Computer Vision in Industry

Spatially Coherent Data

One cool thing about computer vision, is that the techniques that we will learn about, need not only be used with camera images - but also images created with other sensors. So those techniques that you will learn, will be useful for any data, that has what we call, “spatial coherency”.

And spatially coherent data can be thought of as any data that predictably varies over space, like sound, for example. If you hear sound from a speaker close up it will sound very loud, but the farther you get away, the softer the sound will get. And so the volume of a sound can give you spatial information!

Examples of Computer Vision Applications

In general, computer vision is used in many applications to recognize objects and their behavior. Below are some examples.

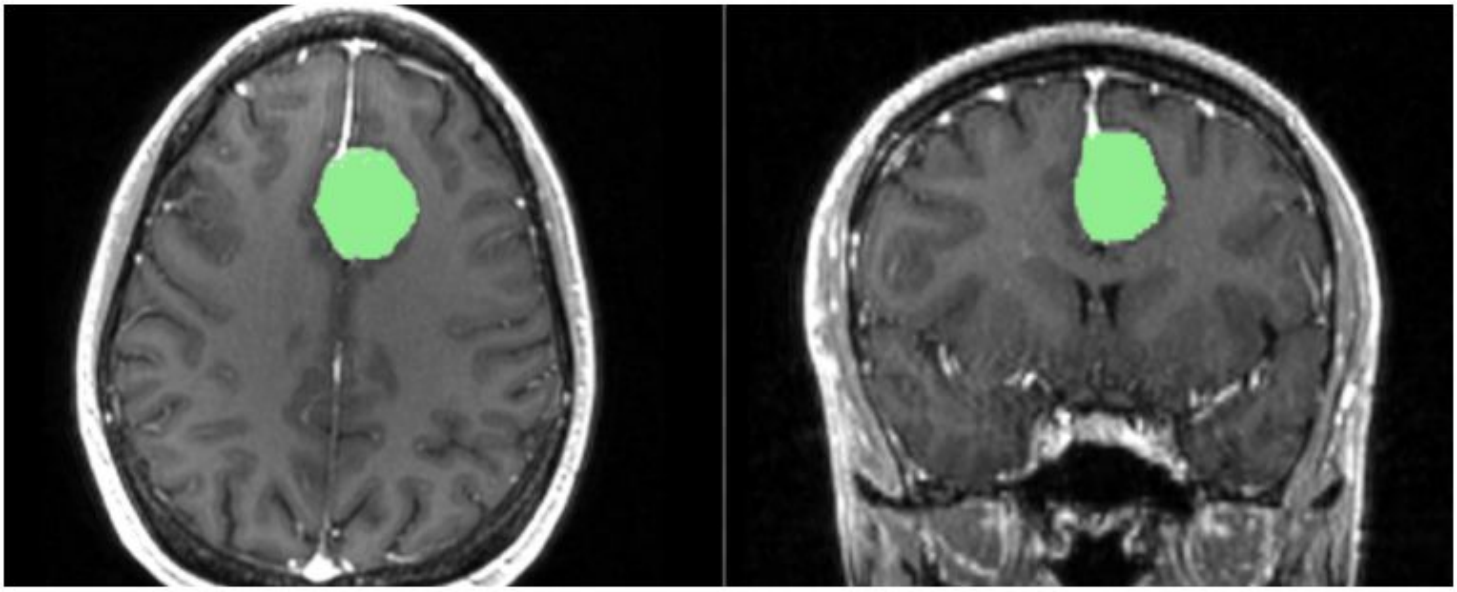
Self-Driving Car

Computer vision is used for vehicle and pedestrian recognition and tracking (to determine their speed and predict movement).

Udacity's self-driving car.

Medical Image Analysis and Diagnosis

An AI system, using computer vision, can learn to recognize images of cancerous tissue and help with early detection and diagnoses.



Brain MRI in which a tumor is recognized and colorized using computer vision.

Photo Tagging and Face Recognition

Computer vision can be trained to recognize and tag (or label) faces or different features in any given photo library. This is already a feature that many of our phones have!

Face recognition with labels for perceived emotions.

Partnership with Industry

In the making of this program, we collaborated with industry leaders from NVIDIA to Affectiva to build a course that showcases how computer vision is being applied on the front-lines of technology today. You've already seen an example of how Affectiva uses facial recognition and deep learning to create systems with emotional intelligence, and in the following video, you'll see the many uses for computer vision technology on NVIDIA platforms.

You'll see examples of a self-driving car that uses computer vision to perform a variety of skills:

- object recognition and lane detection
- face detection
- traffic and test-course navigation, and
- object tracking

All techniques that will be covered in more detail in this Computer Vision program.

Working with NVIDIA Tools

Register for the NVIDIA Developer Program to access the latest NVIDIA SDK tools and be the first to hear about NVIDIA product announcements. Learn more at developer.nvidia.com/developer-program .

Cognitive and Emotional Intelligence

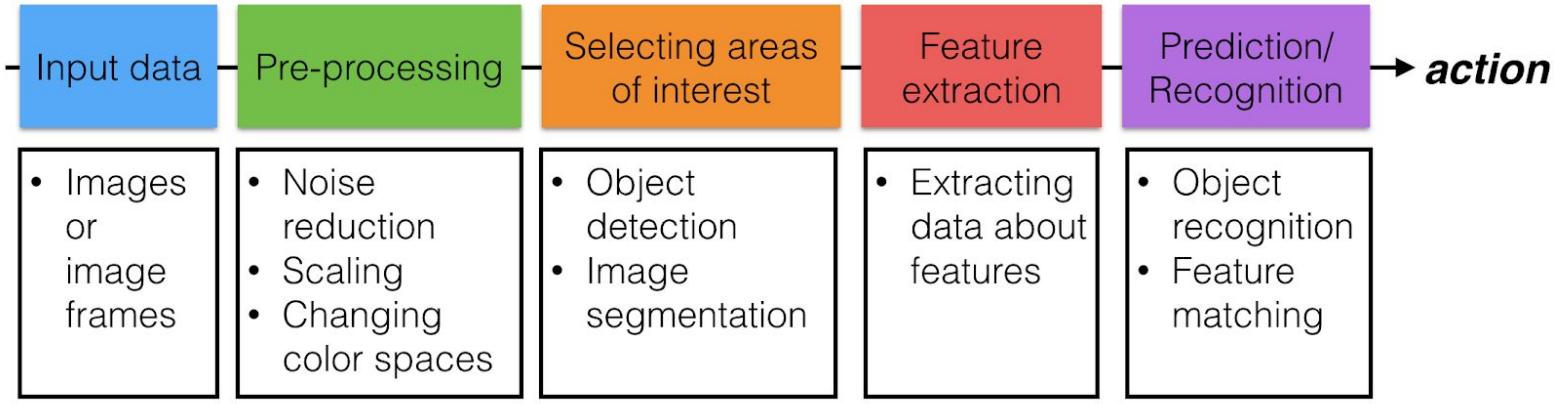
Cognitive intelligence is the ability to reason and understand the world based on observations and facts. It's often what is measured on academic tests and what's measured to calculate a person's IQ.

Emotional intelligence is the ability to understand and influence human emotion. For example, observing that someone looks sad based on their facial expression, body language, and what you know about them - then acting to comfort them or asking them if they want to talk, etc. For humans, this kind of intelligence allows us to form meaningful connections and build a trustworthy network of friends and family. It's also often thought of as *only* a human quality and is not yet a part of traditional AI systems.

If you'd like to learn more about Affectiva and emotion AI, check out their website.

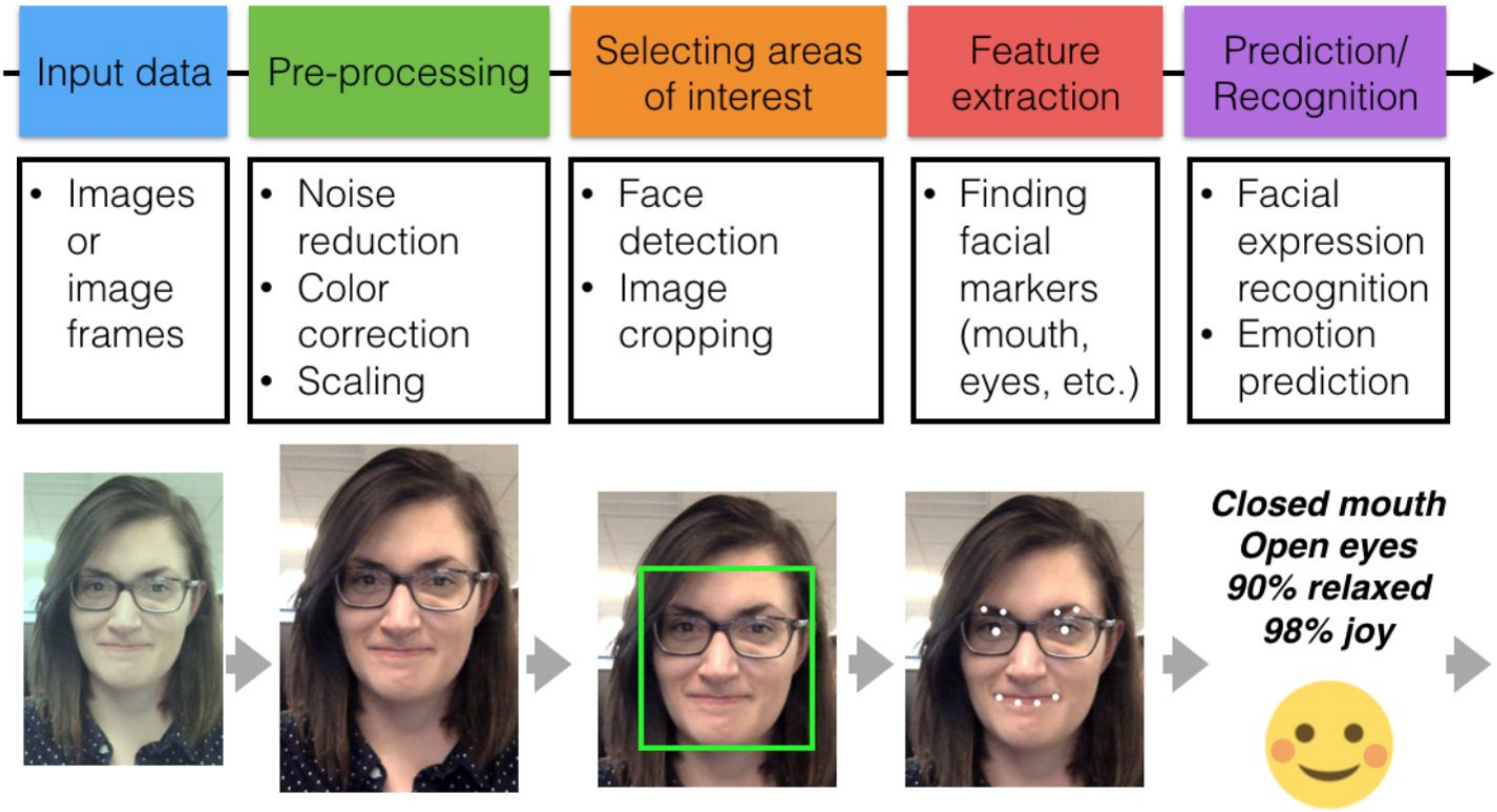
Computer Vision Pipeline

A computer vision pipeline is a series of steps that most computer vision applications will go through. Many vision applications start off by acquiring images and data, then processing that data, performing some analysis and recognition steps, then finally performing an action. The general pipeline is pictured below!



General computer vision processing pipeline

Now, let's take a look at a specific example of a pipeline applied to facial expression recognition.

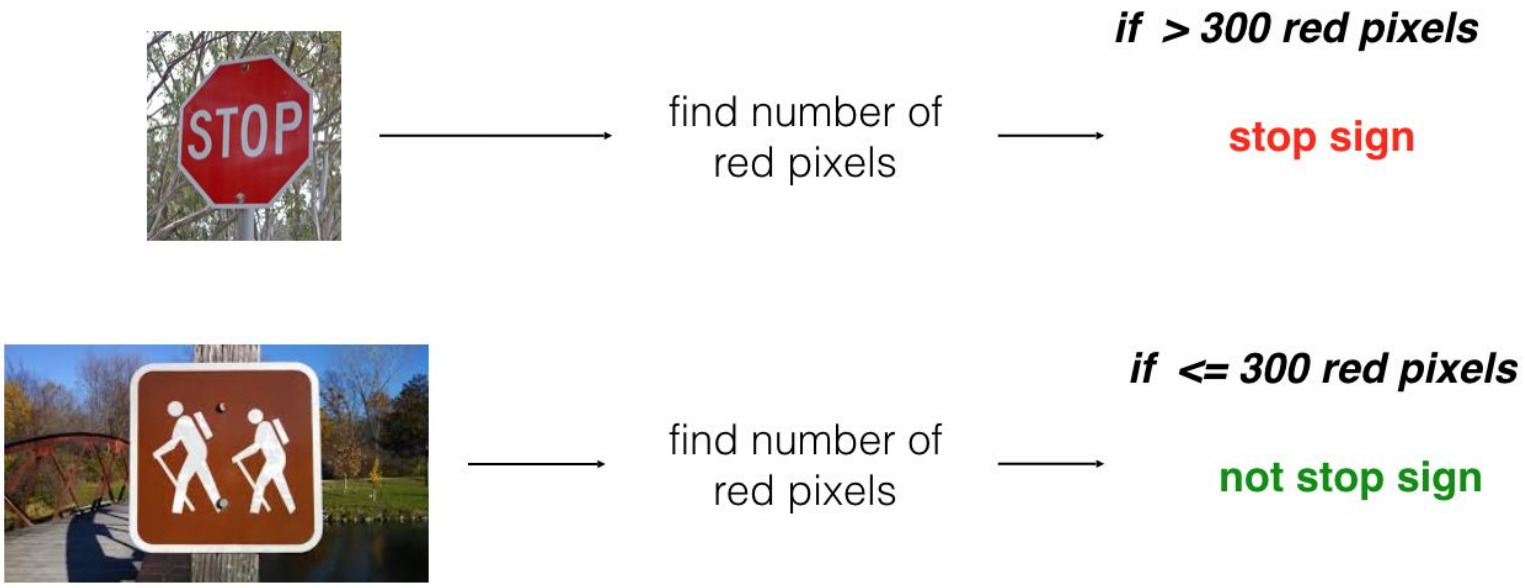


Facial recognition pipeline.

Standardizing Data

Pre-processing images is all about **standardizing** input images so that you can move further along the pipeline and analyze images in the same way. In machine learning tasks, the pre-processing step is often one of the most important.

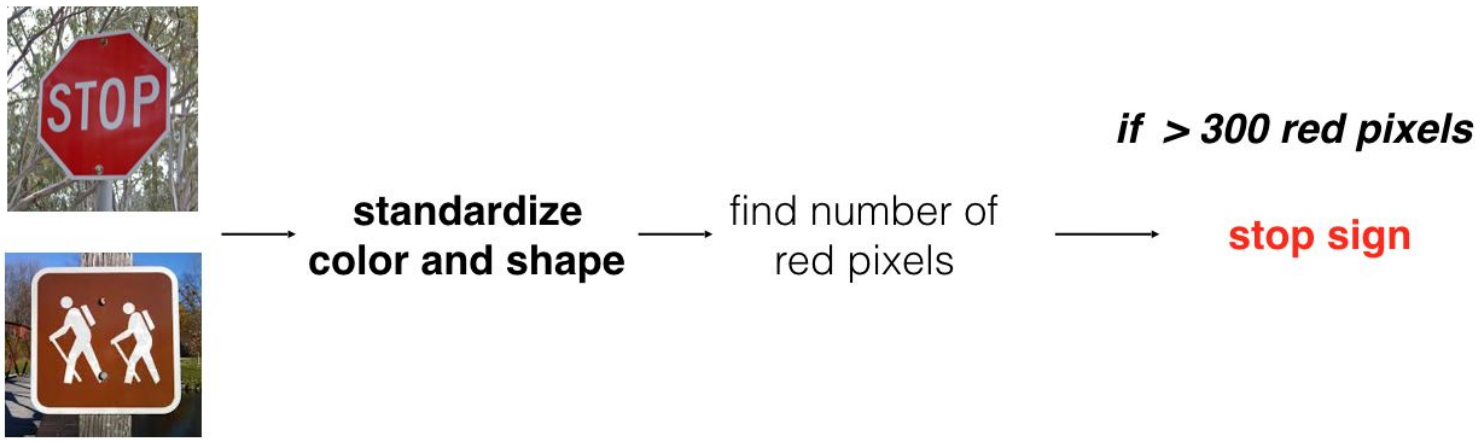
For example, imagine that you've created a simple algorithm to distinguish between stop signs and other traffic lights.



Images of traffic signs; a stop sign is on top and a hiking sign is on the bottom.

If the images are different sizes, or even cropped differently, then this counting tactic will likely fail! So, it's important to pre-process these images so that they are standardized before they move along the pipeline. In the example below, you can see that the images are pre-processed into a standard square size.

The algorithm counts up the number of red pixels in a given image and if there are enough of them, it classifies an image as a stop sign. In this example, we are just extracting a color feature and skipping over selecting an area of interest (we are looking at the *whole* image). In practice, you'll often see a classification pipeline that looks like this.

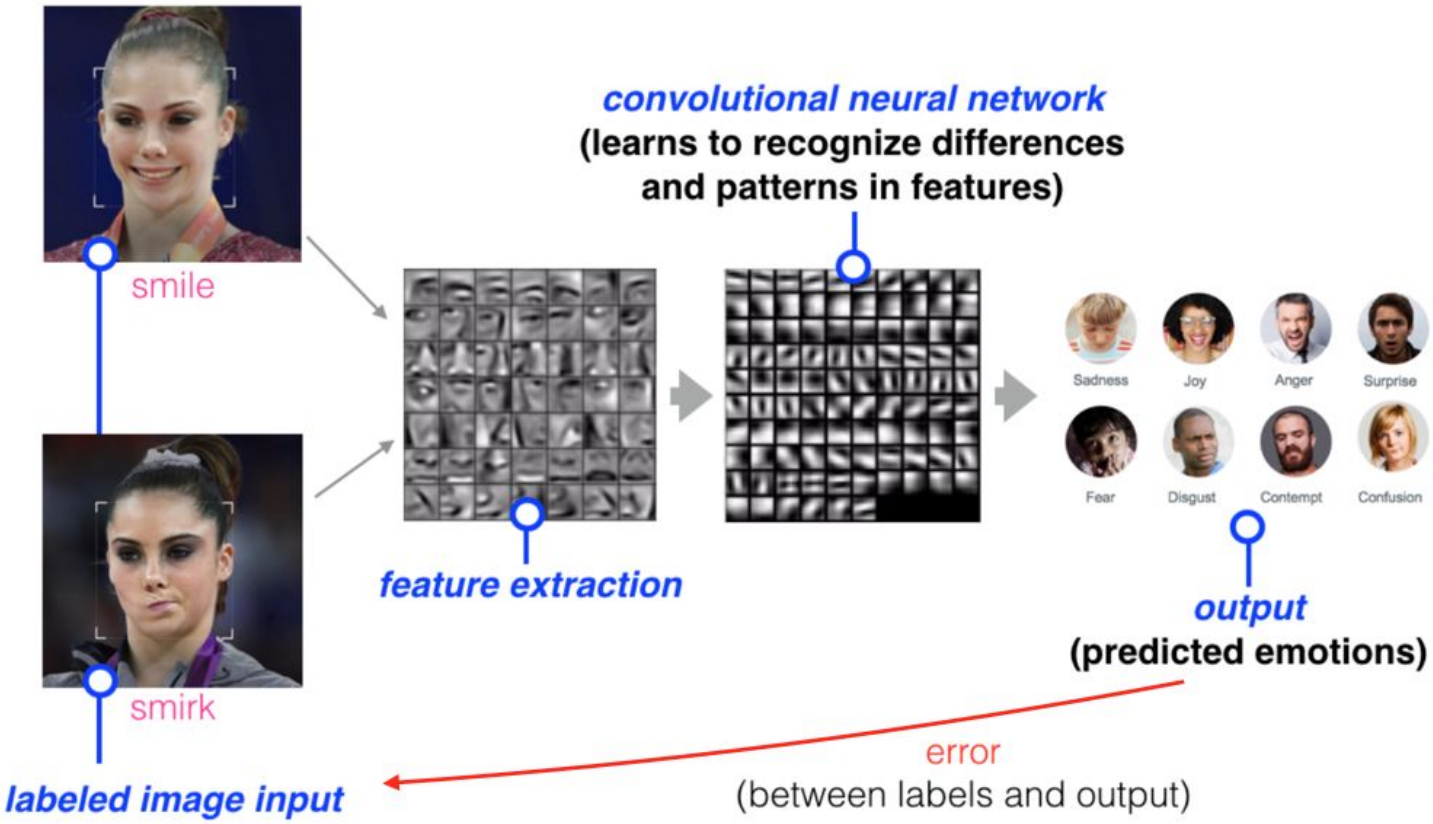


Training a Neural Network

To train a computer vision neural network, we typically provide sets of **labelled images**, which we can compare to the **predicted output** label or recognition measurements. The neural network then monitors any errors it makes (by comparing the correct label to the output label) and corrects for them by modifying how it finds and prioritizes patterns and differences among the image data. Eventually, given enough labelled data, the model should be able to characterize any new, unlabeled, image data it sees!

A training flow is pictured below. This is a convolutional neural network that *learns* to recognize and distinguish between images of a smile and a smirk.

This is a very high-level view of training a neural network, and we'll be diving more into how this works later on in this course. For now, we are explaining this so that you'll be able to jump into coding a computer vision application soon!



Example of a convolutional neural network being trained to distinguish between images of a smile and a smirk.

Gradient descent is a a mathematical way to minimize error in a neural network. More information on this minimization method can be found here.

Convolutional neural networks are a specific type of neural network that are commonly used in computer vision applications. They learn to recognize patterns among a given set of images. If you want to learn more, refer to this resource, and we'll be learning more about these types of networks, and how they work step-by-step, at a different point in this course!

