actual pixels in the image. In practice, mirrored pixels are the most common solution:

Image source: Applied Deep Learning - Part 4: Convolutional Neural Networks

Exercise 2: (20 mts)

You've actually probably used convolutions before. They are the basis for all sorts of basic image filtering. For the next exercise, choose an image and apply a few convolutions.

In [2]:
```
import numpy as np
import scipy.ndimage as nd
import matplotlib.pyplot as plt
from skimage import io, color
from skimage import exposure
```

In [3]:
```
img_path = 'img/letterh.jpg'  # Write the path to your image, relative
 to the location of the notebook.
# Note that smaller images might work better than really large ones, be
cause the relative size of the
# kernels will be larger.
```

Let's now load the image and then convert it to grayscale:

In [4]:
```
plt.figure(figsize = (5,5))
img = io.imread(img_path)    # Load the image
plt.imshow(img, cmap=plt.cm.gray);    # plot the image
plt.axis('off');
```

In [5]:
```
plt.figure(figsize = (5,5))
img = color.rgb2gray(img)       # Convert the image to grayscale (1 cha
nnel)
plt.imshow(img, cmap=plt.cm.gray);    # plot the image
plt.axis('off');
```

2.1

How would you design a 3x3 kernel that blurs an image?

Answer. Mechanically, "blurring" is simply the act of making each pixel less distinguishable from other nearby pixels. This suggests that the convolution matrix ought to give some weight to those nearby pixels when calculating the resulting value for a particular pixel. An easy mathematical way of doing that is to re-define each pixel as the average of its neighbors. We can do this with a 3x3 matrix of all 1s:

In [6]:
```
kernel = np.array([[1,1,1],
                   [1,1,1],
                   [1,1,1]])
```

In [7]:
```
image_new = nd.convolve(img, kernel)

fig, axes = plt.subplots(1,2, figsize = (15,10))
axes[0].imshow(img, cmap=plt.cm.gray)
axes[0].set_title('Original', fontsize=24);
axes[0].axis('off');


axes[1].imshow(image_new, cmap=plt.cm.gray)
axes[1].set_title('Convolution', fontsize=24);
axes[1].axis('off');
```

2.2

How would you change the kernel to blur the image even more?

Answer. We should use a bigger kernel. Recall that the blurring kernel takes the average of nearby pixels, but the kernel will consider something nearby if it is within range; i.e. not farther away than the size of the kernel. So we should increase the size of the kernel to weight more nearby pixels:

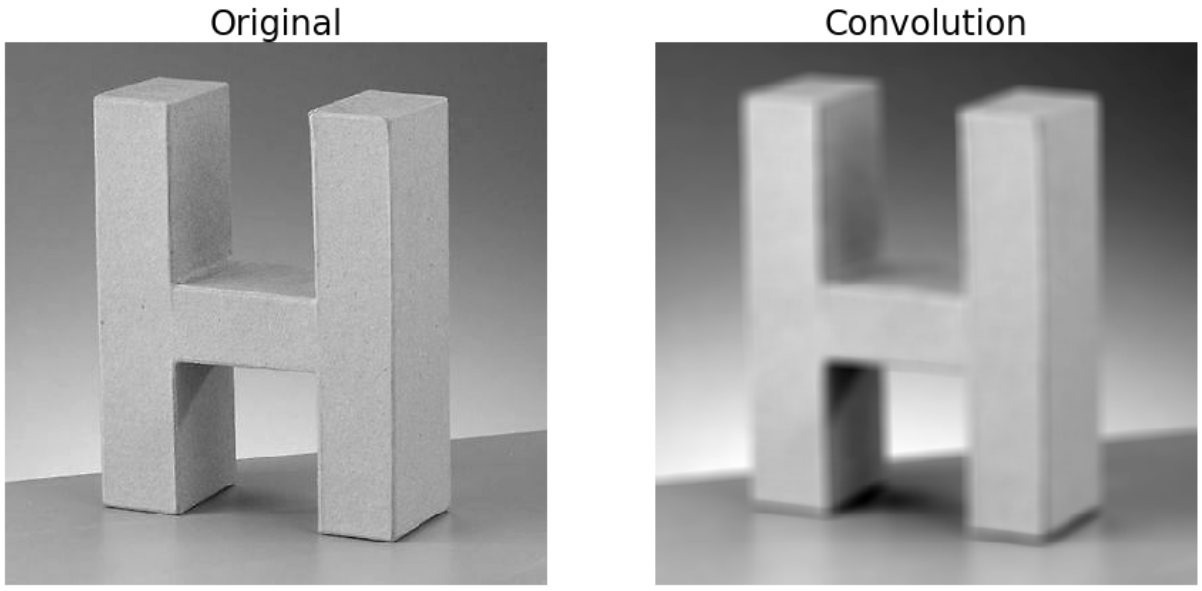
In [8]:
```
kernel = np.array([[1,1,1,1,1,1,1,1,1,1,1],
                   [1,1,1,1,1,1,1,1,1,1,1],
                   [1,1,1,1,1,1,1,1,1,1,1],
                   [1,1,1,1,1,1,1,1,1,1,1],
                   [1,1,1,1,1,1,1,1,1,1,1],
                   [1,1,1,1,1,1,1,1,1,1,1],
                   [1,1,1,1,1,1,1,1,1,1,1],
                   [1,1,1,1,1,1,1,1,1,1,1],
```

```
                    [1,1,1,1,1,1,1,1,1,1,1],
                    [1,1,1,1,1,1,1,1,1,1,1]])
```

In [9]:
```
image_new = nd.convolve(img, kernel)

fig, axes = plt.subplots(1,2, figsize = (15,10))
axes[0].imshow(img, cmap=plt.cm.gray)
axes[0].set_title('Original', fontsize=24);
axes[0].axis('off');


axes[1].imshow(image_new, cmap=plt.cm.gray)
axes[1].set_title('Convolution', fontsize=24);
axes[1].axis('off');
```



2.3

How do we make a 3x3 kernel only notice horizontal edges?

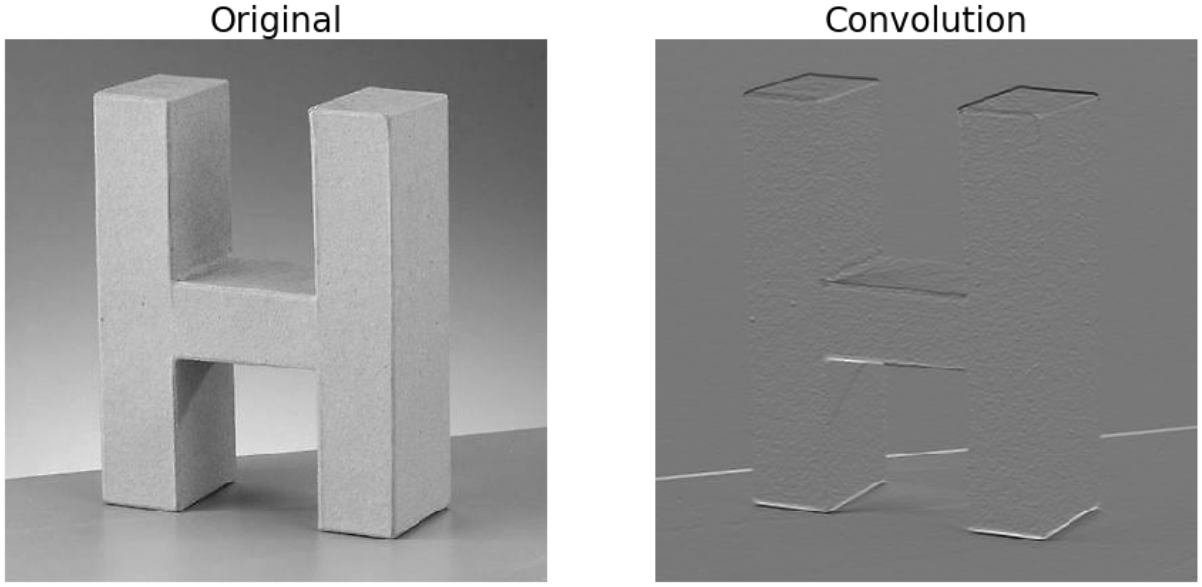
Answer. Since edges often separate two areas of different colors, then in order for the kernel to not register (i.e. give a weighted sum of zero) for the vertical edges the contribution of the entries on the left and right columns of the kernel must each cancel out to zero, while the ones on the top and bottom rows must not. This can be achieved by the following:

In [10]:
```
kernel = np.array([[-1,-1,-1],
                   [0,0,0],
                   [1,1,1]])
```

In [11]:
```
image_new = nd.convolve(img, kernel)

fig, axes = plt.subplots(1,2, figsize = (15,10))
axes[0].imshow(img, cmap=plt.cm.gray)
axes[0].set_title('Original', fontsize=24);
axes[0].axis('off');


axes[1].imshow(image_new, cmap=plt.cm.gray)
axes[1].set_title('Convolution', fontsize=24);
axes[1].axis('off');
```



2.4

How do we make a 3x3 kernel only notice vertical edges?