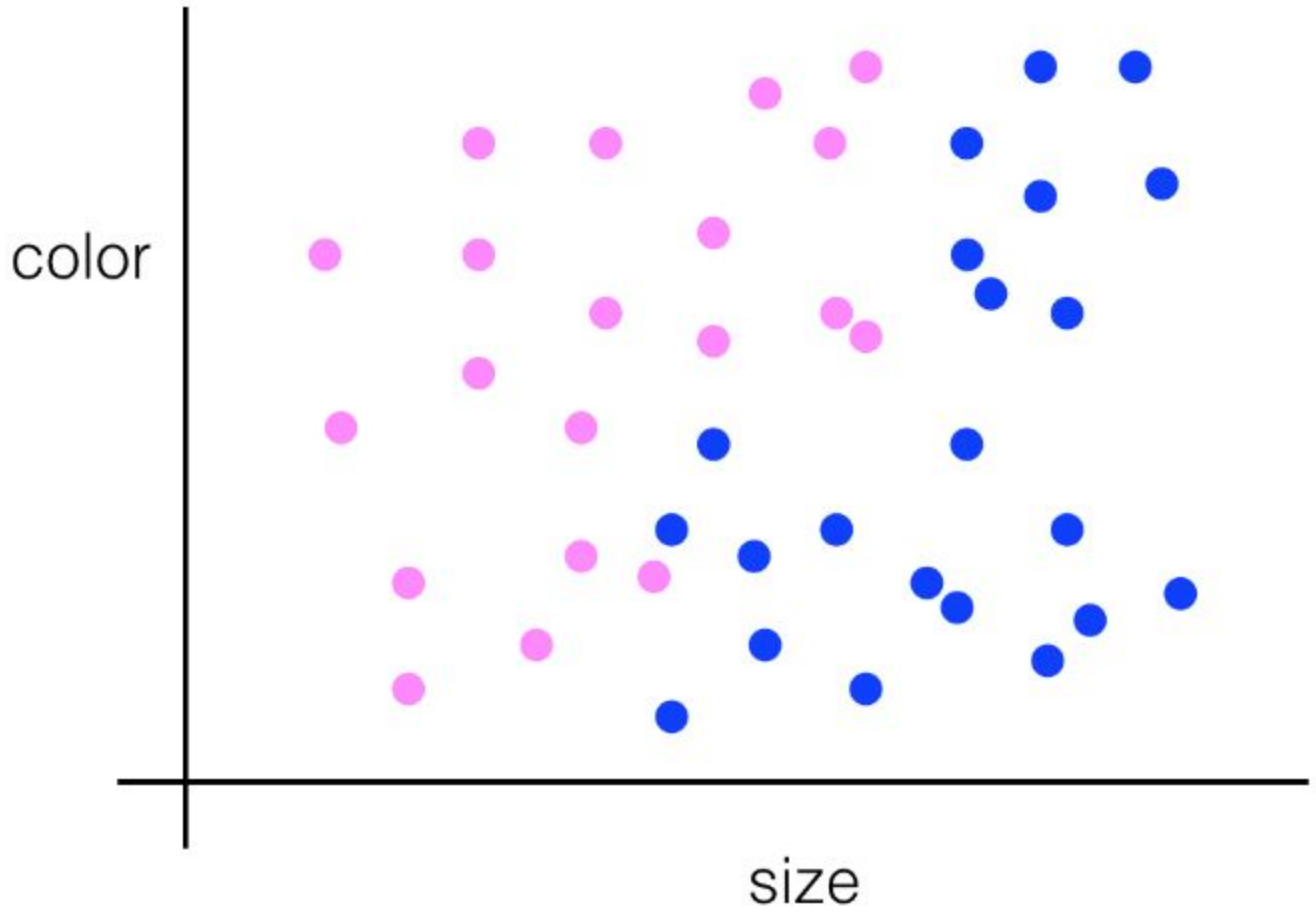
Machine Learning and Neural Networks

When we talk about **machine learning** and **neural networks** used in image classification and pattern recognition, we are really talking about a set of algorithms that can *learn* to recognize patterns in data and sort that data into groups.

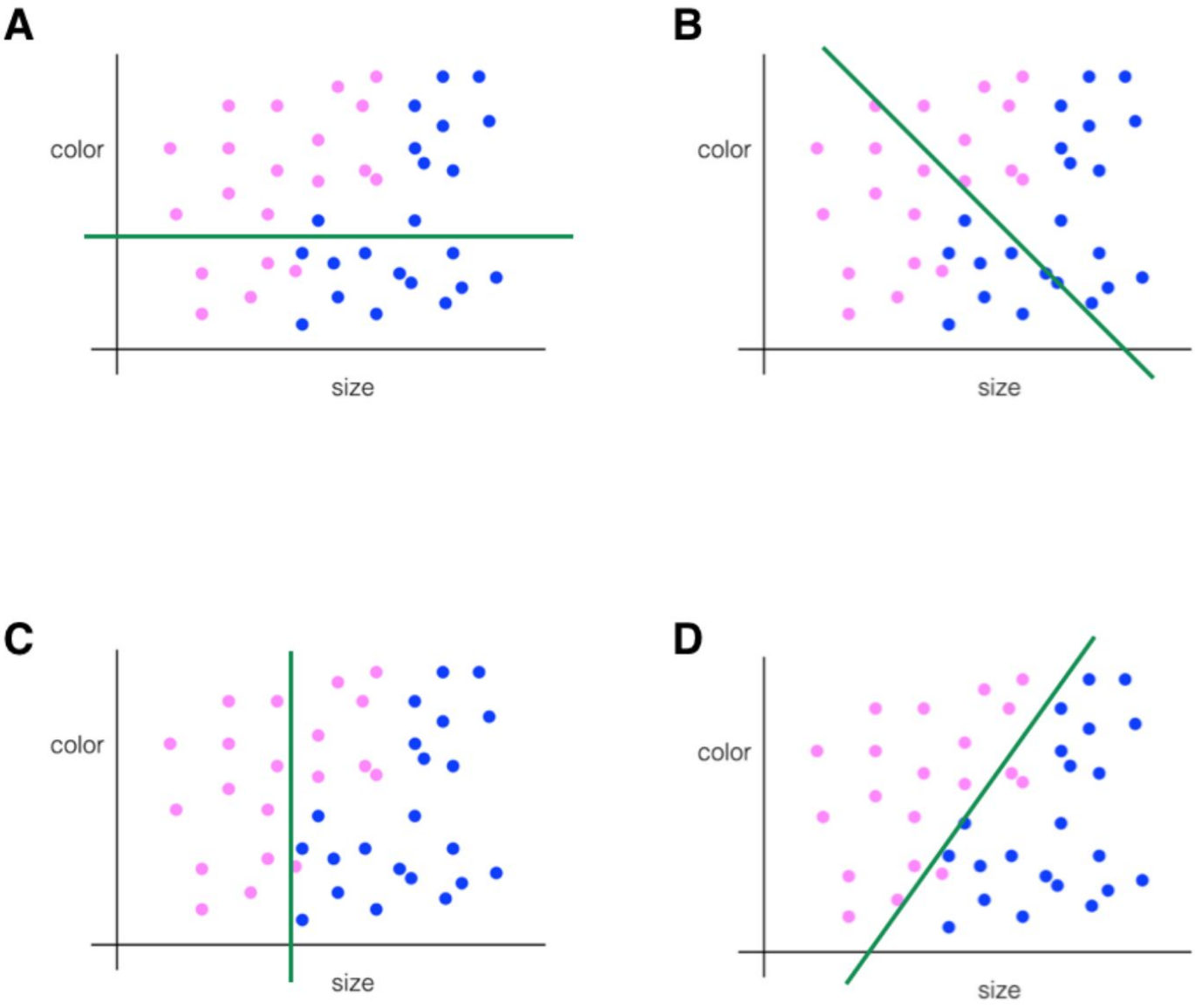
The example we gave earlier was sorting images of facial expressions into two categories: smile or smirk. A neural network might be able to learn to separate these expressions based on their different traits; a neural network can effectively learn how to draw a line that **separates** two kinds of data based on their unique shapes (the different shapes of the eyes and mouth, in the case of a smile and smirk). *Deep* neural networks are similar, only they can draw multiple and more complex separation lines in the sand. Deep neural networks layer separation layers on top of one another to separate complex data into groups.

Separating Data

Say you want to separate two types of image data: images of bikes and of cars. You look at the color of each image and the apparent size of the vehicle in it and plot the data on a graph. Given the following points (pink dots are bikes and blue are cars), how would you choose to separate this data?

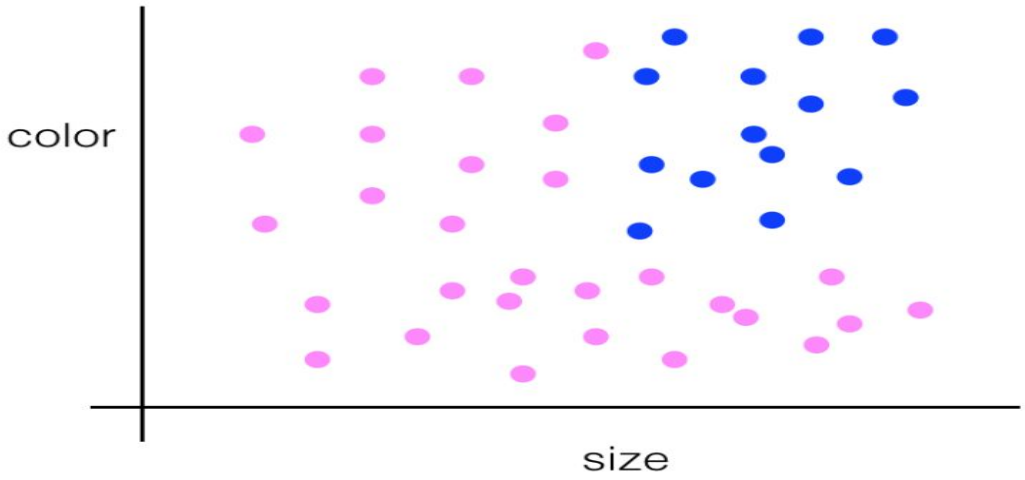


Pink and blue dots representing the size and color of bikes (pink) and cars (blue). The size is on the x-axis and the color on the left axis. Cars tend to be larger than bikes, but both come in a variety of colors.



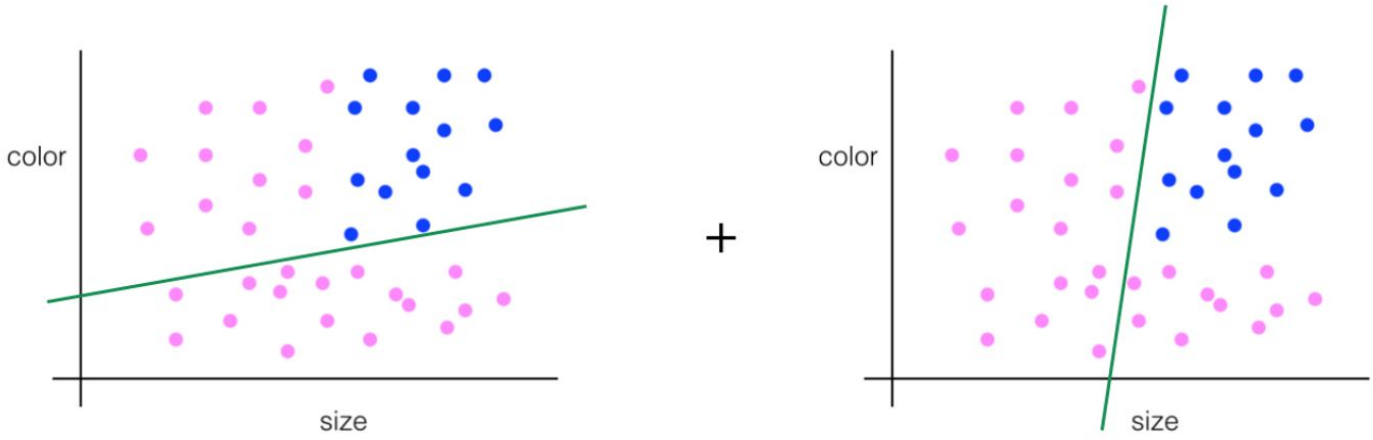
Layers of Separation

What if the data looked like this?

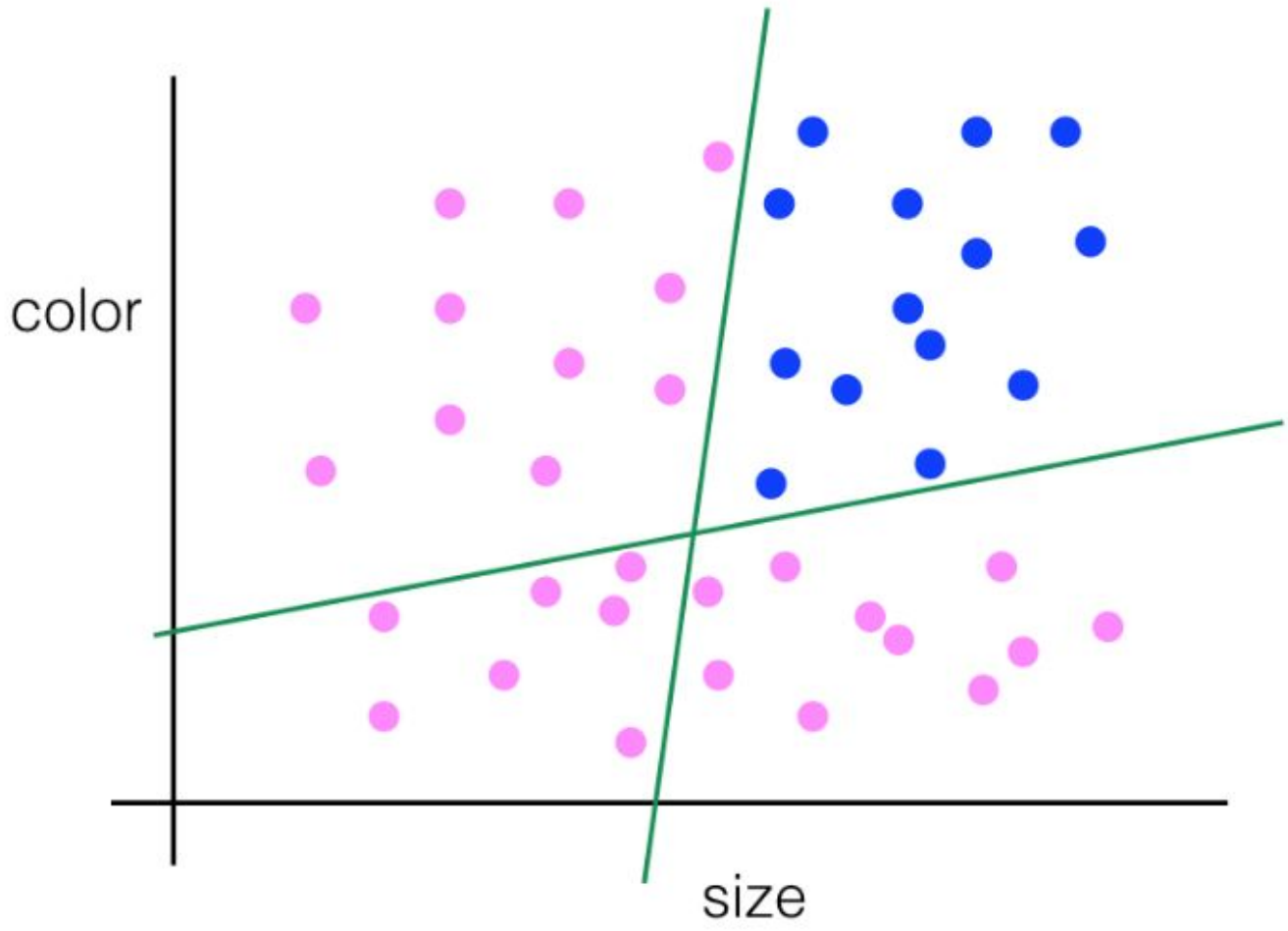


Pink (bike) and blue (car) dots on a similar size-color graph. This time, the blue dots are collected in the top right quadrant of the graph, indicating that cars come in a more limited color palette.

You could combine two different lines of separation! You could even plot a curved line to separate the blue dots from the pink, and this is what machine learning *learns* to do — to choose the best algorithm to separate any given data.



Two, slightly-angled lines, each of which divides the data into two groups.



Images as Numerical Data

Every pixel in an image is just a numerical value and, we can also change these pixel values. We can multiply every single one by a scalar to change how bright the image is, we can shift each pixel value to the right, and many more operations!

Treating images as grids of numbers is the basis for many image processing techniques.

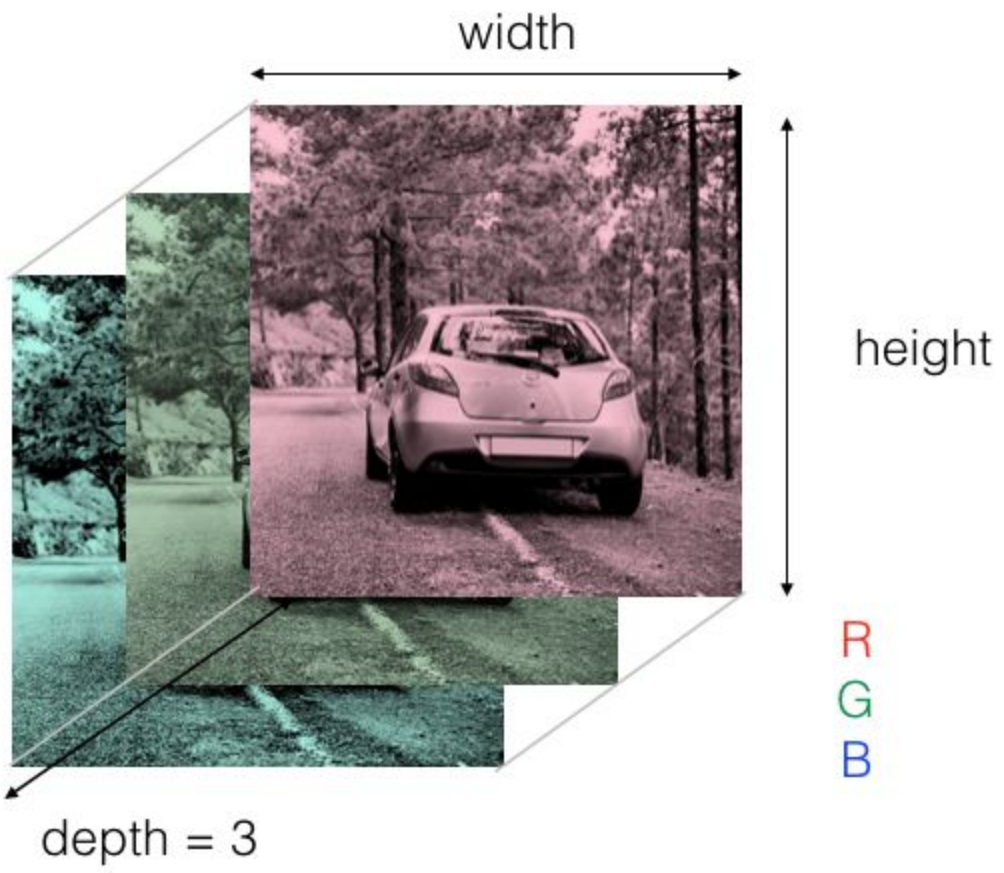
Most color and shape transformations are done just by mathematically operating on an image and changing it pixel-by-pixel.

Color Images

Color images are interpreted as 3D cubes of values with width, height, and depth!

The depth is the number of colors. Most color images can be represented by combinations of only 3 colors: red, green, and blue values; these are known as RGB images. And for RGB images, the depth is 3!

It’s helpful to think of the depth as three stacked, 2D color layers. One layer is Red, one Green, and one Blue. Together they create a complete color image.



RGB layers of a car image.

Importance of Color

In general, when you think of a classification challenge, like identifying lane lines or cars or people, you can decide whether color information and color images are useful by thinking about your own vision.

If the identification problem is easier in color for us humans, it's likely easier for an algorithm to see color images too!

OpenCV

OpenCV is a popular computer vision library that has meany built in tools for image analysis and understanding!

Note: In the example above and in later examples, I'm using my own Jupyter notebook and sets of images stored on my personal computer. You're encouraged to set up a similar environment and use images of your own to practice! You'll also be given some code quizzes (coming up next), with images provided, to practice these techniques.

Why BGR instead of RGB?

OpenCV reads in images in BGR format (instead of RGB) because when OpenCV was first being developed, BGR color format was popular among camera manufacturers and image software providers. The red channel was considered one of the least important color channels, so was listed last, and many bitmaps use BGR format for image storage. However, now the standard has changed and most image software and cameras use RGB format, which is why, in these examples, it's good practice to initially convert BGR images to RGB before analyzing or manipulating them.

Changing Color Spaces

To change color spaces, we used OpenCV's `cvtColor` function, whose documentation is here.

Color Selection

To select the most accurate color boundaries, it's often useful to use a color picker and choose the color boundaries that define the region you want to select!

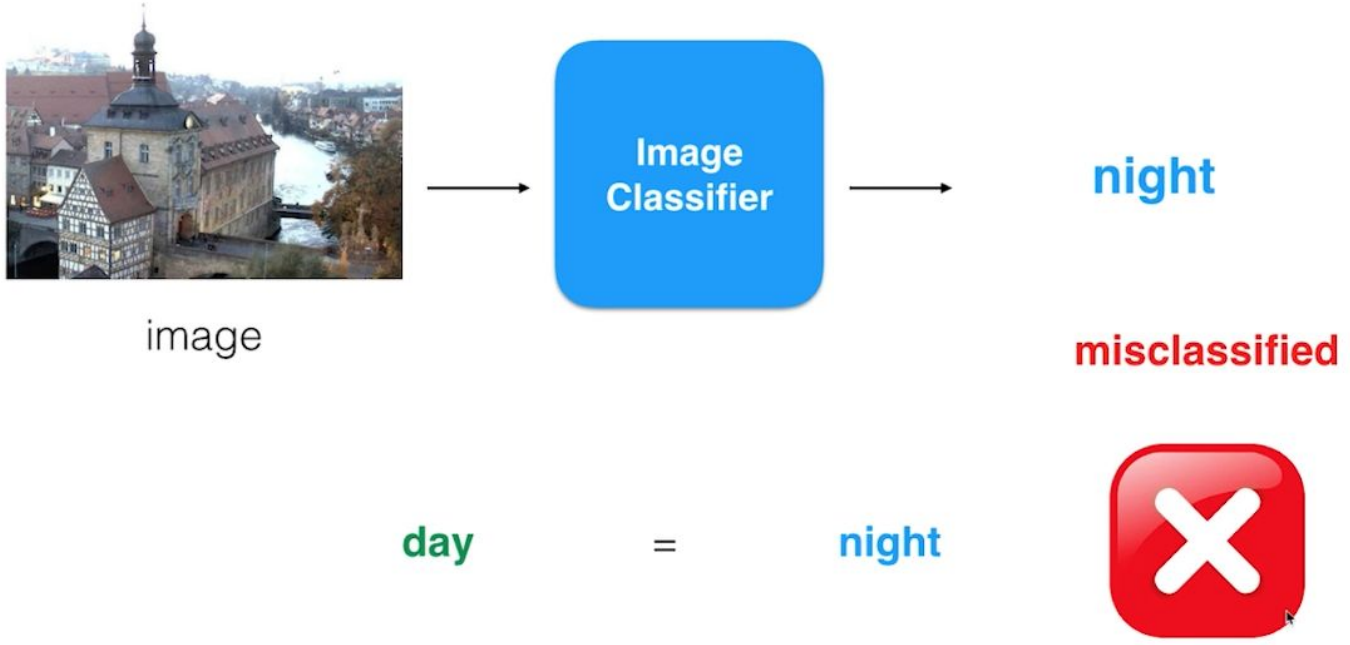
Why do we need labels?

You can tell if an image is night or day, but a computer cannot unless we tell it explicitly with a label!

This becomes especially important when we are testing the accuracy of a classification model.

A classifier takes in an image as input and should output a `predicted_label` that tells us the predicted class of that image. Now, when we load in data, like you've seen, we load in what are called the `true_labels` which are the *correct* labels for the image.

To check the accuracy of a classification model, we compare the predicted and true labels. If the true and predicted labels match, then we've classified the image correctly! Sometimes the labels do not match, which means we've misclassified an image.



A misclassified image example. The true_label is "day" and the predicted_label is "night".

Accuracy

After looking at many images, the accuracy of a classifier is defined as the number of correctly classified images (for which the predicted_label matches the true label) divided by the total number of images. So, say we tried to classify 100 images total, and we correctly classified 81 of them. We'd have 0.81 or 81% accuracy!

We can tell a computer to check the accuracy of a classifier only when we have these predicted and true labels to compare. We can also learn from any mistakes the classifier makes, as we'll see later in this lesson.

Numerical labels

It’s good practice to use numerical labels instead of strings or categorical labels. They're easier to track and compare. So, for our day and night, binary class example, instead of "day" and "night" labels we’ll use the numerical labels: 0 for night and 1 for day.

Okay, now you’re familiar with the day and night image data AND you know what a label is and why we use them; you’re ready for the next steps. We’ll be building a classification pipeline from start to end!

Let’s first brainstorm what steps we’ll take to classify these images.

Distinguishing and Measurable Traits

When you approach a classification challenge, you may ask yourself: how can I tell these images apart? What traits do these images have that differentiate them, and how can I write code to represent their differences? Adding on to that, how can I ignore irrelevant or overly similar parts of these images?

You may have thought about a number of distinguishing features: day images are much brighter, generally, than night images. Night images also have these really bright small spots, so the brightness over the whole image varies a lot more than the day images. There is a lot more of a gray/blue color palette in the day images.

There are lots of measurable traits that distinguish these images, and these measurable traits are referred to as **features**.

A feature a measurable component of an image or object that is, ideally, unique and recognizable under varying conditions - like under varying light or camera angle. And we’ll learn more about features soon.

Standardizing and Pre-processing

But we’re getting ahead of ourselves! To extract features from any image, we have to pre-process and standardize them!

Next we’ll take a look at the standardization steps we should take before we can consistently extract features.

Numerical vs. Categorical

Let's learn a little more about labels. After visualizing the image data, you'll have seen that each image has an attached label: "day" or "night," and these are known as **categorical values**.

Categorical values are typically text values that represent various traits about an image. A couple examples are:

- An "animal" variable with the values: "cat," "tiger," "hippopotamus," and "dog."
- A "color" variable with the values: "red," "green," and "blue."

Each value represents a different category, and most collected data is labeled in this way!

These labels are descriptive for us, but may be inefficient for a classification task. Many machine learning algorithms do not use categorical data; they require that all output be numerical. Numbers are easily compared and stored in memory, and for this reason, we often have to convert categorical values into **numerical labels**. There are two main approaches that you'll come across:

1. Integer encoding
2. One hot-encoding

Integer Encoding

Integer encoding means to assign each category value an integer value. So, day = 1 and night = 0. This is a nice way to separate binary data, and it's what we'll do for our day and night images.

One-hot Encoding

One-hot encoding is often used when there are more than 2 values to separate. A one-hot label is a 1D list that's the length of the number of classes. Say we are looking at the animal variable with the values: "cat,"

"tiger," "hippopotamus," and "dog." There are 4 classes in this category and so our one-hot labels will be a list of length four. The list will be all 0's and one 1; the 1 indicates which class a certain image is.

For example, since we have four classes (cat, tiger, hippopotamus, and dog), we can make a list in that order: [cat value, tiger value, hippopotamus value, dog value]. In general, order does not matter.

If we have an image and it's one-hot label is `[0, 1, 0, 0]`, what does that indicate?

Accuracy

The accuracy of a classification model is found by comparing predicted and true labels. For any given image, if the `predicted_label` matches the`true_label`, then this is a correctly classified image, if not, it is misclassified.

The accuracy is given by the number of correctly classified images divided by the total number of images. We'll test this classification model on new images, this is called a test set of data.

Test Data

Test data is previously unseen image data. The data you *have* seen, and that you used to help build a classifier is called training data, which we've been referring to. The idea in creating these two sets is to have one set that you can analyze and learn from (training), and one that you can get a sense of how your classifier might work in a real-world, general scenario. You could imagine going through each image in the training set and creating a classifier that can classify all of these training images correctly, but, you actually want to build a classifier that **recognizes general patterns in data**, so that when it is faced with a real-world scenario, it will still work!

So, we use a new, test set of data to see how a classification model might work in the real-world and to determine the accuracy of the model.

Misclassified Images

In this and most classification examples, there are a few misclassified images in the test set. To see how to improve, it’s useful to take a look at these misclassified images; look at what they were mistakenly labeled as and where your model fails. It will be up to you to look at these images and think about how to improve the classification model!

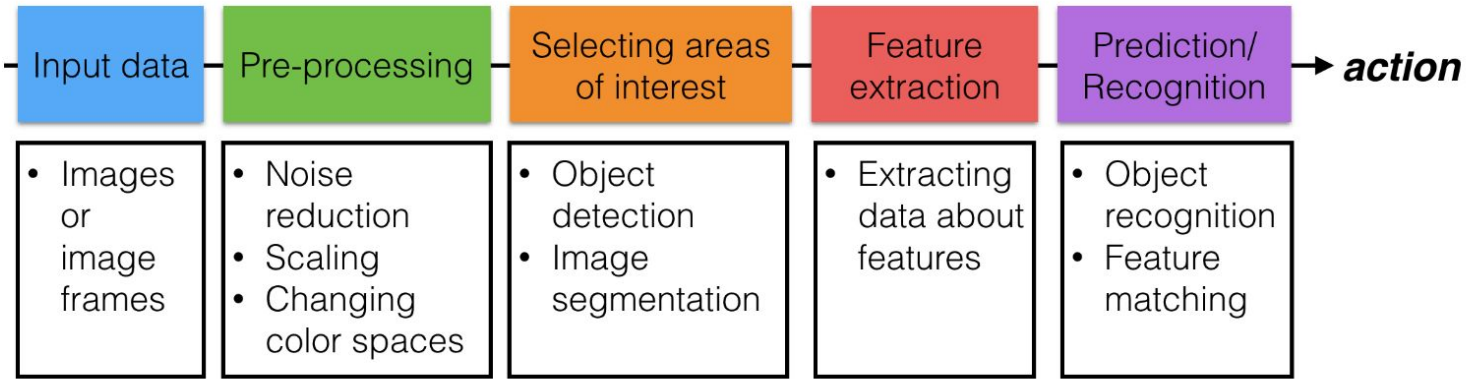
Review and the Computer Vision Pipeline

In this lesson, you’ve really made it through a lot of material, from learning how images are represented to programming an image classifier!

You approached the classification challenge by completing each step of the **Computer Vision Pipeline** ste-by-step. First by looking at the classification problem, visualizing the image data you were working with, and planning out a complete approach to a solution.

The steps include **pre-processing** images so that they could be further analyzed in the same way, this included changing color spaces. Then we moved on to **feature extraction**, in which you decided on distinguishing traits in each class of image, and tried to isolate those features! You may note that skipped the pipeline step of "Selecting Areas of Interest," and this is because we focused on classifying an image as a whole and did not need break it up into different segments, but we'll see where this step can be useful later in this course.

Finally, you created a complete **classifier** that output a label or a class for a given image, and analyzed your classification model to see its accuracy!



Computer Vision Pipeline.

Filters

Now, we've seen how to use color to help isolate a desired portion of an image and even help classify an image!

In addition to taking advantage of color information, we also have knowledge about patterns of grayscale intensity in an image. Intensity is a measure of light and dark similar to brightness, and we can use this knowledge to detect other areas or objects of interest. For example, you can often identify the edges of an object by looking at an abrupt change in intensity, which happens when an image changes from a very dark to light area, or vice versa.

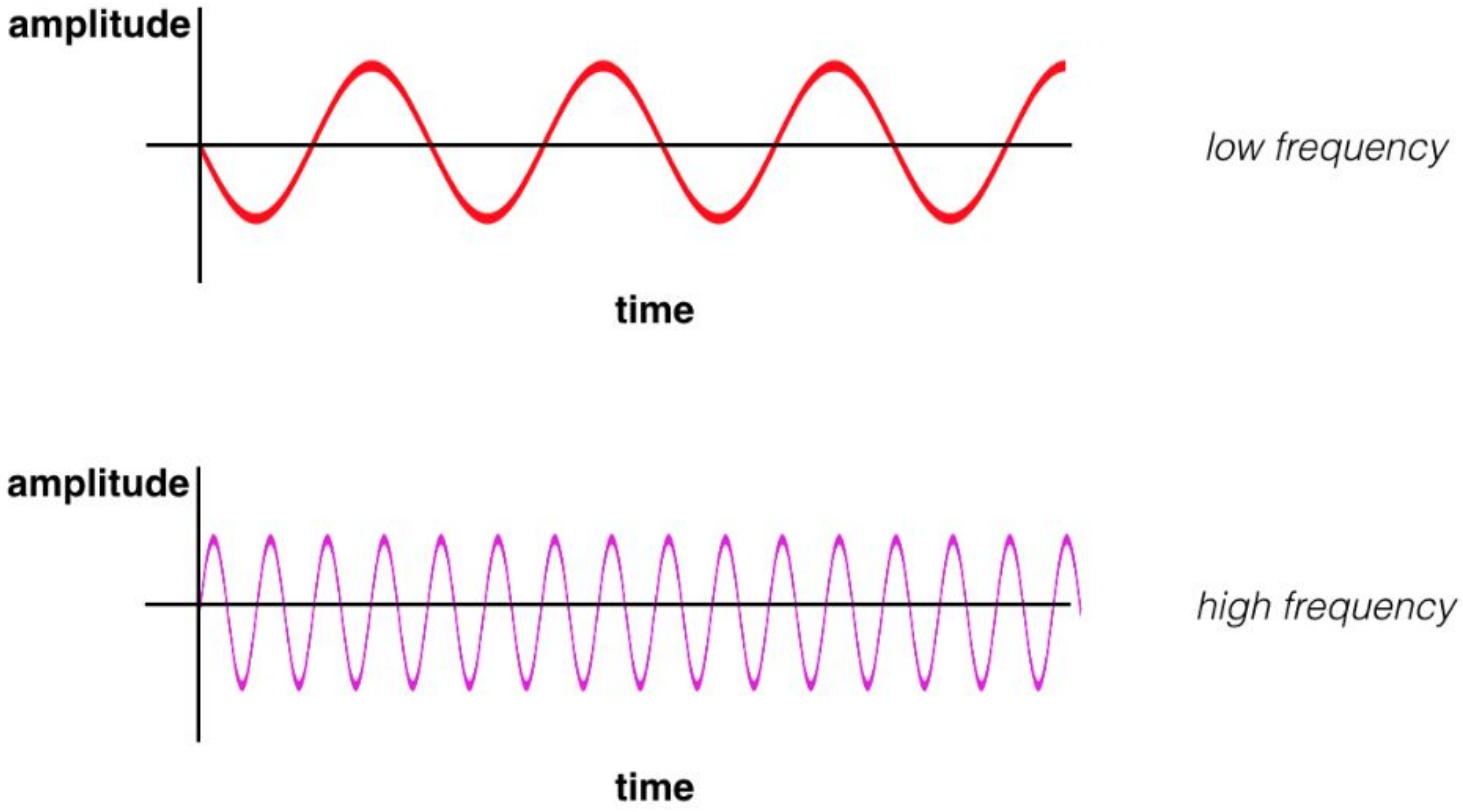
To detect these changes, you'll be using and creating specific image filters that look at groups of pixels and detect big changes in intensity in an image. These filters produce an output that shows these edges.

So, let's take a closer look at these filters and see when they're useful in processing images and identifying traits of interest.

Frequency in images

We have an intuition of what frequency means when it comes to sound. High-frequency is a high pitched noise, like a bird chirp or violin. And low frequency sounds are low pitch, like a deep voice or a bass drum. For sound,

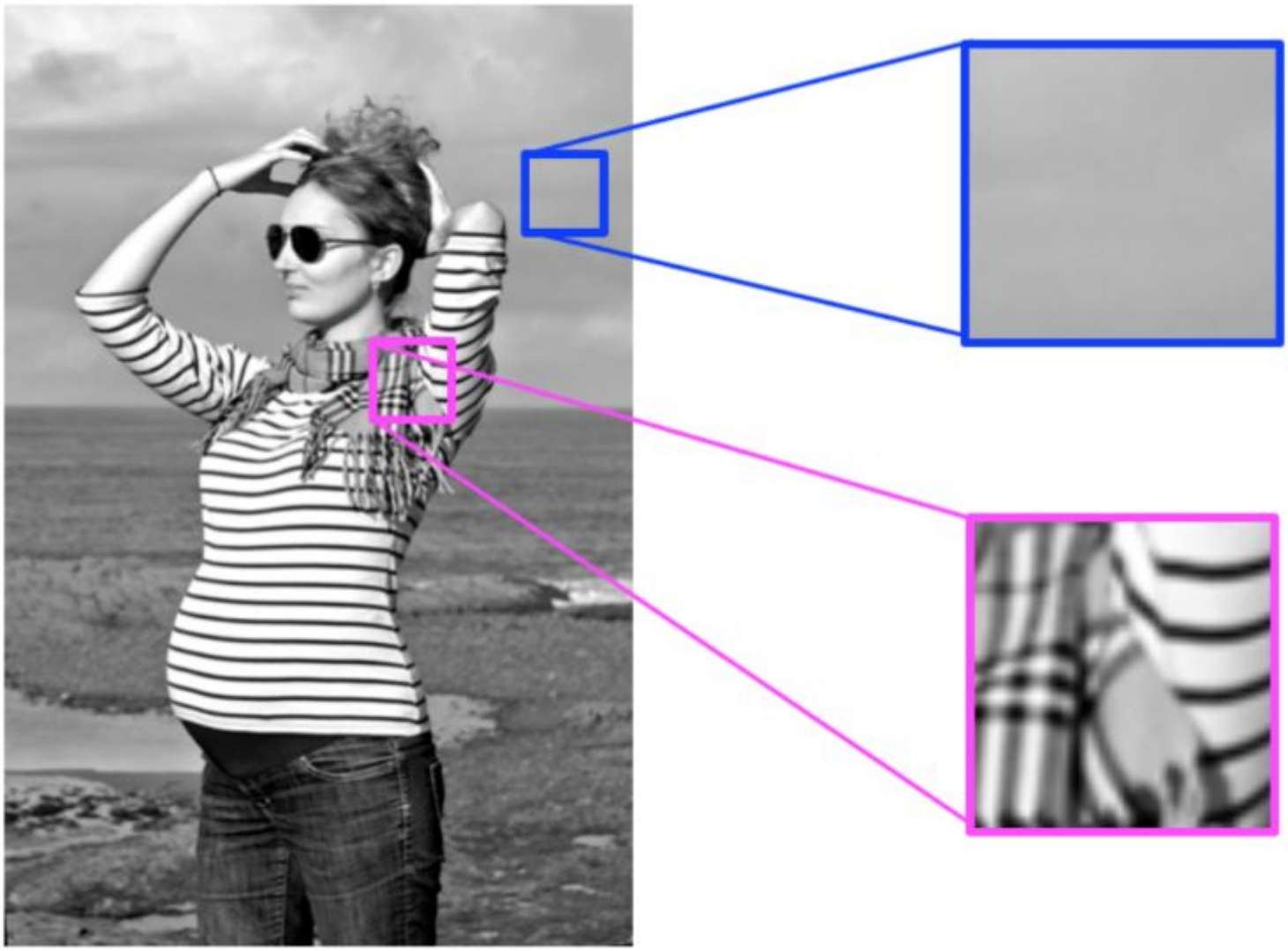
frequency actually refers to how fast a sound wave is oscillating; oscillations are usually measured in cycles/s (Hz), and high pitches and made by high-frequency waves. Examples of low and high-frequency sound waves are pictured below. On the y-axis is amplitude, which is a measure of sound pressure that corresponds to the perceived loudness of a sound and on the x-axis is time.



(Top image) a low frequency sound wave (bottom) a high frequency sound wave.

High and low frequency

Similarly, frequency in images is a **rate of change**. But, what does it means for an image to change? Well, images change in space, and a high frequency image is one where the intensity changes a lot. And the level of brightness changes quickly from one pixel to the next. A low frequency image may be one that is relatively uniform in brightness or changes very slowly. This is easiest to see in an example.



High and low frequency image patterns.

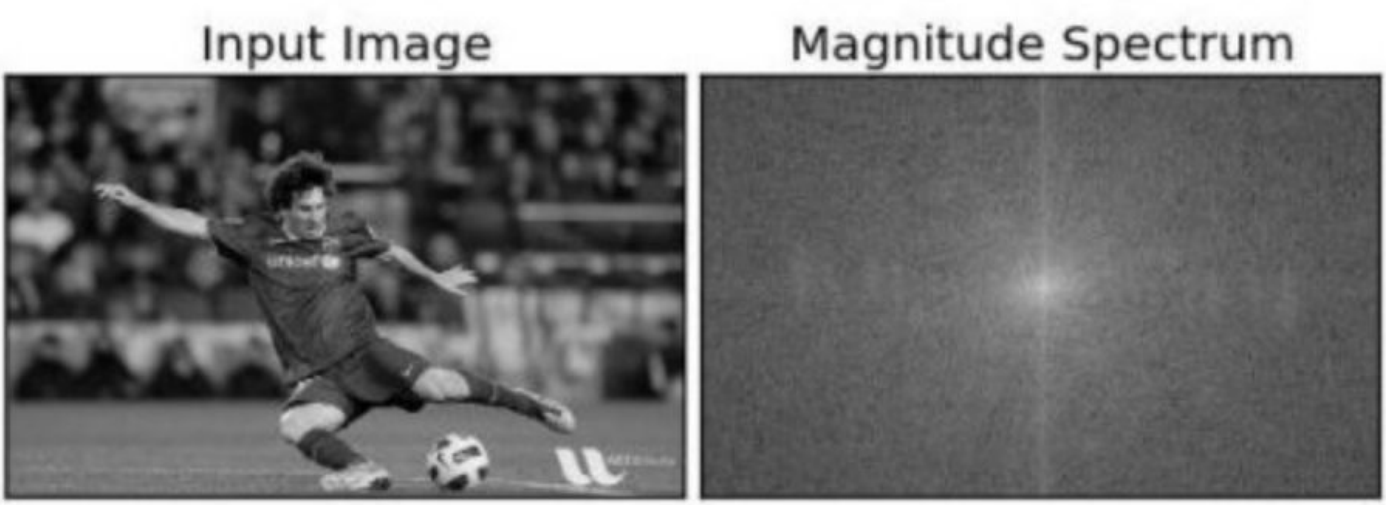
Most images have both high-frequency and low-frequency components. In the image above, on the scarf and striped shirt, we have a high-frequency image pattern; this part changes very rapidly from one brightness to another. Higher up in this same image, we see parts of the sky and background that change very gradually, which is considered a smooth, low-frequency pattern.

High-frequency components also correspond to the edges of objects in images, which can help us classify those objects.

Fourier Transform

The Fourier Transform (FT) is an important image processing tool which is used to decompose an image into its frequency components. The output of an FT represents the image in the frequency domain, while the input image is the spatial domain (x, y) equivalent. In the frequency domain image, each point represents a particular frequency contained in the spatial domain image. So, for images with a lot of high-frequency components (edges, corners, and stripes), there will be a number of points in the frequency domain at high frequency values.

Take a look at how FT's are done with OpenCV, here.



An image of a soccer player and the corresponding frequency domain image (right). The concentrated points in the center of the frequency domain image mean that this image has a lot of low frequency (smooth background) components.

This decomposition is particularly interesting in the context of bandpass filters, which can isolate a certain range of frequencies and mask an image according to a low and high frequency threshold.

Gradients

Gradients are a measure of intensity change in an image, and they generally mark object boundaries and changing area of light and dark. If we think back to treating images as functions, *F(x, y)*, we can think of the gradient as a derivative operation *F'(x, y)*. Where the derivative is a measurement of intensity change.

Sobel filters

The Sobel filter is very commonly used in edge detection and in finding patterns in intensity in an image. Applying a Sobel filter to an image is a way of **taking (an approximation) of the derivative of the image** in the x or y direction. The operators for $Sobel_x$ and $Sobel_y$, respectively, look like this:

$$S_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

$$S_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

Sobel filters

Next, let's see an example of these two filters applied to an image of the brain.

Sobel x

Sobel y

Sobel x and y filters (left and right) applied to an image of a brain

x vs. y

In the above images, you can see that the gradients taken in both the x and the y directions detect the edges of the brain and pick up other edges. Taking the gradient in the x direction emphasizes edges closer to vertical. Alternatively, taking the gradient in the y direction emphasizes edges closer to horizontal.