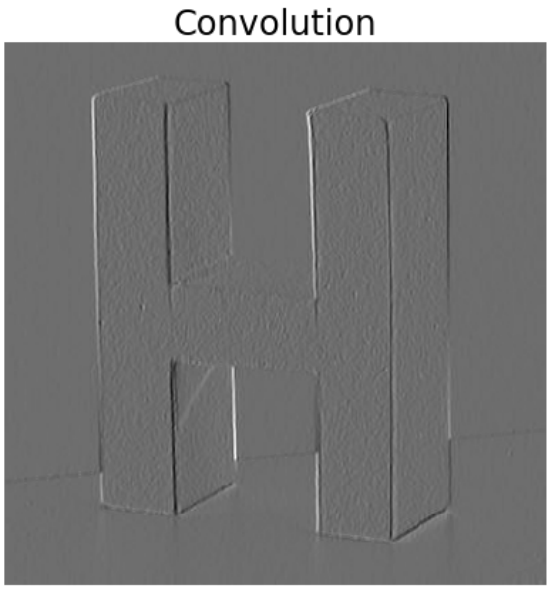
Answer. This is exactly the same logic as the previous exercise, except that we rotate the matrix so that the 1s and -1s run vertically:

In [12]:
```
kernel = np.array([[1,0,-1],
                   [1,0,-1],
                   [1,0,-1]])
```

In [13]:
```
image_new = nd.convolve(img, kernel)

fig, axes = plt.subplots(1,2, figsize = (15,10))
axes[0].imshow(img, cmap=plt.cm.gray)
axes[0].set_title('Original', fontsize=24);
axes[0].axis('off');


axes[1].imshow(image_new, cmap=plt.cm.gray)
axes[1].set_title('Convolution', fontsize=24);
axes[1].axis('off');
```



2.5

How would you design a 3x3 kernel so that it gives a result of 0 when sliding over solid color areas, but gives higher numbers when sliding over an edge (i.e. the kernel detects all edges but only edges)?

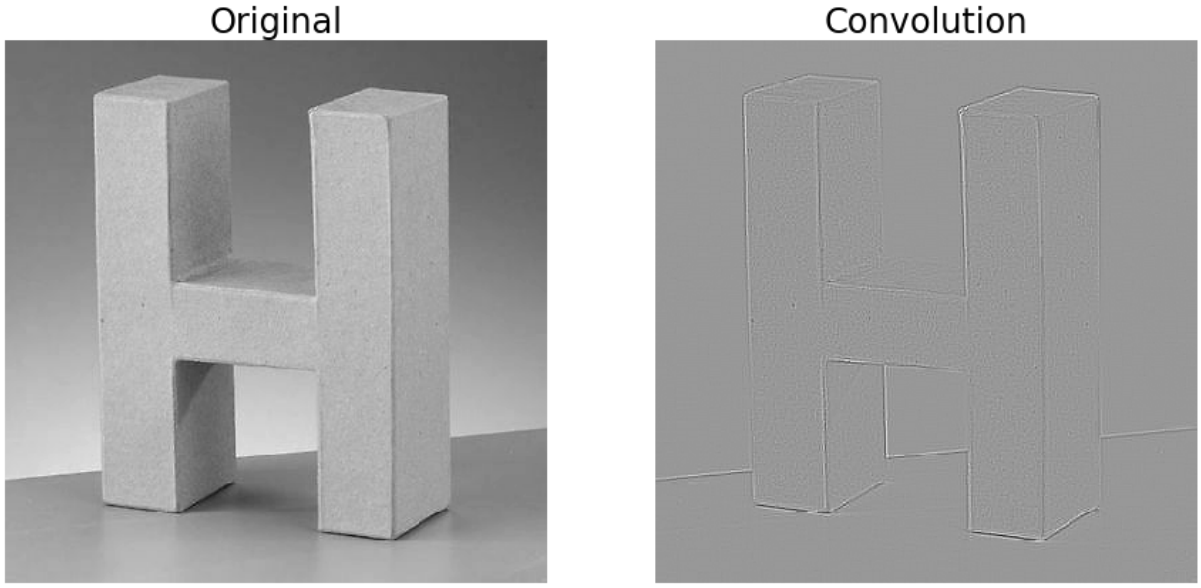
Answer. Since we need to get a sum of 0 over solid color areas, we ought to make sure that the sum of the entries of the 3x3 kernel is zero (so that all the contributions of equal solid colors net out). However, an edge pixel will be of a different color than the surrounding pixels, so we need to weight it as heavily as possible to differentiate it from the nearby pixels and ensure that the other pixels don't cancel it out. This can be done by assigning a very large positive number to the center entry and small negative numbers to the surrounding entries:

In [14]:
```
# Answer: Edge detection kernel
kernel = np.array([[-1,-1,-1],
                   [-1,8,-1],
                   [-1,-1,-1]])
```

In [15]:
```
image_new = nd.convolve(img, kernel)

fig, axes = plt.subplots(1,2, figsize = (15,10))
axes[0].imshow(img, cmap=plt.cm.gray)
axes[0].set_title('Original', fontsize=24);
axes[0].axis('off');


axes[1].imshow(image_new, cmap=plt.cm.gray)
axes[1].set_title('Convolution', fontsize=24);
axes[1].axis('off');
```



Convolutions in neural networks (20 mts)

Each of the convolutions above are extracting features from the image. For every location, the convolution tells us whether or not a feature is present in the area around that location (within the size of the kernel). In the first layers the only patterns that you can detect are things like edges or lines with a particular orientation, but subsequent layers can combine whatever the previous layers found.

Convolutions don't have to just be in 2D. Here is what a 3D convolution looks like. We would use this for color images:

Image source: A Beginner's Guide to Convolutional Neural Networks (CNNs)

Moreover, we usually perform several convolutions and then stack the results together into deeper and deeper layers:

Image source: Applied Deep Learning - Part 4: Convolutional Neural Networks


Pooling (10 mts)

Once we have detected several features, we don't actually care about their precise location in the image. We want to preserve some spatial information, but we care more about the presence or absence of a feature and its *relative* location to others.

One way to get rid of all that extra spatial information is simply to **pool** the layer. This means dividing the layer into non-overlapping squares and performing an operation that aggregates them together. The most common operation is simply to output the maximum value found within that region. This turns a large matrix into a much smaller matrix with the maximum values of the previous.

By reducing the spatial information, we have fewer computations to perform and fewer parameters that could lead to overfitting. Each pooling step makes the image narrower, while preserving depth:

Image source: What exactly does CNN see?

From this, the architecture of VGG 19 starts to make more sense. As with the retina, we start out

with a layer that just detects the levels of light at all points in a surface. We then use convolutions to extract features, and pool the features to get rid of unnecessary spatial information. We do this for several iterations, until we have a narrow, deep set of extracted features. Those features then go into several fully connected layers, which in the end produce a final decision for the model. This is different from biological vision at the detailed level, but the basic architecture is the same and so is the net effect. The CNN can understand lots of high level features, much like a person can:

Image source: What exactly does CNN see?

Data augmentation (5 mts)

One last trick to know about is **data augmentation**. It's always good to train your network on a wide variety of images. Whatever image set you have, you can effectively expand it by an order of magnitude simply by creating variants of your images. Apply various rotations, croppings, and distortions to simulate how real images would vary. The result is a much more general CNN, which can detect features even when they are distorted in real life for whatever reason:

Image source: Applied Deep Learning - Part 4: Convolutional Neural Networks

Starting image

What does a CNN see? (10 mts)

The first layers in a CNN can only see really simple features, like the ones that we saw earlier (e.g. edges, horizontal lines). As we go deeper into the CNN, each layer can combine the features detected in previous layers into ever more complicated features. Straght edges become corners, curves, and repeating patterns. Those patterns in turn soon become eyes and hands.

In the following collage, we see pictures that cause particularly high activation in the feature maps of a fully trained CNN, along with a representation of what that feature map activation looks like. We can think of it as what the network noticed in each image:

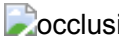
Layer 1

Layer 2

Layer 3

Source: Visualizing and Understanding Convolutional Networks

Another way to know what the CNN is noticing is to hide parts of the image with a gray block, and see which pixels are important for correct classification. If you slide the gray block around, you can see in blue the parts of the image that were essential for good classification:



Source: Visualizing what ConvNets learn

Other CNN architectures (5 mts)

Over the years, different individuals have developed variants of the basic CNN architecture. We saw the details of VGG-19 above, and in the following demonstration we'll use a pre-trained ResNet34. This is U-Net, an architecture that has proven really useful for biomedical image segmentation:

Image source: U-Net: Convolutional Networks for Biomedical Image Segmentation

 Here is a comparison of several different architectures for image recognition in general. Image source: Neural Network Architectures

Conclusions (5 mts)

In this case, we look at the internals of one of the most useful deep learning architectures out there. We learned how to generate our own convolutions to detect simple features, and learned how a CNN synthesizes these basic inputs to generate more and more complex features as it goes deeper into the layers. We then used powerful pre-trained CNN models to solve custom problems in machine vision.

Takeaways (5 mts)

Just a few years ago, computers couldn't tell the difference between cats and dogs. These days, even for a simple demonstration with off-the-shelf parts, it's possible to tell apart specific human celebrities from one another. That's what what we call progress.

 birds.png