Magnitude

Sobel also detects which edges are *strongest*. This is encapsulated by the **magnitude** of the gradient; the greater the magnitude, the stronger the edge is. The magnitude, or absolute value, of the gradient is just the square root of the squares of the individual x and y gradients. For a gradient in both the x **and** y directions, the magnitude is the square root of the sum of the squares.

abs_sobelx= $\sqrt{(sobel_x)^2}$

abs_sobely= $\sqrt{(sobel_y)^2}$

abs_sobelxy= $\sqrt{(sobel_x)^2 + (sobel_y)^2}$

Direction

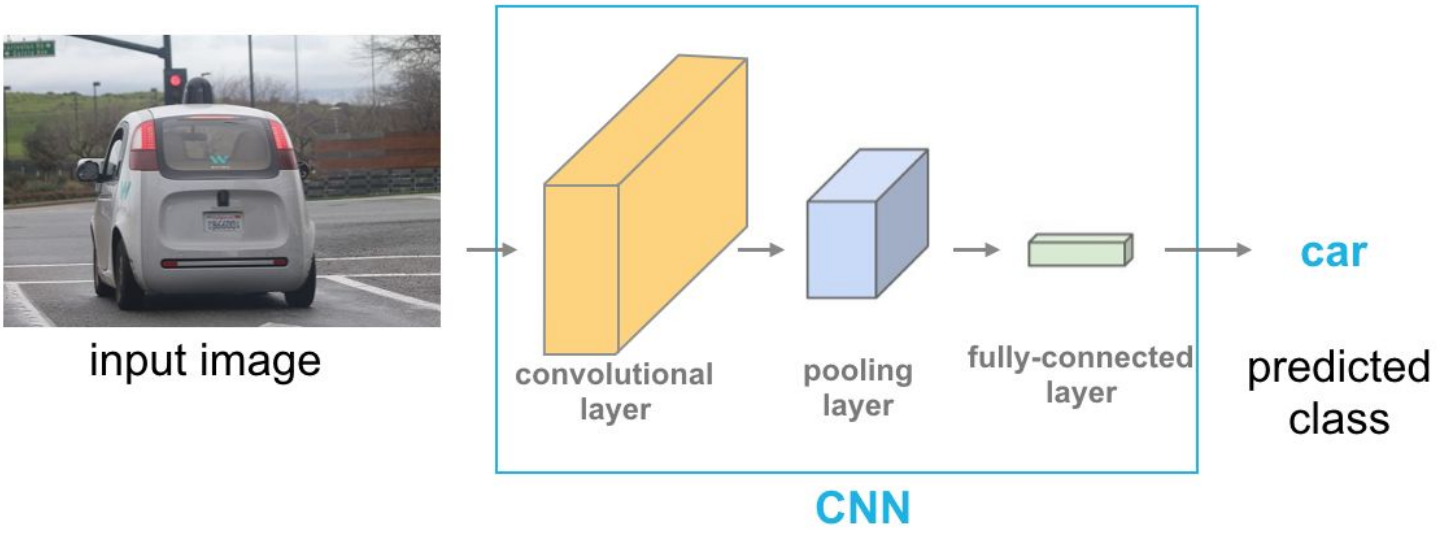
In many cases, it will be useful to look for edges in a particular orientation. For example, we may want to find lines that only angle upwards or point left. By calculating the direction of the image gradient in the x and y directions separately, we can determine the direction of that gradient!

The direction of the gradient is simply the inverse tangent (arctangent) of the y gradient divided by the x gradient:

$tan^{-1}(sobel_y / sobel_x)$.

The Importance of Filters

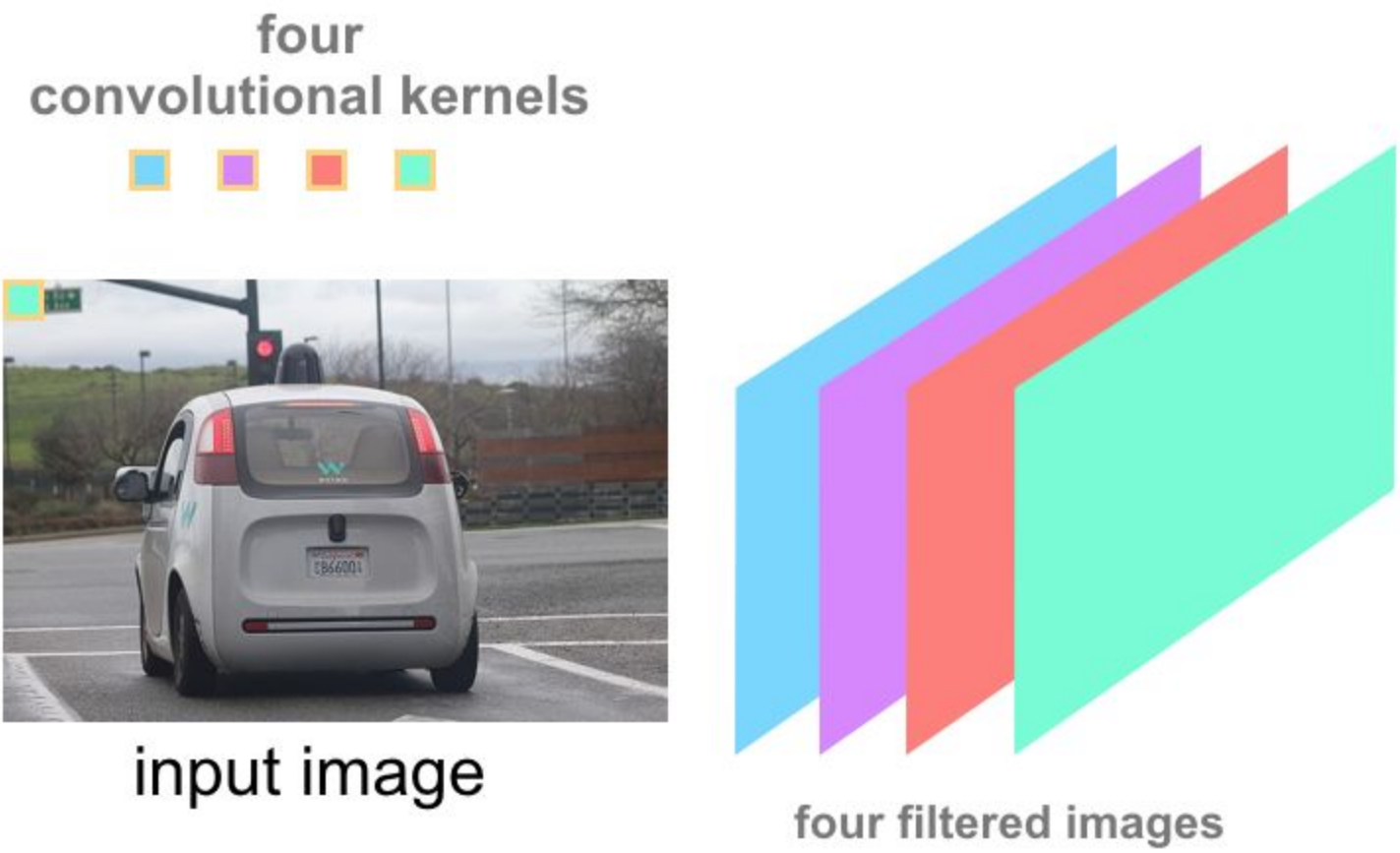
What you've just learned about different types of filters will be really important as you progress through this course, especially when you get to Convolutional Neural Networks (CNNs). CNNs are a kind of deep learning model that can learn to do things like image classification and object recognition. They keep track of spatial information and *learn* to extract features like the edges of objects in something called a **convolutional layer**. Below you'll see an simple CNN structure, made of multiple layers, below, including this "convolutional layer".



Layers in a CNN.

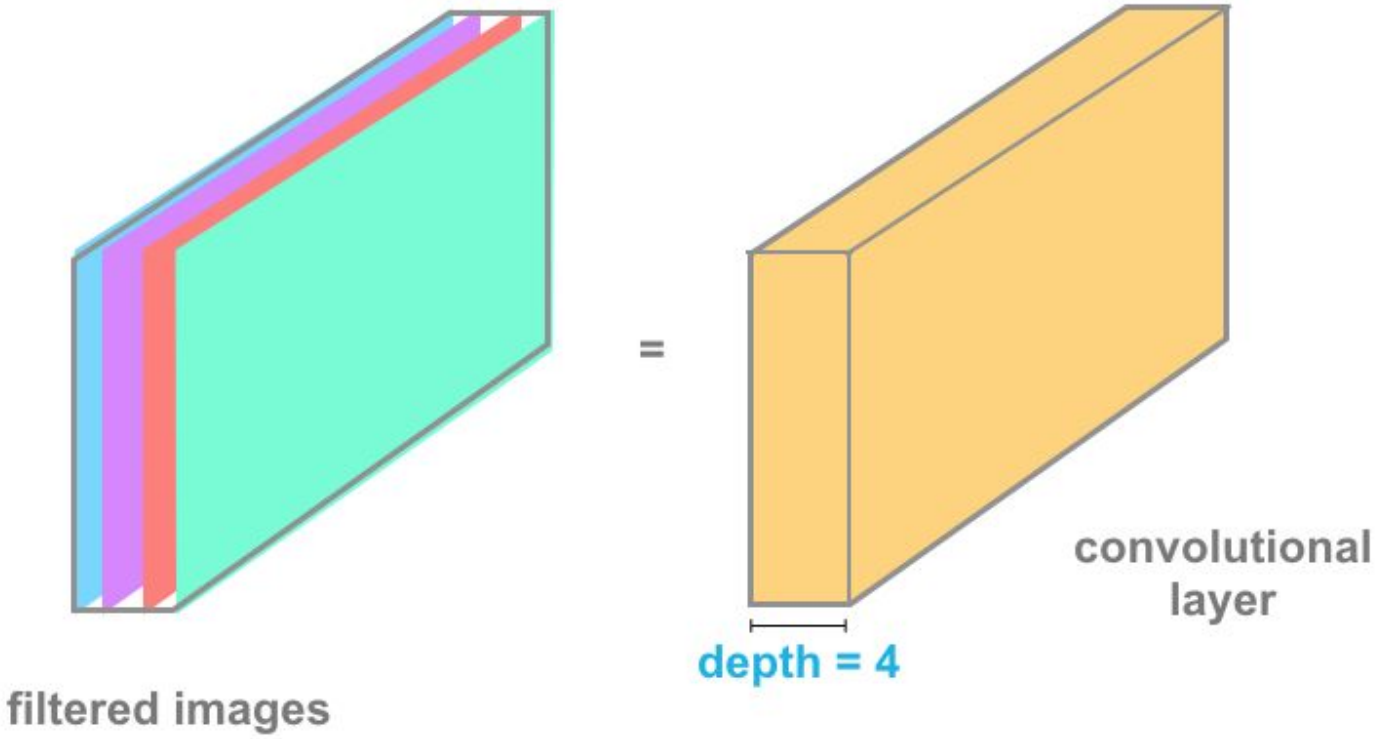
Convolutional Layer

The convolutional layer is produced by applying a series of many different image filters, also known as convolutional kernels, to an input image.



4 kernels = 4 filtered images.

In the example shown, 4 different filters produce 4 differently filtered output images. When we stack these images, we form a complete convolutional layer with a depth of 4!



A convolutional layer.

Learning

In the code you've been working with, you've been setting the values of filter weights explicitly, but neural networks will actually *learn* the best filter weights as they train on a set of image data. You'll learn all about this type of neural network later in this section, but know that high-pass and low-pass filters are what define the behavior of a network like this, and you know how to code those from scratch!

In practice, you'll also find that many neural networks learn to detect the edges of images because the edges of objects contain valuable information about the shape of an object.

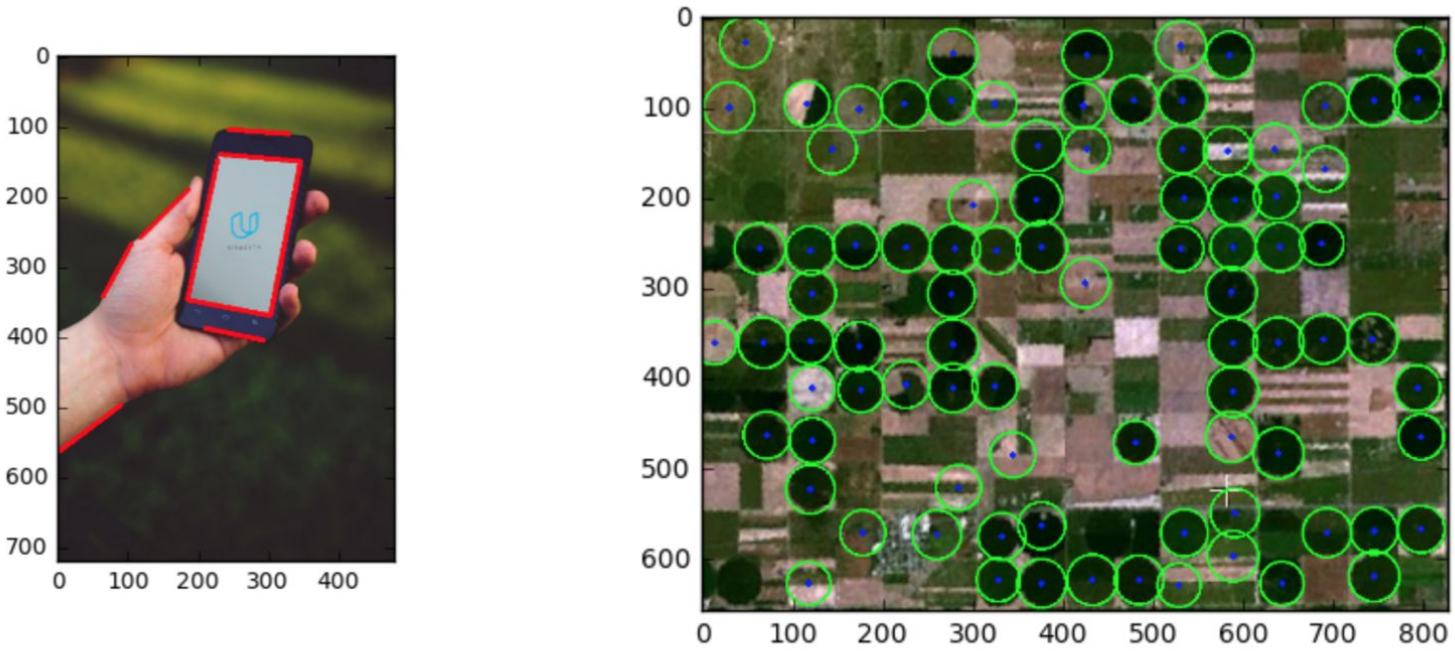
Edge Detection

Now that you've seen how to define and use image filters for smoothing images and detecting the edges (high-frequency) components of objects in an image, let's move one step further. The next few videos will be all about how we can use what we know about pattern recognition in images to begin identifying unique shapes and then objects.

Edges to Boundaries and Shapes

We know how to detect the edges of objects in images, but how can we begin to find unifying boundaries around objects? We'll want to be able to do this to separate and locate multiple objects in a given image. Next, we'll discuss the Hough transform, which transforms image data from the x-y coordinate system into Hough space, where you can easily identify simple boundaries like lines and circles.

The Hough transform is used in a variety of shape-recognition applications, as seen in the images pictured below. On the left you see how a Hough transform can find the edges of a phone screen and on the right you see how it's applied to an aerial image of farms (green circles in this image).



Hough transform applied to phone-edge and circular farm recognition.

Feature Extraction and Object Recognition

So, you've seen how to detect consistent shapes with something like the Hough transform that transforms shapes in x-y coordinate space into intersecting lines in Hough space. You've also gotten experience programming your own image filters to perform edge detection. Filtering images is a feature extraction technique because it filters out unwanted image information and extracts unique and identfiying features like edges or corners.

Extracting features and patterns in image data, using things like image filters, is the basis for many object recognition techniques. In the image below, we see a classification pipeline that is looking at an image of a banana; the image first goes through some filters and processing steps to form a feature that represent that banana, and this is used to help classify it. And we'll learn more about feature types and extraction methods in the next couple lessons.

Training data (an image of a banana) going through some feature extraction and classification steps.

Haar Cascade and Face Recognition

In the next video, we'll see how we can use a feature-based classifier to do face recognition.

The method we'll be looking at is called a **Haar cascade classifier**. It's a machine learning based approach where a cascade function is trained to solve a binary classification problem: face or not-face; it trains on a lot of positive (face) and negative (not-face) images, as seen below.

Images of faces and not-faces, going some feature extraction steps.

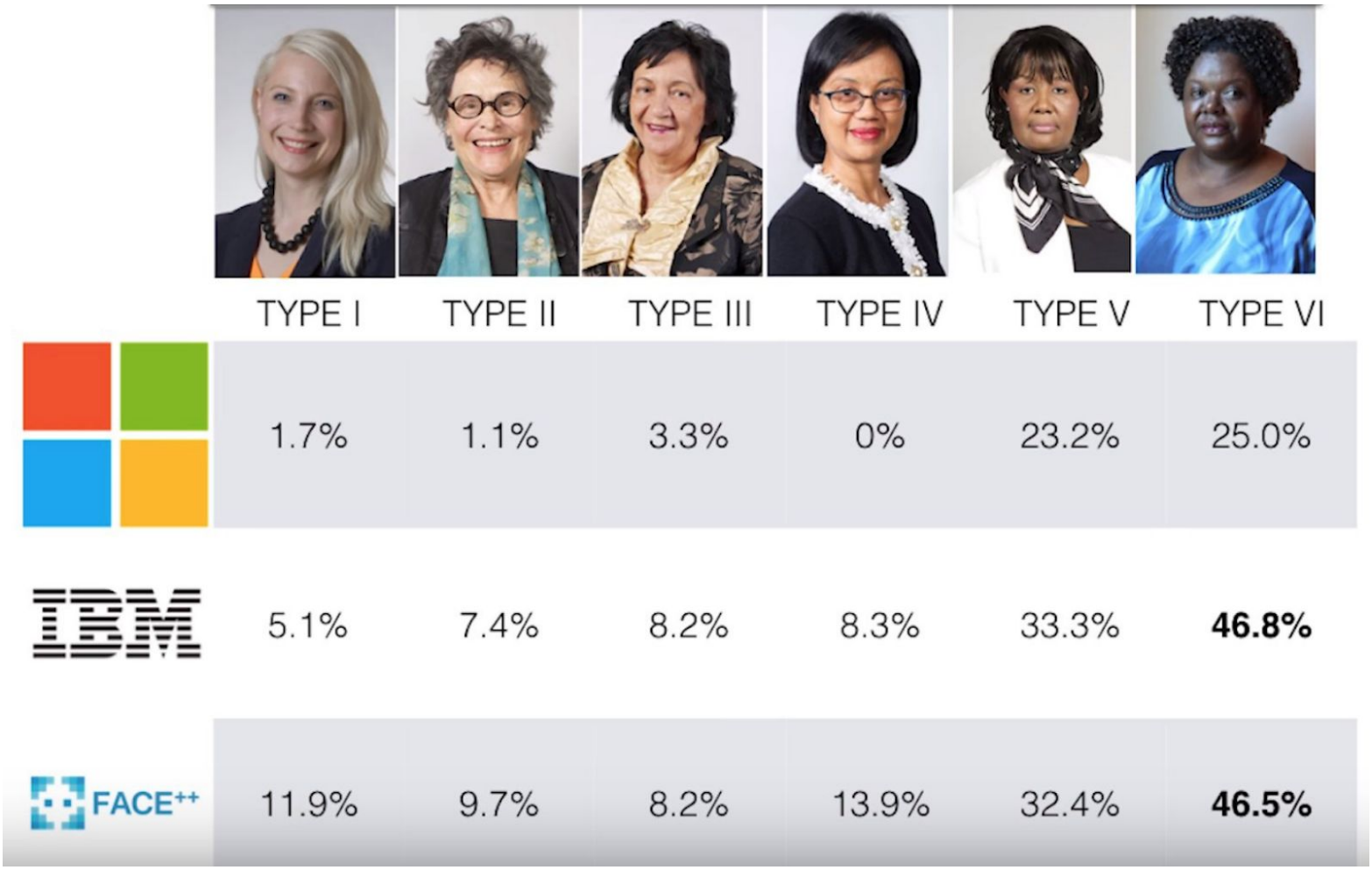
After the classifier sees an image of a face or not-face, it extracts features from it. For this, Haar filters shown in the below image are used. They are just like the image filters you've programmed! A new, filtered image is produced when the input image is convolved with one of these filters at a time.

Algorithms with Human and Data Bias

Most of the models you've seen and/or programmed, rely on large sets of data to train and learn. When you approach a challenge, it's up to you as a programmer, to define functions and a model for classifying image data. Programmers and data define how classification algorithms like face recognition work.

It's important to note that both data and humans come with their own biases, with unevenly distributed image types or personal preferences, respectively. And it's important to note that these biases propagate into the creation of algorithms. If we consider face recognition, think about the case in which a model like a Haar Cascade is trained on faces that are mainly white and female; this network will then excel at detecting those kinds of faces but not others. If this model is meant for general face recognition, then the biased data has ended up creating a biased model, and algorithms that do not reflect the diversity of the users it aims to serve is not very useful at all.

The computer scientist, Joy Buolamwini, based out of the MIT Media Lab, has studied bias in decision-making algorithms, and her work has revealed some of the extent of this problem. One study looked at the error rates of facial recognition programs for women by shades of skin color; results pictured below.



	TYPE I	TYPE II	TYPE III	TYPE IV	TYPE V	TYPE VI
Microsoft	1.7%	1.1%	3.3%	0%	23.2%	25.0%
IBM	5.1%	7.4%	8.2%	8.3%	33.3%	**46.8%**
FACE++	11.9%	9.7%	8.2%	13.9%	32.4%	**46.5%**

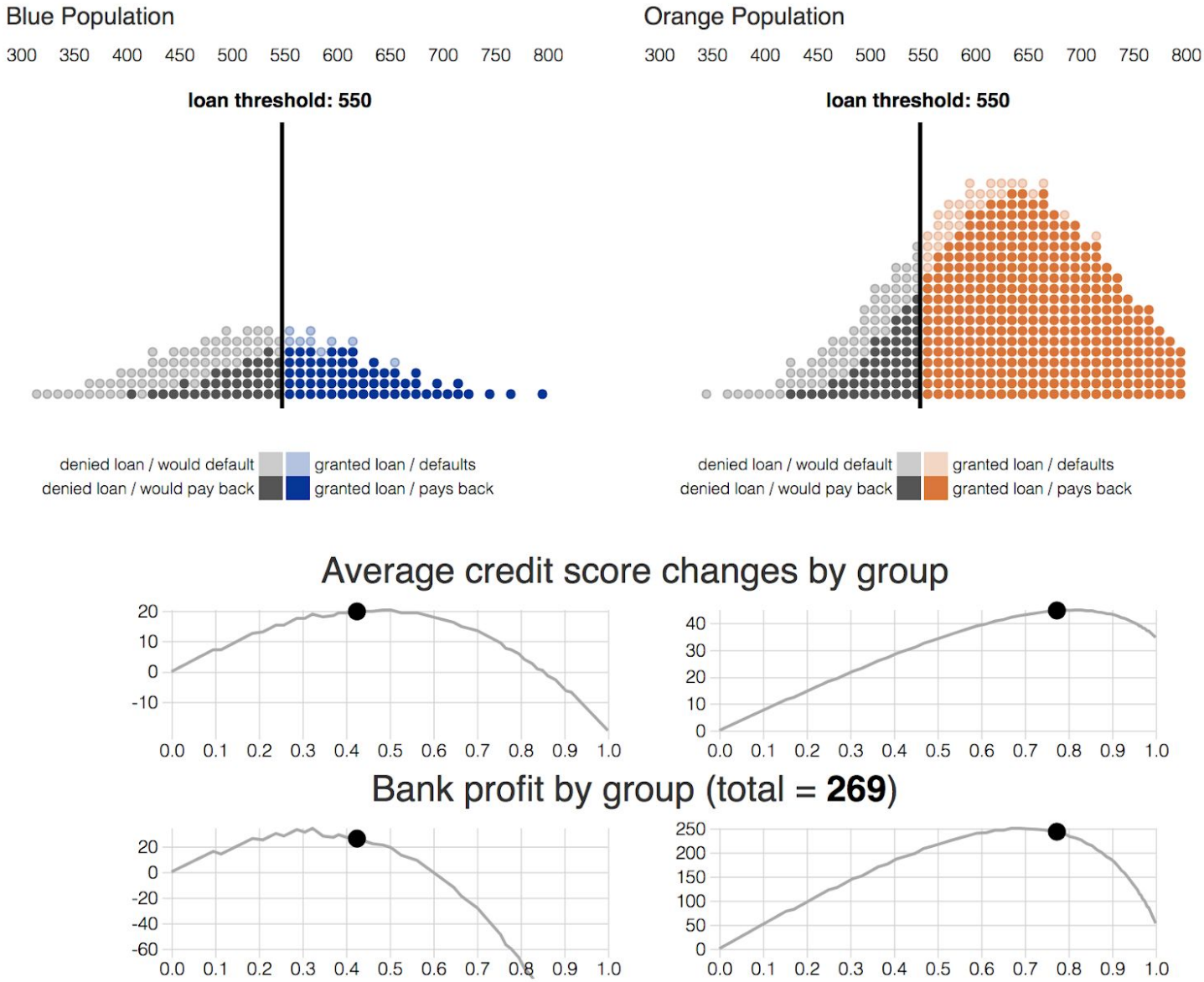
Image of facial recognition error rates, taken from MIT Media Lab's

gender shades website.

Analyzing Fairness

Identifying the fairness of a given algorithm is an active area of research. Here is an example of using a GAN (Generative Adversarial Network) to help a classifier detect bias and correct it's predictions: Implementing a fair classifier in PyTorch. And another paper that shows how "fair" credit loans affect diff populations (with helpful,

interactive plots). I think that as computer vision becomes more ubiquitous, this area of research will become more and more important, and it is worth reading about and educating yourself!



From credit loan paper, Delayed Impact of Fair Machine Learning.

Working to Eliminate Bias

Biased results are the effect of bias in programmers and in data, and we can work to change this. We must be critical of our own work, critical of what we read, and develop methods for testing such algorithms. As you learn more about AI and deep learning models, you'll learn some methods for visualizing what a neural network has learned, and you're encouraged to look at your data and make sure that it is balanced; data is the foundation for

any machine and deep learning model. It's also good practice to test any algorithm for bias; as you develop deep learning models, it's a good idea to test how they respond to a variety of challenges and see if they have any weaknesses.

If you'd like to learn about eliminating bias in AI, check out this Harvard Business Review article. I'd also recommend listening to Joy Buolamwini's TED talk and reading the original Gender Shades paper.

Further Reading

If you are really curious about bias in algorithms, there are also some excellent books on ethics in software engineering:

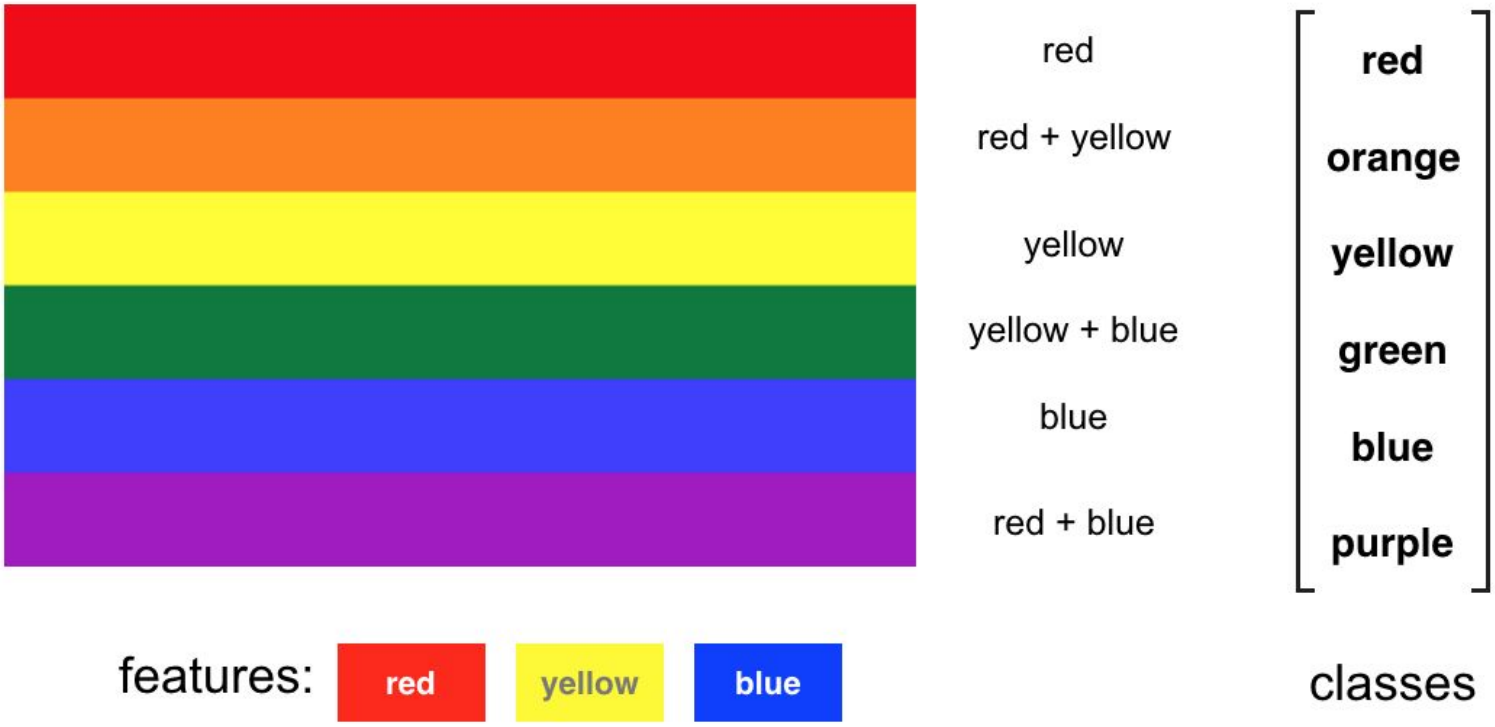
- Weapons of Math Destruction, Cathy O'Neil
- Algorithms of Oppression, Safiya Umoja Noble
- Automating Inequality, Virginia Eubanks
- Technically Wrong, Sara Wachter-Boettchera

Features

Features and feature extraction is the basis for many computer vision applications. The idea is that any set of data, such as a set of images, can be represented by a smaller, simpler model made of a combination of visual features: a few colors and shapes. (This is true with one exception: completely random data!)

If you can find a good model for any set of data, then you can start to find ways to identify patterns in data based on similarities and differences in the features in an image. This is especially important when we get to deep learning models for image classification, which you'll see soon.

Below is an example of a simple model for rainbow colors. Each of the colors below is actually a combination of a smaller set of color features: red, yellow, and blue. For example, purple = red + blue. And these simple features give us a way to represent a variety of colors and classify them accoring to their red, yellow, and blue components!



Color classification model.

Types of Features

We've described features as measurable pieces of data in an image that help distinguish between different classes of images.

There are two main types of features:

1. Color-based and

2. Shape-based

Both of these are useful in different cases and they are often powerful together. We know that color is all you need should you want to classify day/night images or implement a green screen. Let's look at another example: say I wanted to classify a stop sign vs. any other traffic sign. Stop signs are *supposed* to stand out in color and shape! A stop sign is an octagon (it has 8 flat sides) and it is very red. It's red color is often enough to distinguish it, but the sign can be obscured by trees or other artifacts and the shape ends up being important, too.

As a different example, say I want to detect a face and perform facial recognition. I'll first want to detect a face in a given image; this means at least recognizing the boundaries and some features on that face, which are all determined by shape. Specifically, I'll want to identify the edges of the face and the eyes and mouth on that face, so that I can identify the face and recognize it. Color is not very useful in this case, but shape is critical.

A note on shape

Edges are one of the simplest shapes that you can detect; edges often define the boundaries between objects but they may not provide enough information to find and identify small features on those objects (such as eyes on a face) and in the next videos, we'll look at methods for finding even more complex shapes.

As you continue learning, keep in mind that selecting the right feature is an important computer vision task.

Example Application: Lane Finding

You've already had some practice with this concept, but you can use feature/edge detection and color transforms to very effectively detect lane lines on a road. If you'd like to learn more about this technique, I suggest checking out this blog post.

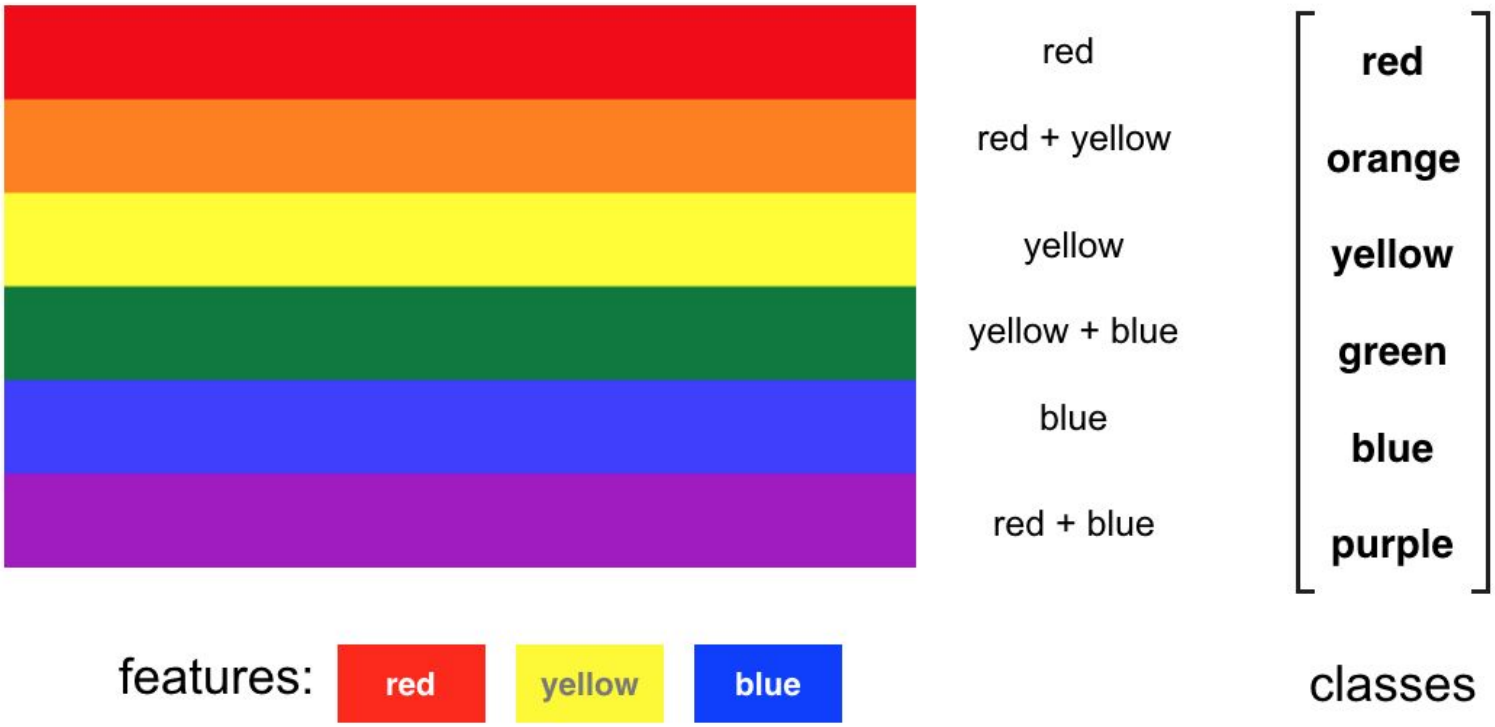
Identifying edges and lane markings on a road.

Features

Features and feature extraction is the basis for many computer vision applications. The idea is that any set of data, such as a set of images, can be represented by a smaller, simpler model made of a combination of visual features: a few colors and shapes. (This is true with one exception: completely random data!)

If you can find a good model for any set of data, then you can start to find ways to identify patterns in data based on similarities and differences in the features in an image. This is especially important when we get to deep learning models for image classification, which you'll see soon.

Below is an example of a simple model for rainbow colors. Each of the colors below is actually a combination of a smaller set of color features: red, yellow, and blue. For example, purple = red + blue. And these simple features give us a way to represent a variety of colors and classify them accoring to their red, yellow, and blue components!



Color classification model.

Types of Features

We've described features as measurable pieces of data in an image that help distinguish between different classes of images.

There are two main types of features:

1. Color-based and
2. Shape-based

Both of these are useful in different cases and they are often powerful together. We know that color is all you need should you want to classify day/night images or implement a green screen. Let's look at another example: say I

wanted to classify a stop sign vs. any other traffic sign. Stop signs are *supposed* to stand out in color and shape! A stop sign is an octagon (it has 8 flat sides) and it is very red. It's red color is often enough to distinguish it, but the sign can be obscured by trees or other artifacts and the shape ends up being important, too.

As a different example, say I want to detect a face and perform facial recognition. I'll first want to detect a face in a given image; this means at least recognizing the boundaries and some features on that face, which are all determined by shape. Specifically, I'll want to identify the edges of the face and the eyes and mouth on that face, so that I can identify the face and recognize it. Color is not very useful in this case, but shape is critical.

A note on shape

Edges are one of the simplest shapes that you can detect; edges often define the boundaries between objects but they may not provide enough information to find and identify small features on those objects (such as eyes on a face) and in the next videos, we'll look at methods for finding even more complex shapes.

As you continue learning, keep in mind that selecting the right feature is an important computer vision task.

Example Application: Lane Finding

You've already had some practice with this concept, but you can use feature/edge detection and color transforms to very effectively detect lane lines on a road. If you'd like to learn more about this technique, I suggest checking out this blog post.

Identifying edges and lane markings on a road.

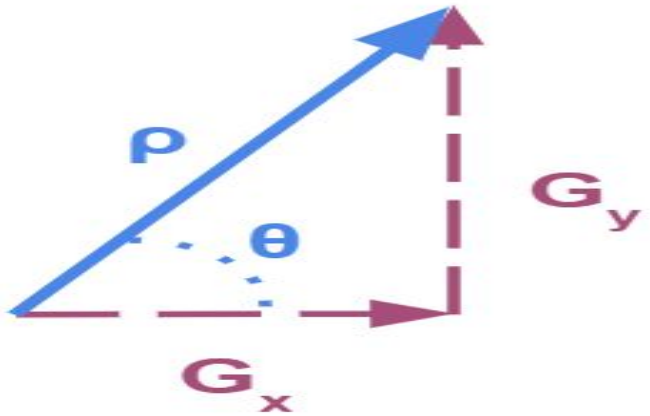
Correction: The magnitude should be calculated as the sum of the Gx and Gy components,

`rho = sqrt(Gx^2 + Gy^2)`.

Summary

A corner can be located by following these steps:

- Calculate the gradient for a small window of the image, using sobel-x and sobel-y operators (without applying binary thesholding).
- Use vector addition to calculate the magnitude and direction of the *total* gradient from these two values.



- Apply this calculation as you slide the window across the image, calculating the gradient of each window. When a big variation in the direction & magnitude of the gradient has been detected - a corner has been found!

Further Reading

You can learn more about Harris Corner Detection in OpenCV, here.

Documentation for OpenCV Harris Corner Detection can be found here.

Dilation and Erosion

Dilation and erosion are known as **morphological operations**. They are often performed on binary images, similar to contour detection. Dilation enlarges bright, white areas in an image by adding pixels to the perceived boundaries of objects in that image. Erosion does the opposite: it removes pixels along object boundaries and shrinks the size of objects.

Often these two operations are performed in sequence to enhance important object traits!

Dilation

To dilate an image in OpenCV, you can use the `dilate` function and three inputs: an original binary image, a kernel that determines the size of the dilation (None will result in a default size), and a number of iterations to perform the dilation (typically = 1). In the below example, we have a 5x5 kernel of ones, which move over an image, like a filter, and turn a pixel white if any of its surrounding pixels are white in a 5x5 window! We'll use a simple image of the cursive letter "j" as an example.

```
# Reads in a binary image
```

```
image = cv2.imread('j.png', 0)



# Create a 5x5 kernel of ones

kernel = np.ones((5,5),np.uint8)



# Dilate the image

dilation = cv2.dilate(image, kernel, iterations = 1)
```

Erosion

To erode an image, we do the same but with the `erode` function.

```
# Erode the image

erosion = cv2.erode(image, kernel, iterations = 1)
```

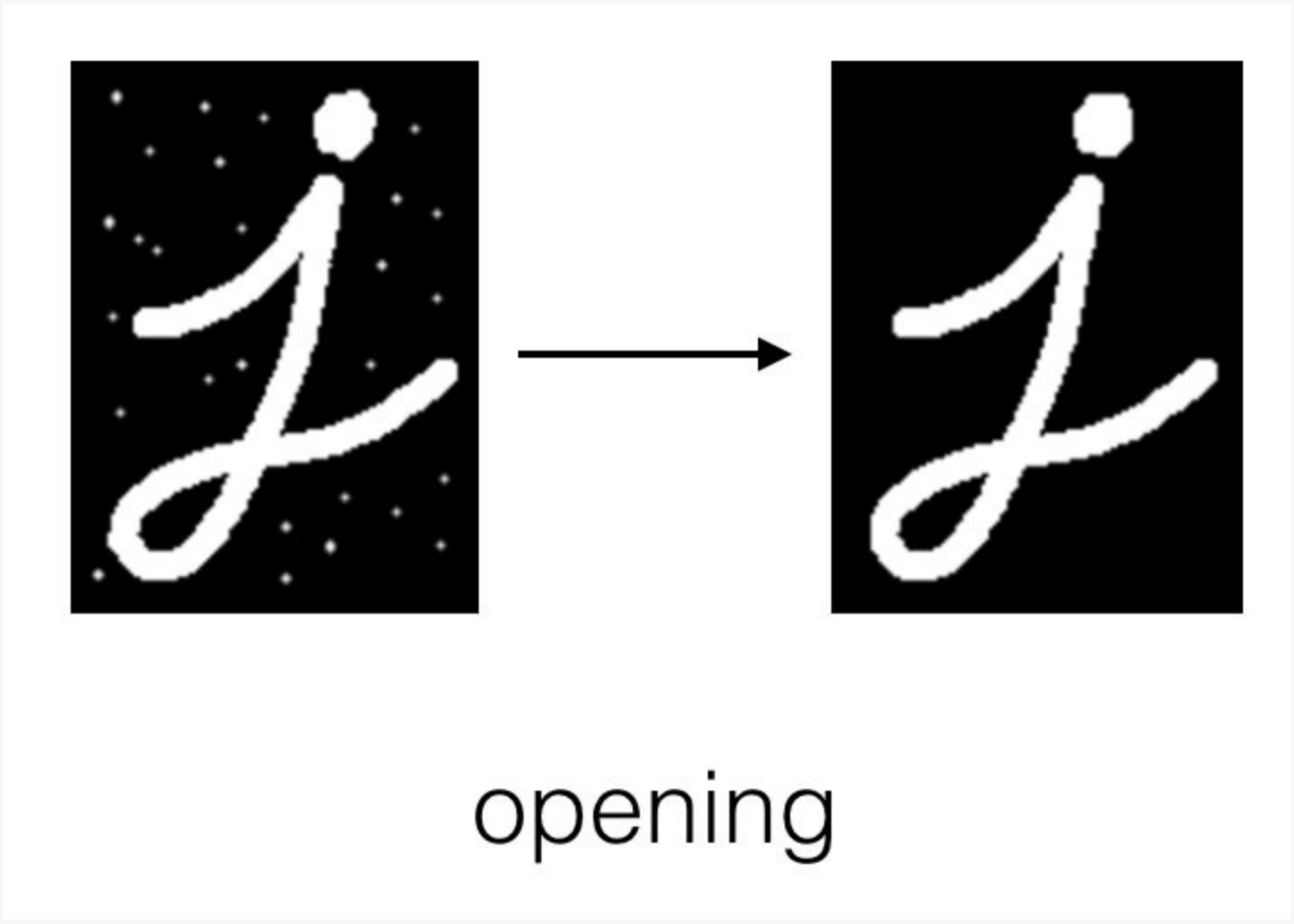
The letter "j": (left) erosion, (middle) original image, (right) dilation

Opening

As mentioned, above, these operations are often *combined* for desired results! One such combination is called **opening**, which is **erosion followed by dilation**. This is useful in noise reduction in which erosion first gets rid of noise (and shrinks the object) then dilation enlarges the object again, but the noise will have disappeared from the previous erosion!

To implement this in OpenCV, we use the function `morphologyEx` with our original image, the operation we want to perform, and our kernel passed in.

```
opening = cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel)
```



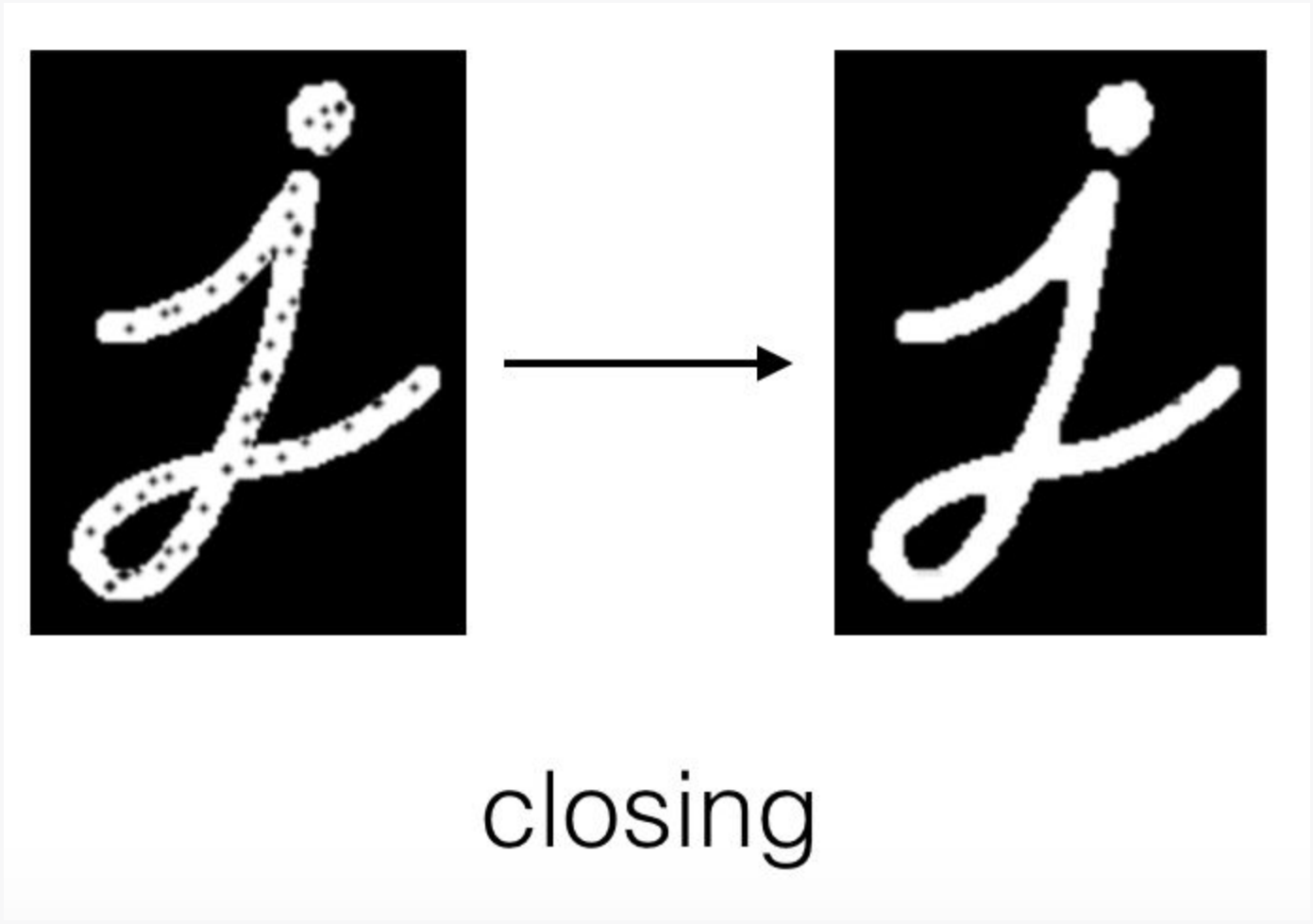
Opening

Closing

Closing is the reverse combination of opening; it's **dilation followed by erosion**, which is useful in *closing* small holes or dark areas within an object.

Closing is reverse of Opening, Dilation followed by Erosion. It is useful in closing small holes inside the foreground objects, or small black points on the object.

```
closing = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel)
```



Closing

Many of these operations try to extract better (less noisy) information about the shape of an object or enlarge important features, as in the case of corner detection!

Image Segmentation

Now that we are familiar with a few simple feature types, it may be useful to look at how we can group together different parts of an image by using these features. Grouping or segmenting images into distinct parts is known as image segmentation.

The simplest case for image segmentation is in background subtraction. In video and other applications, it is often the case that a human has to be isolated from a static or moving background, and so we have to use segmentation methods to distinguish these areas. Image segmentation is also used in a variety of complex recognition tasks, such as in classifying every pixel in an image of the road.

In the next few videos, we'll look at a couple ways to segment an image:

1. using contours to draw boundaries around different parts of an image, and
2. clustering image data by some measure of color or texture similarity.

Partially-segmented image of a road; the image separates areas that contain a pedestrian from areas in the image that contain the street or cars.

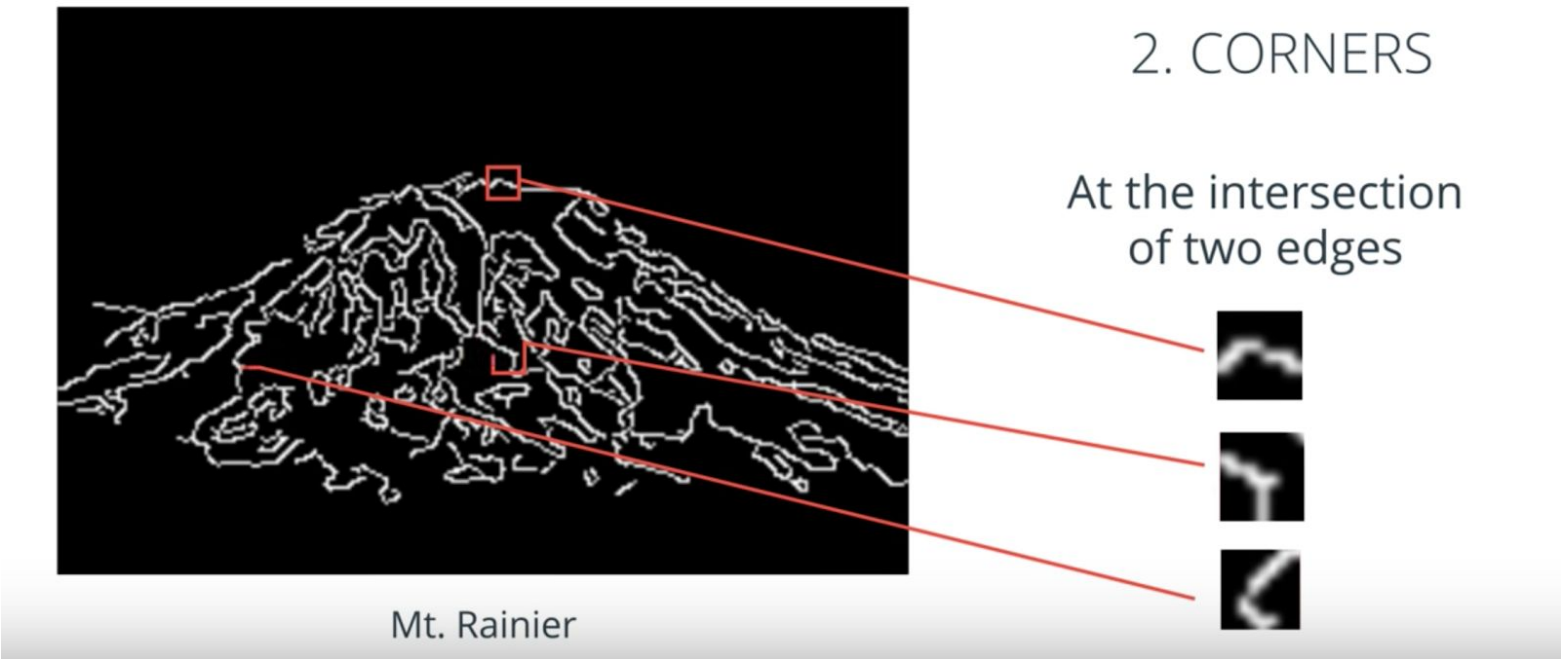
Contours and Features

Explore the OpenCV documentation to learn all about the information that contour detection provides!

Features alone and together

Now, you've seen examples of shape-based features, like corners, that can be extracted from images, but how can we actually uses the features to detect whole objects?

Well, let's think about an example we've seen of corner detection for an image of a mountain.



Corner detection on an image of Mt. Rainier

Say we want a way to detect this mountain in other images, too. A single corner will not be enough to identify this mountain in any other images, but, we can take a **set of features that define the shape of this mountain, group them together into an array or vector, and then use that set of features to create a mountain detector**!

In this lesson, you'll learn how to create feature vectors that can then be used to recognize different objects.

Learning to Find Features

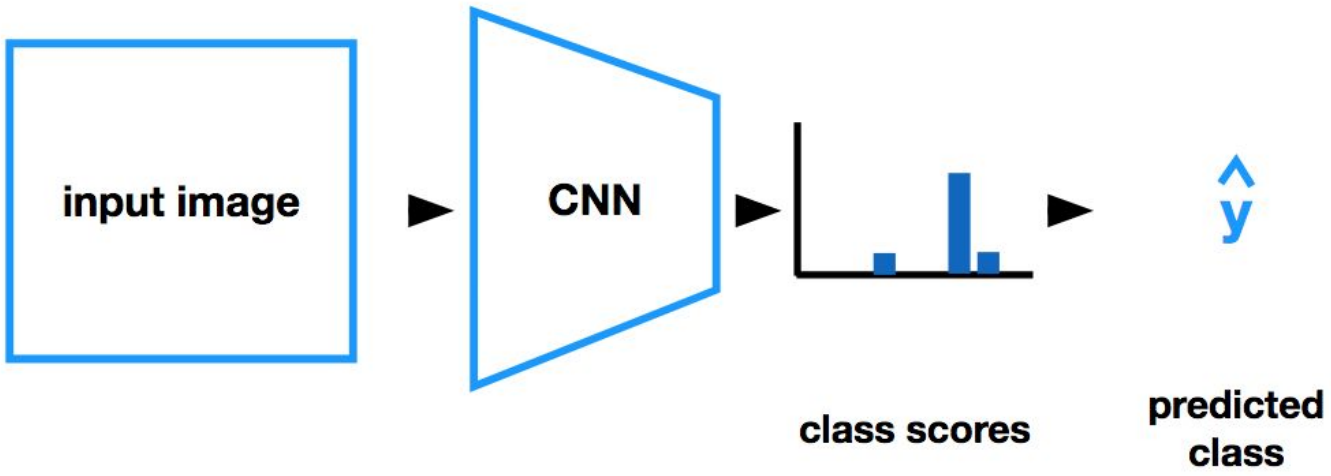
Now that you've seen a number of feature extraction techniques, you should have a good understanding of how different objects and areas in an image can be identified by their unique shapes and colors.

Convolutional filters and ORB and HOG descriptors all rely on patterns of intensity to identify different shapes (like edges) and eventually whole objects (with feature vectors). You've even seen how k-means clustering can be used to group data without any labels.

Next, we'll see how to define and train a Convolutional Neural Network (CNN) that *learns* to extract important features from images.

CNN Structure

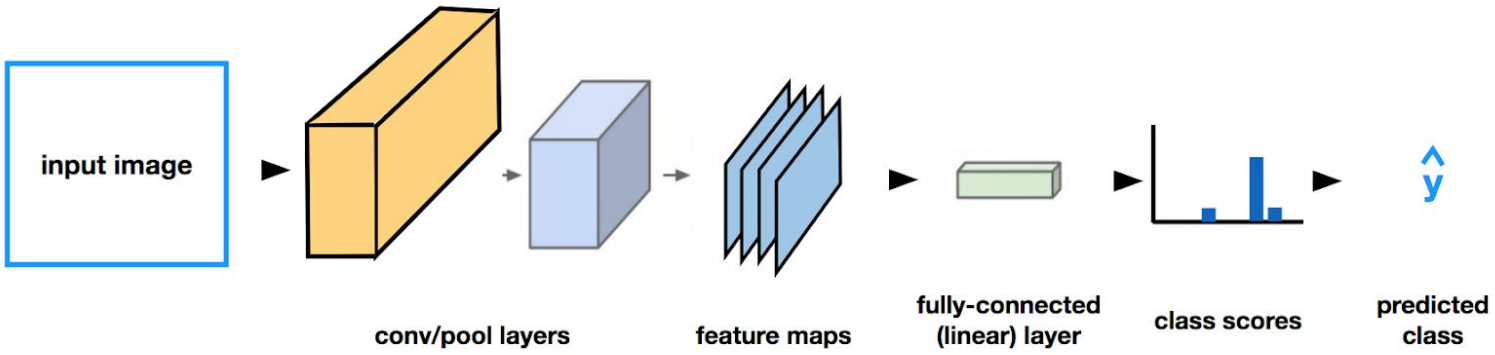
A classification CNN takes in an input image and outputs a distribution of class scores, from which we can find the most likely class for a given image. As you go through this lesson, you may find it useful to consult this blog post, which describes the image classification pipeline and the layers that make up a CNN.



A classification CNN structure.

CNN Layers

The CNN itself is comprised of a number of layers; layers that extract features from input images, reduce the dimensionality of the input, and eventually produce class scores. In this lesson, we'll go over all of these different layers, so that you know how to define and train a complete CNN!



Detailed layers that make up a classification CNN.

Elective: Review and Learn PyTorch

For this course, you are expected to know how neural networks train through backpropagation and have an idea of what loss functions you can use to train a CNN for a classification task. If you'd like to review this material, you can look at the **Elective Section, Review: Training a Neural Network** (at the bottom of all the lessons on the main course page) and pay close attention to the videos in that section.

This review section covers:

- How neural networks train and update their weights through backpropagation
- How to construct models in PyTorch

So, if you'd like a brief overview of neural networks and deep learning *or* if the PyTorch framework is new to you, make sure to take a look at that section!

PyTorch icon.

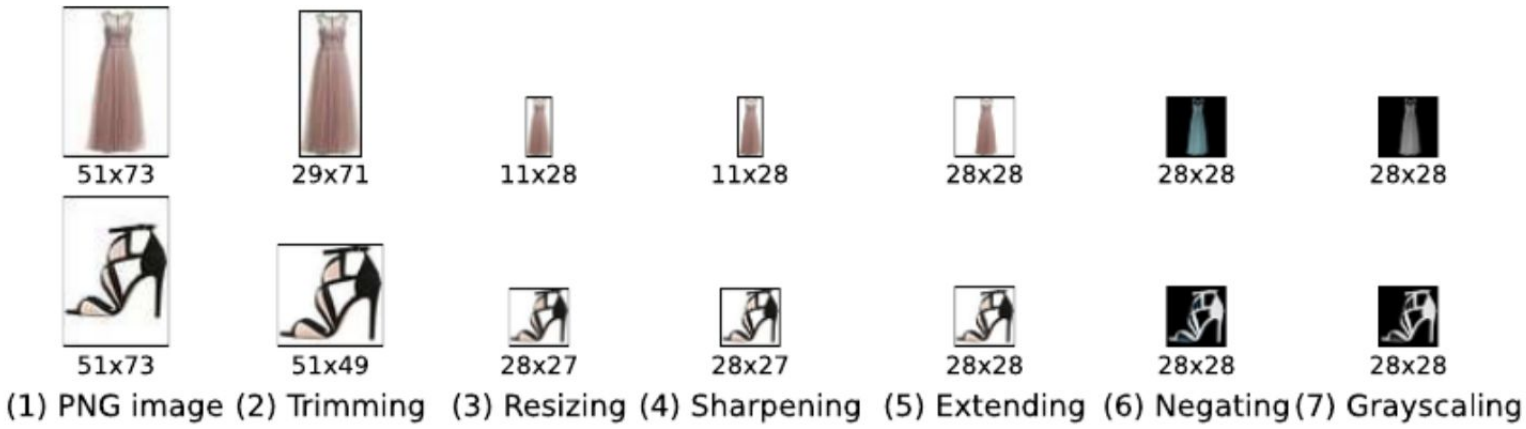
Why PyTorch?

We'll be using PyTorch throughout this program. PyTorch is definitely a newer framework, but it's fast and intuitive when compared to Tensorflow variables and sessions. PyTorch is designed to look and act a lot like normal Python code: PyTorch neural nets have their layers and feedforward behavior defined in a class. defining a network in a class means that you can instantiate multiple networks, dynamically change the structure of a model, and these class functions are called during training and testing.

PyTorch is also great for testing different model architectures, which is highly encouraged in this course! PyTorch networks are modular, which makes it easy to change a single layer in a network or modify the loss function and see the effect on training. If you'd like to see a review of PyTorch vs. TensorFlow, I recommend this blog post.

Pre-processing

Look at the steps below to see how pre-processing plays a major role in the creation of this dataset.



Pre-processing steps for FashionMNIST data creation.

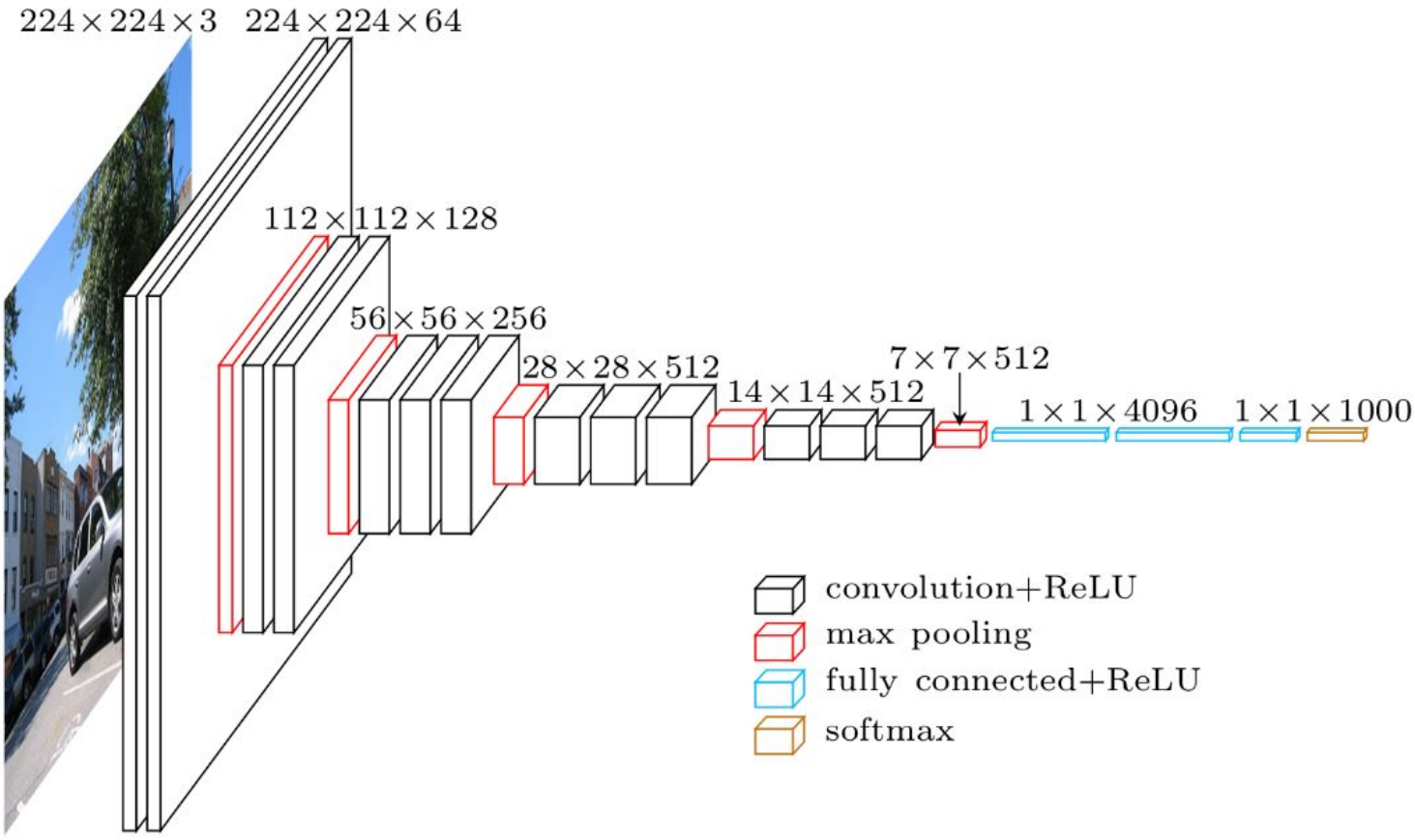
A note on Workspaces

As you go through the exercises in this section, please make sure your workspace is up-to-date. There is a button on the bottom-left of every workspace that will have a small red icon appear when an update is available. You may need to select this button and type `Reset data` to get the latest version of workable code. These updates only happen occasionally, for example, when a new version of PyTorch is released!

Convolutional Neural Networks (CNN's)

The type of deep neural network that is most powerful in image processing tasks, such as sorting images into groups, is called a Convolutional Neural Network (CNN). CNN's consist of layers that process visual information. A CNN first takes in an input image and then passes it through these layers. There are a few different types of layers, and we'll start by learning about the most commonly used layers: convolutional, pooling, and fully-connected layers.

First, let's take a look at a complete CNN architecture; below is a network called VGG-16, which has been trained to recognize a variety of image classes. It takes in an image as input, and outputs a predicted class for that image. The various layers are labeled and we'll go over each type of layer in this network in the next series of videos.



VGG-16 architecture

Convolutional Layer

The first layer in this network, that processes the input image directly, is a convolutional layer.

- A convolutional layer takes in an image as input.
- A convolutional layer, as its name suggests, is made of a set of convolutional filters (which you've already seen and programmed).
- Each filter extracts a specific kind of feature, ex. a high-pass filter is often used to detect the edge of an object.

- The output of a given convolutional layer is a set of **feature maps** (also called activation maps), which are filtered versions of an original input image.

Activation Function

You may also note that the diagram reads "convolution + ReLu," and the **ReLu** stands for Rectified Linear Unit (ReLU) activation function. This activation function is zero when the input x <= 0 and then linear with a slope = 1 when x > 0. ReLu's, and other activation functions, are typically placed after a convolutional layer to slightly transform the output so that it's more efficient to perform backpropagation and effectively train the network.

Introducing Alexishttps://classroom.udacity.com/nanodegrees/nd891/parts/0c8ef9cd-0234-49b5-bf10-bd13ea147191/modules/d1e2952d-7eff-4b54-b142-87fa82dd2092/lessons/68357628-74e3-439a-beb6-12884a1fb5d8/concepts/5b79e8f3-4b88-43c5-8e04-8034ac491fb9#

To help us learn about the layers that make up a CNN, I'm happy to introduce Alexis Cook. Alexis is an applied mathematician with M.S. in computer science from Brown University and an M.S. in applied mathematics from the University of Michigan. Next, she'll talk about convolutional and pooling layers.

Define a Network Architecture

The various layers that make up any neural network are documented, here. For a convolutional neural network, we'll use a simple series of layers:

- Convolutional layers
- Maxpooling layers
- Fully-connected (linear) layers

To define a neural network in PyTorch, you'll create and name a new neural network class, define the layers of the network in a function `__init__` and define the feedforward behavior of the network that employs those initialized layers in the function `forward`, which takes in an input image tensor, `x`. The structure of such a class, called `Net` is shown below.

Note: During training, PyTorch will be able to perform backpropagation by keeping track of the network's feedforward behavior and using autograd to calculate the update to the weights in the network.

```
import torch.nn as nn

import torch.nn.functional as F

class Net(nn.Module):



    def __init__(self, n_classes):

        super(Net, self).__init__()



        # 1 input image channel (grayscale), 32 output channels/feature maps

        # 5x5 square convolution kernel

        self.conv1 = nn.Conv2d(1, 32, 5)



        # maxpool layer

        # pool with kernel_size=2, stride=2
```

```
        self.pool = nn.MaxPool2d(2, 2)



        # fully-connected layer

        # 32*4 input size to account for the downsampled image size after pooling

        # num_classes outputs (for n_classes of image data)

        self.fc1 = nn.Linear(32*4, n_classes)



    # define the feedforward behavior

    def forward(self, x):

        # one conv/relu + pool layers

        x = self.pool(F.relu(self.conv1(x)))



        # prep for linear layer by flattening the feature maps into feature vectors

        x = x.view(x.size(0), -1)

        # linear layer

        x = F.relu(self.fc1(x))
```

```
        # final output

        return x



# instantiate and print your Net

n_classes = 20 # example number of classes

net = Net(n_classes)

print(net)
```

Let's go over the details of what is happening in this code.

Define the Layers in `__init__`

Convolutional and maxpooling layers are defined in `__init__`:

```
# 1 input image channel (for grayscale images), 32 output channels/feature maps, 3x3 square convolution kernel

self.conv1 = nn.Conv2d(1, 32, 3)



# maxpool that uses a square window of kernel_size=2, stride=2

self.pool = nn.MaxPool2d(2, 2)
```

Refer to Layers in `forward`

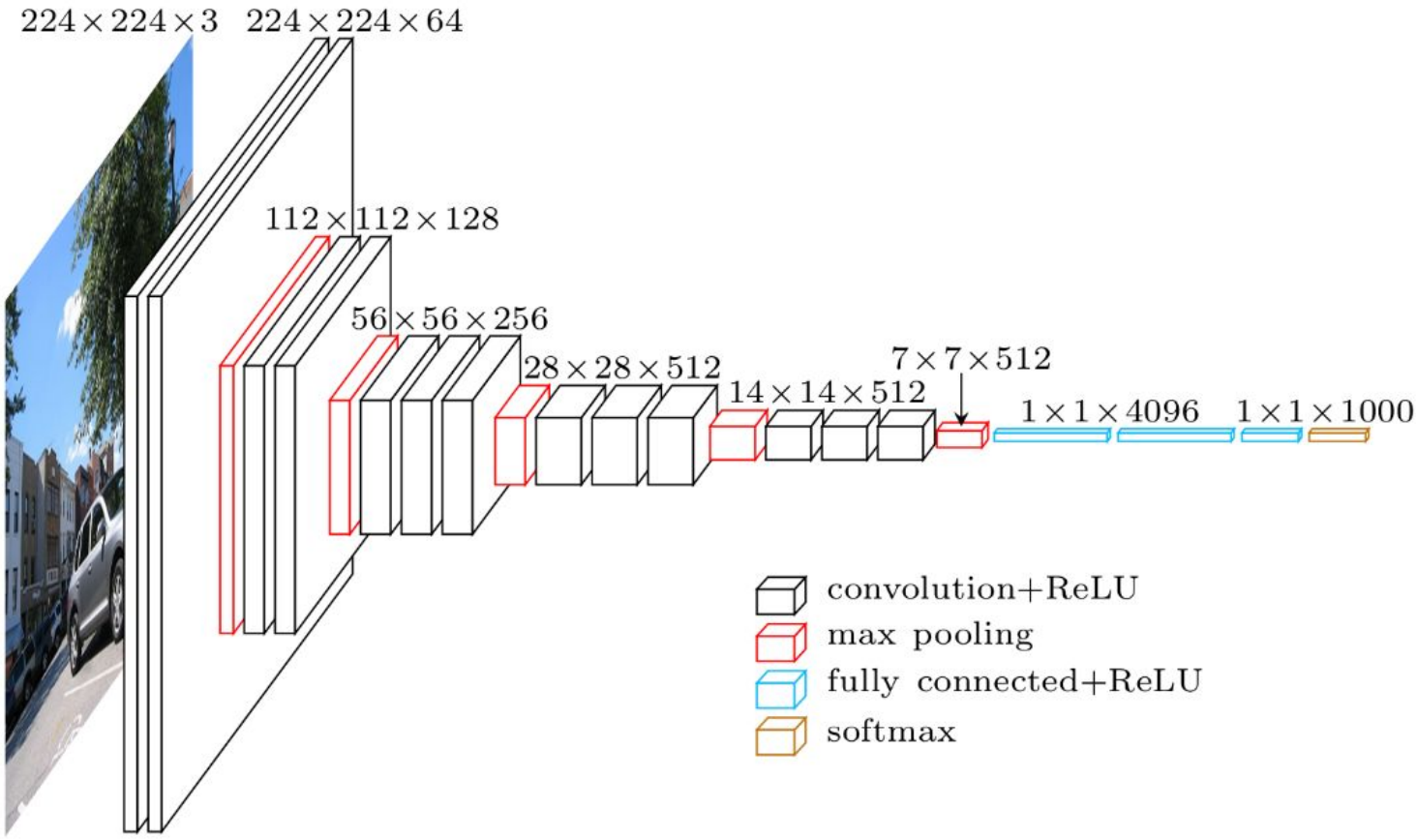
Then these layers are referred to in the `forward` function like this, in which the conv1 layer has a ReLu activation applied to it before maxpooling is applied:

```
x = self.pool(F.relu(self.conv1(x)))
```

Best practice is to place any layers whose weights will change during the training process in `__init__` and refer to them in the `forward` function; any layers or functions that always behave in the same way, such as a pre-defined activation function, may appear in the `__init__` or in the `forward` function; it is mostly a matter of style and readability.

VGG-16 Architecture

Take a look at the layers after the initial convolutional layers in the VGG-16 architecture.

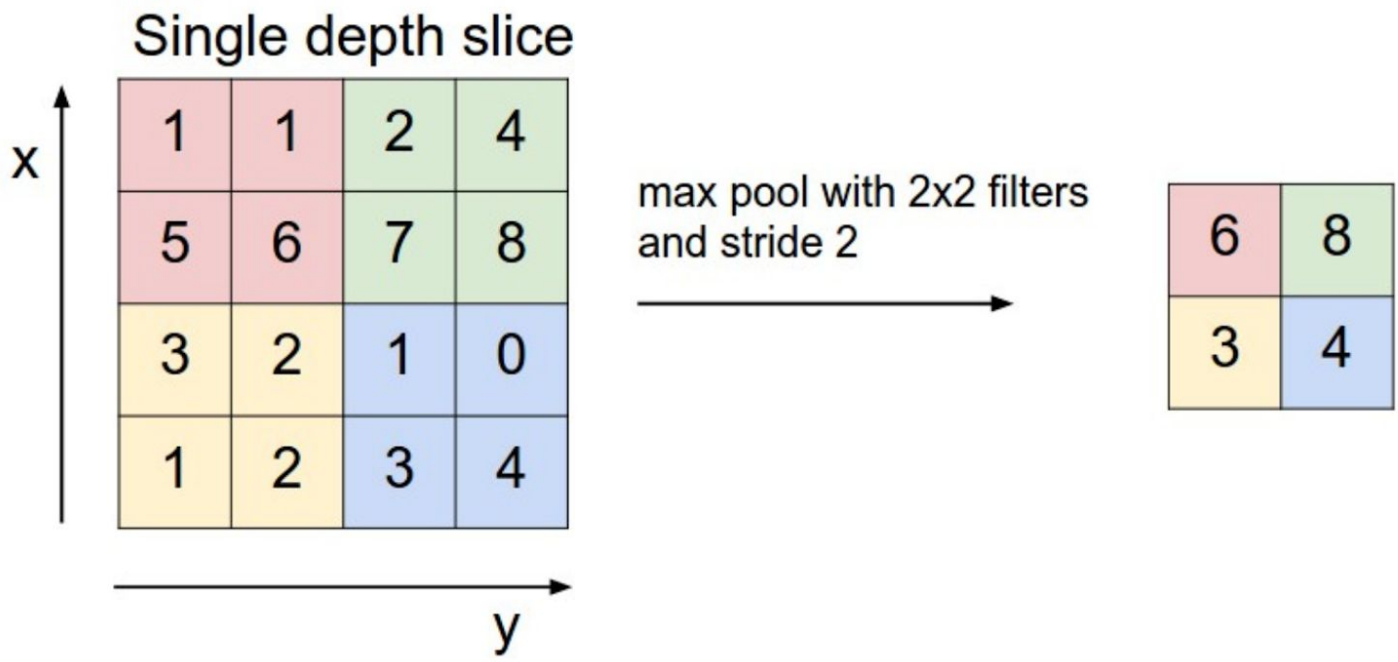


VGG-16 architecture

Pooling Layer

After a couple of convolutional layers (+ReLu's), in the VGG-16 network, you'll see a maxpooling layer.

- Pooling layers take in an image (usually a filtered image) and output a reduced version of that image
- Pooling layers reduce the dimensionality of an input
- **Maxpooling** layers look at areas in an input image (like the 4x4 pixel area pictured below) and choose to keep the maximum pixel value in that area, in a new, reduced-size area.
- Maxpooling is the most common type of pooling layer in CNN's, but there are also other types such as average pooling.

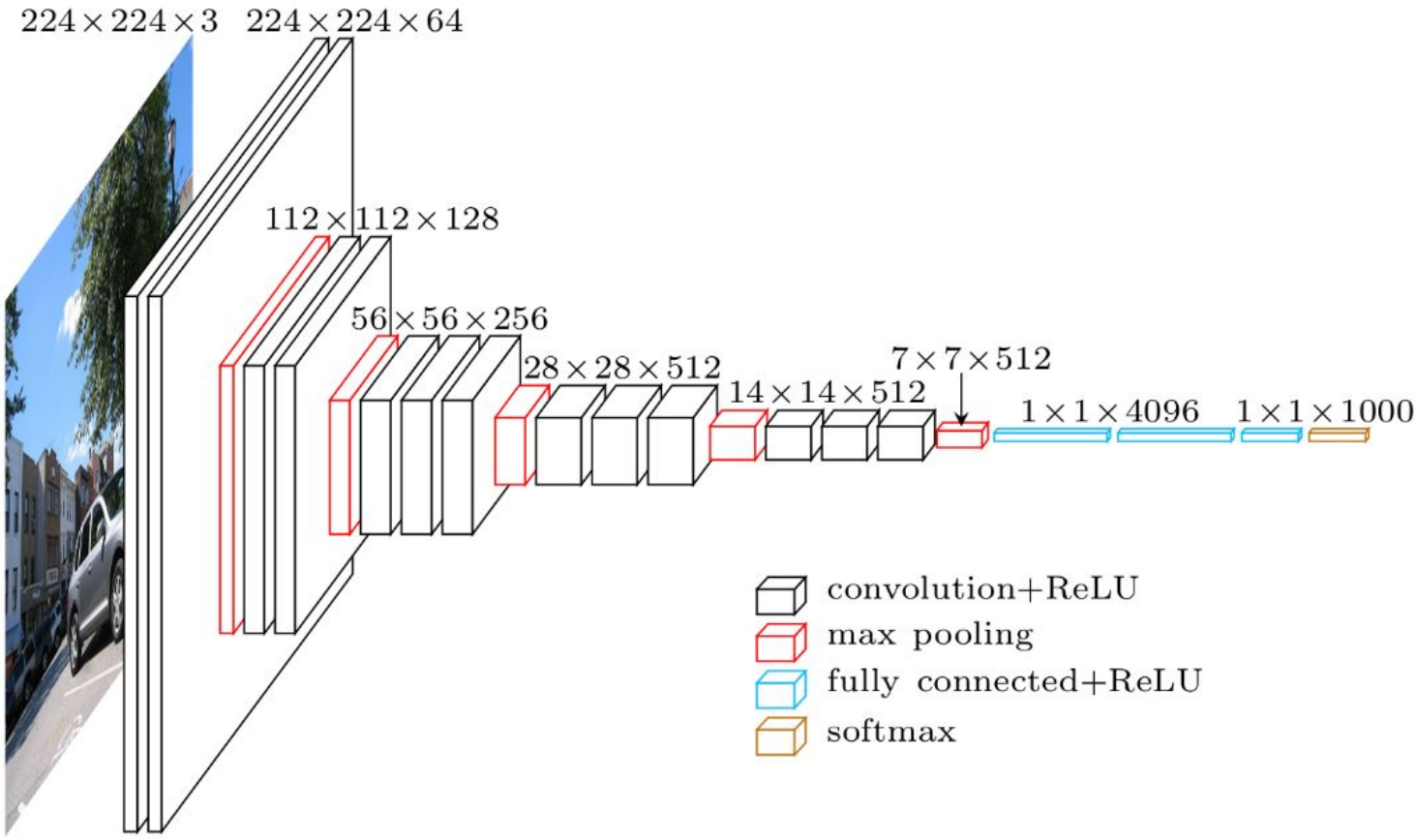


Maxpooling with a 2x2 area and stride of 2

* We'll link to PyTorch pooling documentation, since that's what we'll use in this course.

VGG-16 Architecture

Take a look at the layers near the end of this model; the fully-connected layers that come after a series of convolutional and pooling layers. Take note of their flattened shape.



VGG-16 architecture

Fully-Connected Layer

A fully-connected layer's job is to connect the input it sees to a desired form of output. Typically, this means converting a matrix of image features into a feature vector whose dimensions are 1xC, where C is the number of classes. As an example, say we are sorting images into ten classes, you could give a fully-connected layer a set of [pooled, activated] feature maps as input and tell it to use a combination of these features (multiplying them, adding them, combining them, etc.) to output a 10-item long feature vector. This vector compresses the information from the feature maps into a single feature vector.

Softmax

The *very* last layer you see in this network is a softmax function. The softmax function, can take any vector of values as input and returns a vector of the same length whose values are all in the range (0, 1) and, together,

these values will add up to 1. This function is often seen in classification models that have to turn a feature vector into a probability distribution.

Consider the same example again; a network that groups images into one of 10 classes. The fully-connected layer can turn feature maps into a single feature vector that has dimensions 1x10. Then the softmax function turns that vector into a 10-item long probability distribution in which each number in the resulting vector represents the probability that a given input image falls in class 1, class 2, class 3, ... class 10. This output is sometimes called the **class scores** and from these scores, you can extract the most likely class for the given image!

Overfitting

Convolutional, pooling, and fully-connected layers are all you need to construct a complete CNN, but there are additional layers that you can add to avoid overfitting, too. One of the most common layers to add to prevent overfitting is a dropout layer.

Dropout layers essentially turn off certain nodes in a layer with some probability, p. This ensures that all nodes get an equal chance to try and classify different images during training, and it reduces the likelihood that only a few, heavily-weighted nodes will dominate the process.

Now, you're familiar with all the major components of a complete convolutional neural network, and given some examples of PyTorch code, you should be well equipped to build and train your own CNN's! Next, it'll be up to you to define and train a CNN for clothing recognition!

Training in PyTorch

Once you've loaded a training dataset, next your job will be to define a CNN and train it to classify that set of images.

Loss and Optimizer

To train a model, you'll need to define *how* it trains by selecting a loss function and optimizer. These functions decide how the model updates its parameters as it trains and can affect how quickly the model converges, as well.

Learn more about loss functions and optimizers in the online documentation.

For a classification problem like this, one typically uses cross entropy loss, which can be defined in code like: `criterion = nn.CrossEntropyLoss()`. PyTorch also includes some standard stochastic optimizers like stochastic gradient descent and Adam. You're encouraged to try different optimizers and see how your model responds to these choices as it trains.

Clasisification vs. Regression

The loss function you should choose depends on the kind of CNN you are trying to create; cross entropy is generally good for classification tasks, but you might choose a different loss function for, say, a regression problem that tried to predict (x,y) locations for the center or edges of clothing items instead of class scores.

Training the Network

Typically, we train any network for a number of epochs or cycles through the training dataset

Here are the steps that a training function performs as it iterates over the training dataset:

1. Prepares all input images and label data for training
2. Passes the input through the network (forward pass)
3. Computes the loss (how far is the predicted classes are from the correct labels)
4. Propagates gradients back into the network’s parameters (backward pass)

5. Updates the weights (parameter update)

It repeats this process until the average loss has sufficiently decreased.

And in the next notebook, you'll see how to train and test a CNN for clothing classification, in detail.

Please also checkout the linked, exercise repo for multiple solutions to the following training challenge!

Dropout and Momentum

The next solution will show a different (improved) model for clothing classification. It has two main differences when compared to the first solution:

1. It has an additional dropout layer
2. It includes a momentum term in the optimizer: stochastic gradient descent

So, why are these improvements?

Dropout

Dropout randomly turns off perceptrons (nodes) that make up the layers of our network, with some specified probability. It may seem counterintuitive to throw away a connection in our network, but as a network trains, some nodes can dominate others or end up making large mistakes, and dropout gives us a way to balance our network so that every node works equally towards the same goal, and if one makes a mistake, it won't dominate the behavior of our model. You can think of dropout as a technique that makes a network resilient; it makes all the nodes work well as a team by making sure no node is too weak or too strong. In fact it makes me think of the Chaos Monkey tool that is used to test for system/website failures.

I encourage you to look at the PyTorch dropout documentation, here, to see how to add these layers to a network.

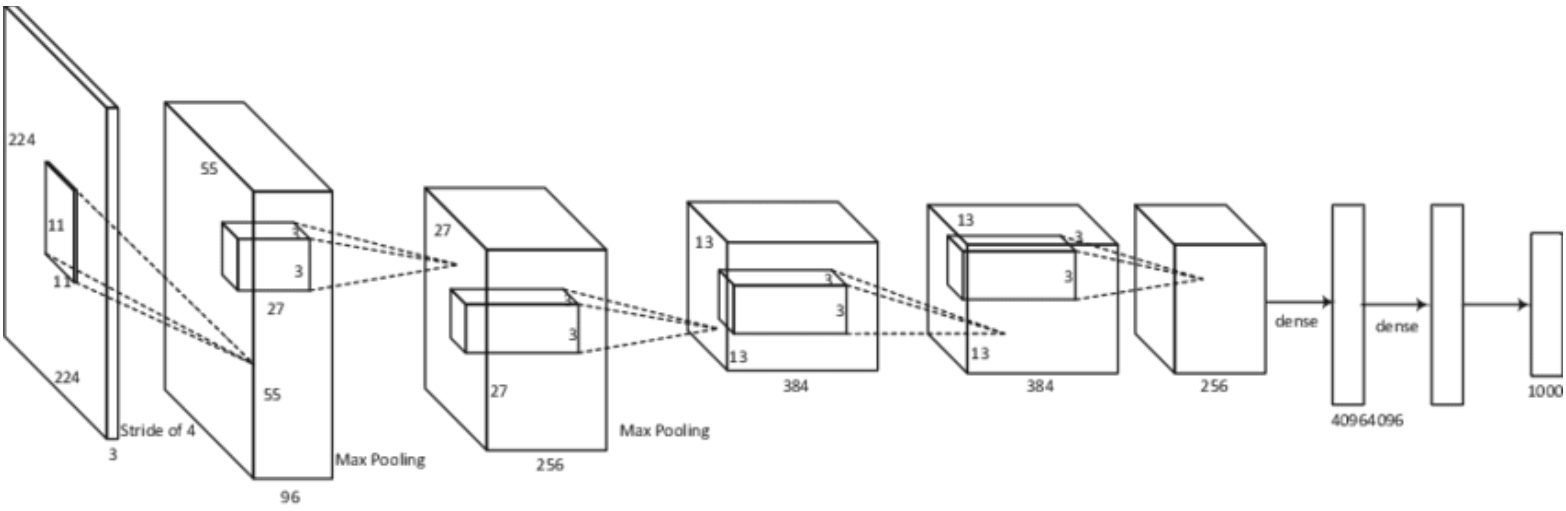
Momentum

When you train a network, you specify an optimizer that aims to reduce the errors that your network makes during training. The errors that it makes should generally reduce over time but there may be some bumps along the way. Gradient descent optimization relies on finding a local minimum for an error, but it has trouble finding the *global minimum* which is the lowest an error can get. So, we add a momentum term to help us find and then move on from local minimums and find the global minimum!

How can you decide on a network structure?

At this point, deciding on a network structure: how many layers to create, when to include dropout layers, and so on, may seem a bit like guessing, but there is a rationale behind defining a good model.

I think a lot of people (myself included) build up an intuition about how to structure a network from existing models. Take AlexNet as an example; linked is a nice, concise walkthrough of structure and reasoning.



AlexNet structure.

Preventing Overfitting

Often we see batch norm applied after early layers in the network, say after a set of conv/pool/activation steps since this normalization step is fairly quick and reduces the amount by which hidden weight values shift around.

Dropout layers often come near the end of the network; placing them in between fully-connected layers for example can prevent any node in those layers from overly-dominating.

Convolutional and Pooling Layers

As far as conv/pool structure, I would again recommend looking at existing architectures, since many people have already done the work of throwing things together and seeing what works. In general, more layers = you can see more complex structures, but you should always consider the size and complexity of your training data (many layers may not be necessary for a simple task).

As You Learn

When you are first learning about CNN's for classification or any other task, you can improve your intuition about model design by approaching a simple task (such as clothing classification) and *quickly* trying out new approaches. You are encouraged to:

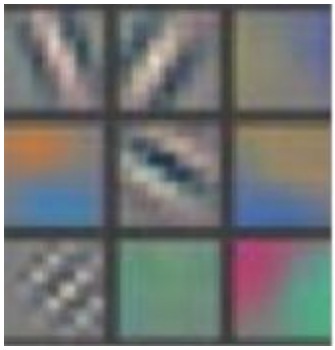
1. Change the number of convolutional layers and see what happens
2. Increase the size of convolutional kernels for larger images
3. Change loss/optimization functions to see how your model responds (especially change your **hyperparameters** such as learning rate and see what happens -- you will learn more about hyperparameters in the second module of this course)
4. Add layers to prevent overfitting
5. Change the batch_size of your data loader to see how larger batch sizes can affect your training

Always watch how **much** and how **quickly** your model loss decreases, and learn from improvements as well as mistakes!

Visualizing CNNs

Let’s look at an example CNN to see how it works in action.

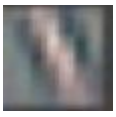
The CNN we will look at is trained on ImageNet as described in this paper by Zeiler and Fergus. In the images below (from the same paper), we’ll see *what* each layer in this network detects and see *how* each layer detects more and more complex ideas.



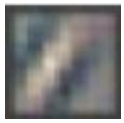
Example patterns that cause activations in the first layer of the network. These range from simple diagonal lines (top left) to green blobs (bottom middle).

The images above are from Matthew Zeiler and Rob Fergus' deep visualization toolbox, which lets us visualize what each layer in a CNN focuses on.

Each image in the above grid represents a pattern that causes the neurons in the first layer to activate - in other words, they are patterns that the first layer recognizes. The top left image shows a -45 degree line, while the middle top square shows a +45 degree line. These squares are shown below again for reference.

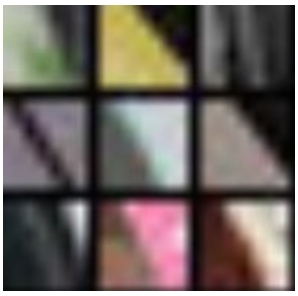


As visualized here, the first layer of the CNN can recognize -45 degree lines.



The first layer of the CNN is also able to recognize +45 degree lines, like the one above.

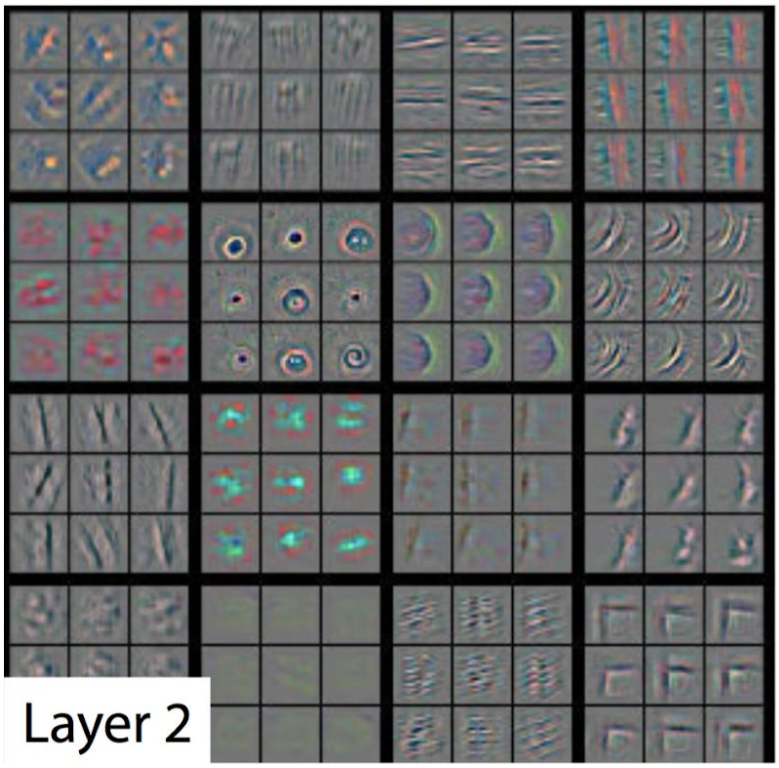
Let's now see some example images that cause such activations. The below grid of images all activated the -45 degree line. Notice how they are all selected despite the fact that they have different colors, gradients, and patterns.

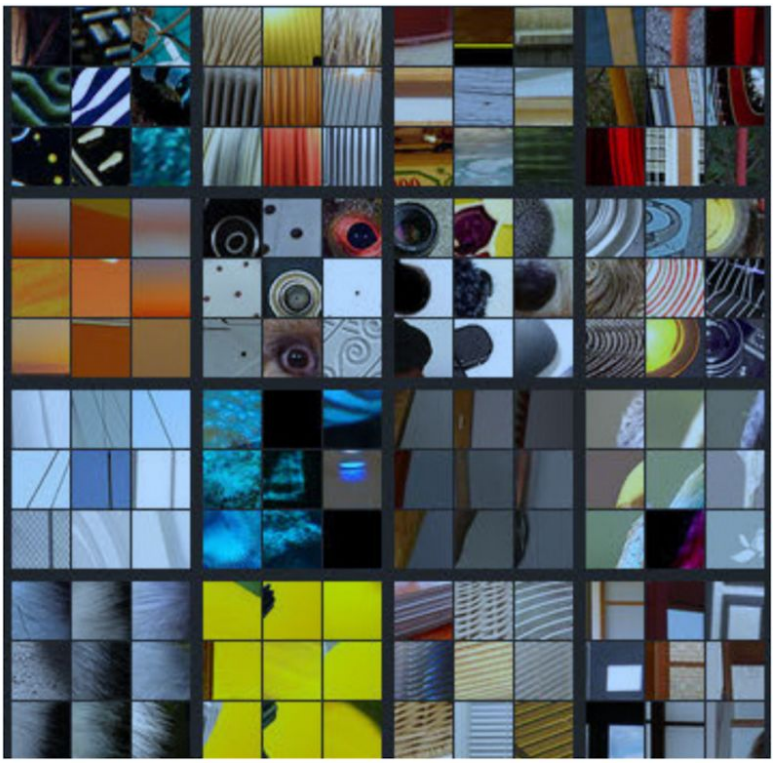


Example patches that activate the -45 degree line detector in the first layer.

So, the first layer of our CNN clearly picks out very simple shapes and patterns like lines and blobs.

Layer 2





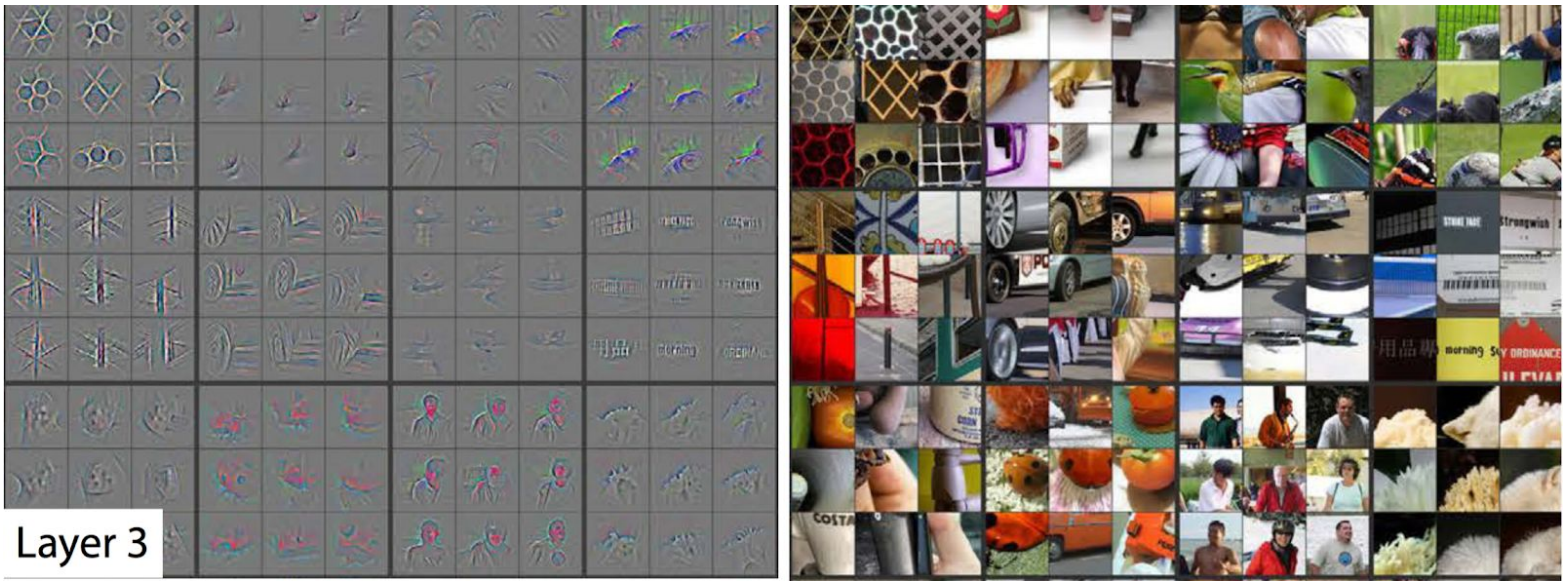
A visualization of the second layer in the CNN. Notice how we are picking up more complex ideas like circles and stripes. The gray grid on the left represents how this layer of the CNN activates (or "what it sees") based on the corresponding images from the grid on the right.

The second layer of the CNN captures complex ideas.

As you see in the image above, the second layer of the CNN recognizes circles (second row, second column), stripes (first row, second column), and rectangles (bottom right).

The CNN learns to do this on its own. There is no special instruction for the CNN to focus on more complex objects in deeper layers. That's just how it normally works out when you feed training data into a CNN.

Layer 3



A visualization of the third layer in the CNN. The gray grid on the left represents how this layer of the CNN activates (or "what it sees") based on the corresponding images from the grid on the right.

The third layer picks out complex combinations of features from the second layer. These include things like grids, and honeycombs (top left), wheels (second row, second column), and even faces (third row, third column).

We'll skip layer 4, which continues this progression, and jump right to the fifth and final layer of this CNN.

Layer 5

Layer 5

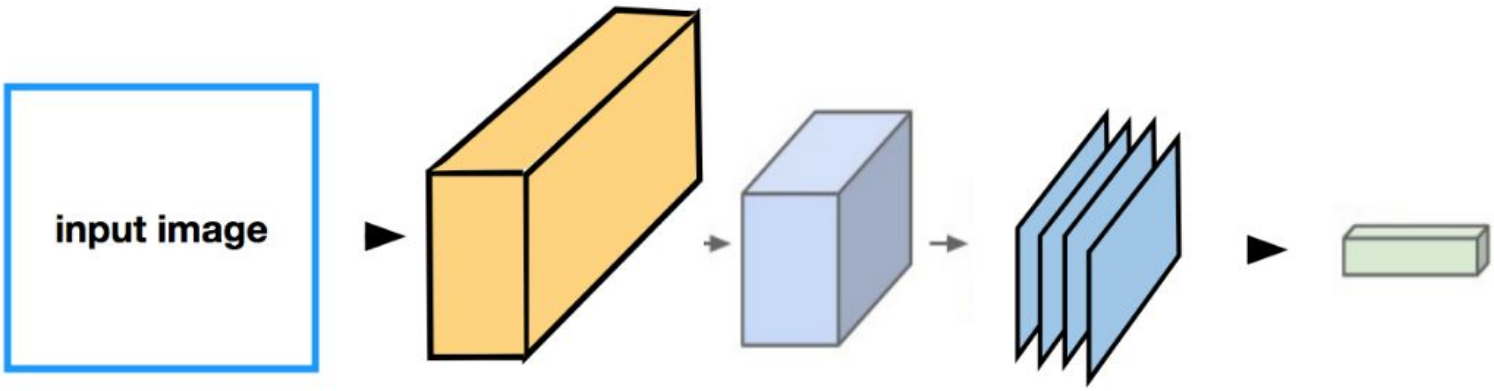
A visualization of the fifth and final layer of the CNN. The gray grid on the left represents how this layer of the CNN activates (or "what it sees") based on the corresponding images from the grid on the right.

The last layer picks out the highest order ideas that we care about for classification, like dog faces, bird faces, and bicycles.

Last Layer

In addition to looking at the first layer(s) of a CNN, we can take the opposite approach, and look at the last linear layer in a model.

We know that the output of a classification CNN, is a fully-connected class score layer, and one layer before that is a **feature vector that represents the content of the input image in some way**. This feature vector is produced after an input image has gone through all the layers in the CNN, and it contains enough distinguishing information to classify the image.



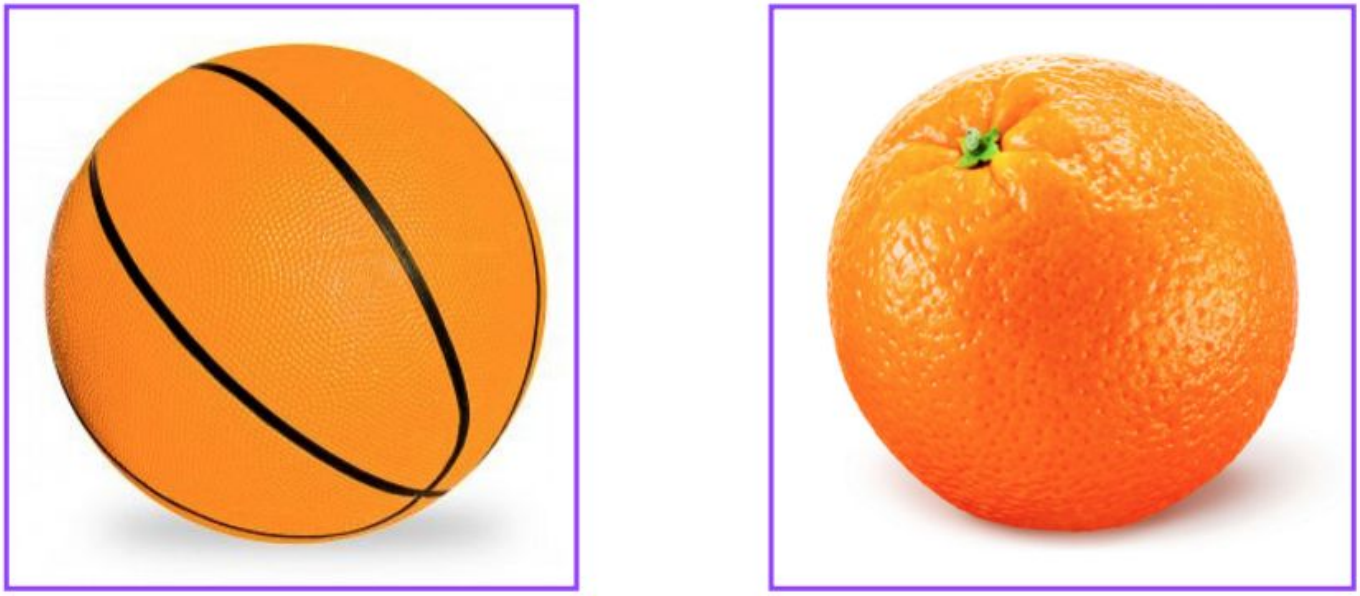
An input image going through some conv/pool layers and reaching a fully-connected layer. In between the feature maps and this fully-connected layer is a flattening step that creates a feature vector from the feature maps.

Final Feature Vector

So, how can we understand what's going on in this final feature vector? What kind of information has it distilled from an image?

To visualize what a vector represents about an image, we can compare it to other feature vectors, produced by the same CNN as it sees different input images. We can run a bunch of different images through a CNN and record the last feature vector for each image. This creates a feature space, where we can compare how similar these vectors are to one another.

We can measure vector-closeness by looking at the **nearest neighbors** in feature space. Nearest neighbors for an image is just an image that is near to it; that matches its pixels values as closely as possible. So, an image of an orange basketball will closely match other orange basketballs or even other orange, round shapes like an orange fruit, as seen below.



A basketball (left) and an orange (right) that are nearest neighbors in pixel space; these images have very similar colors and round shapes in the same x-y area.

Nearest neighbors in feature space

In feature space, the nearest neighbors for a given feature vector are the vectors that most closely match that one; we typically compare these with a metric like MSE or L1 distance. And *these* images may or may not have similar pixels, which the nearest-neighbor pixel images do; instead they have very similar content, which the feature vector has distilled.

In short, to visualize the last layer in a CNN, we ask: which feature vectors are closest to one another and which images do those correspond to?

And you can see an example of nearest neighbors in feature space, below; an image of a basketball that matches with other images of basketballs despite being a different color.

Nearest neighbors in feature space should represent the same kind of object.

Dimensionality reduction

Another method for visualizing this last layer in a CNN is to reduce the dimensionality of the final feature vector so that we can display it in 2D or 3D space.

For example, say we have a CNN that produces a 256-dimension vector (a list of 256 values). In this case, our task would be to reduce this 256-dimension vector into 2 dimensions that can then be plotted on an x-y axis. There are a few techniques that have been developed for compressing data like this.

Principal Component Analysis

One is PCA, principal component analysis, which takes a high dimensional vector and compresses it down to two dimensions. It does this by looking at the feature space and creating two variables (x, y) that are functions of these features; these two variables want to be as different as possible, which means that the produced x and y end up separating the original feature data distribution by as large a margin as possible.

t-SNE

Another really powerful method for visualization is called t-SNE (pronounced, tea-SNEE), which stands for t-distributed stochastic neighbor embeddings. It's a non-linear dimensionality reduction that, again, aims to separate data in a way that clusters similar data close together and separates differing data.

As an example, below is a t-SNE reduction done on the MNIST dataset, which is a dataset of thousands of 28x28 images, similar to FashionMNIST, where each image is one of 10 hand-written digits 0-9.

The 28x28 pixel space of each digit is compressed to 2 dimensions by t-SNE and you can see that this produces ten clusters, one for each type of digits in the dataset!

Other Feature Visualization Techniques

Feature visualization is an active area of research and before we move on, I'd like like to give you an overview of some of the techniques that you might see in research or try to implement on your own!

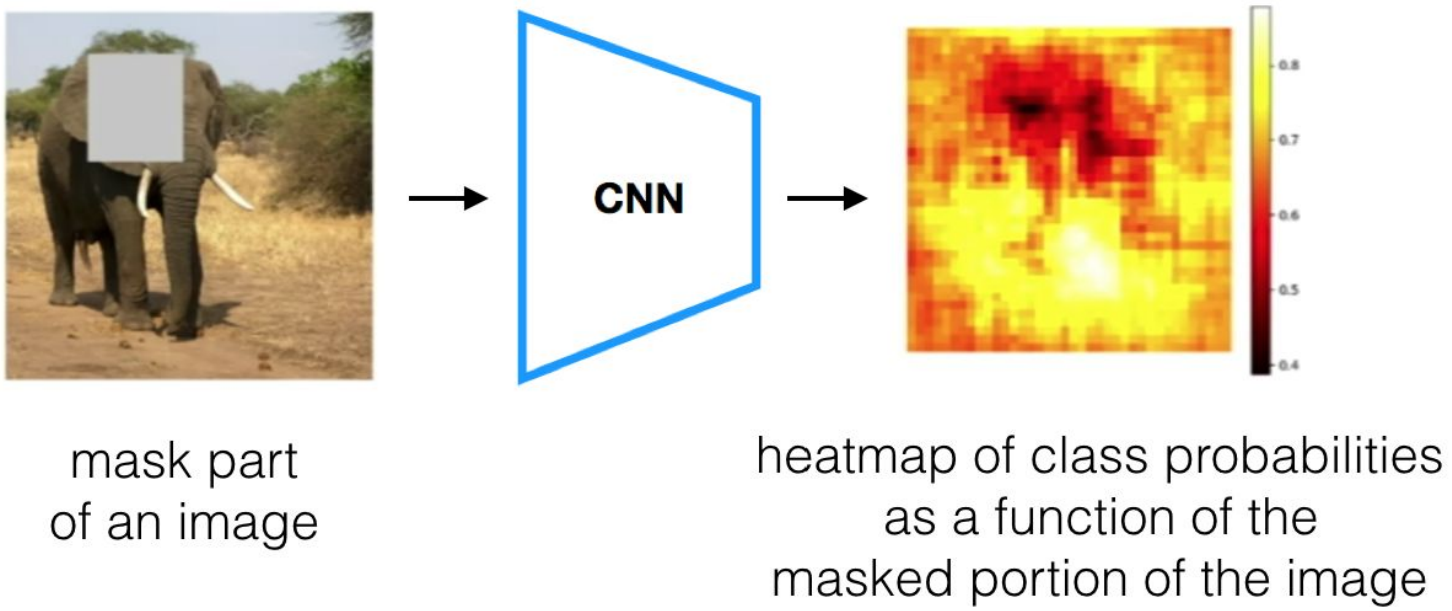
Occlusion Experiments

Occlusion means to block out or mask part of an image or object. For example, if you are looking at a person but their face is behind a book; this person's face is hidden (occluded). Occlusion can be used in feature visualization by blocking out selective parts of an image and seeing how a network responds.

The process for an occlusion experiment is as follows:

1. Mask part of an image before feeding it into a trained CNN,
2. Draw a heatmap of class scores for each masked image,
3. Slide the masked area to a different spot and repeat steps 1 and 2.

The result should be a heatmap that shows the predicted class of an image as a function of which part of an image was occluded. The reasoning is that **if the class score for a partially occluded image is different than the true class, then the occluded area was likely very important**!



Occlusion experiment with an image of an elephant.

Saliency Maps

Salience can be thought of as the importance of something, and for a given image, a saliency map asks: Which pixels are most important in classifying this image?

Not all pixels in an image are needed or relevant for classification. In the image of the elephant above, you don't need all the information in the image about the background and you may not even need all the detail about an elephant's skin texture; only the pixels that distinguish the elephant from any other animal are important.

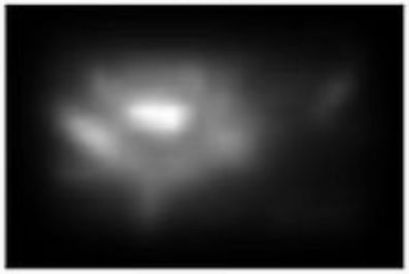
Saliency maps aim to show these important pictures by computing the gradient of the class score with respect to the image pixels. A gradient is a measure of change, and so, the gradient of the class score with respect to the image pixels is a measure of how much a class score for an image changes if a pixel changes a little bit.

Measuring change

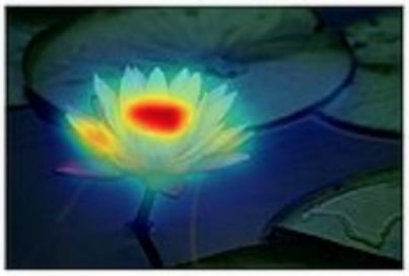
A saliency map tells us, for each pixel in an input image, if we change it's value slightly (by *dp*), how the class output will change. If the class scores change a lot, then the pixel that experienced a change, dp, is important in the classification task.

Looking at the saliency map below, you can see that it identifies the most important pixels in classifying an image of a flower. These kinds of maps have even been used to perform image segmentation (imagine the map overlay acting as an image mask)!

original image



saliency map



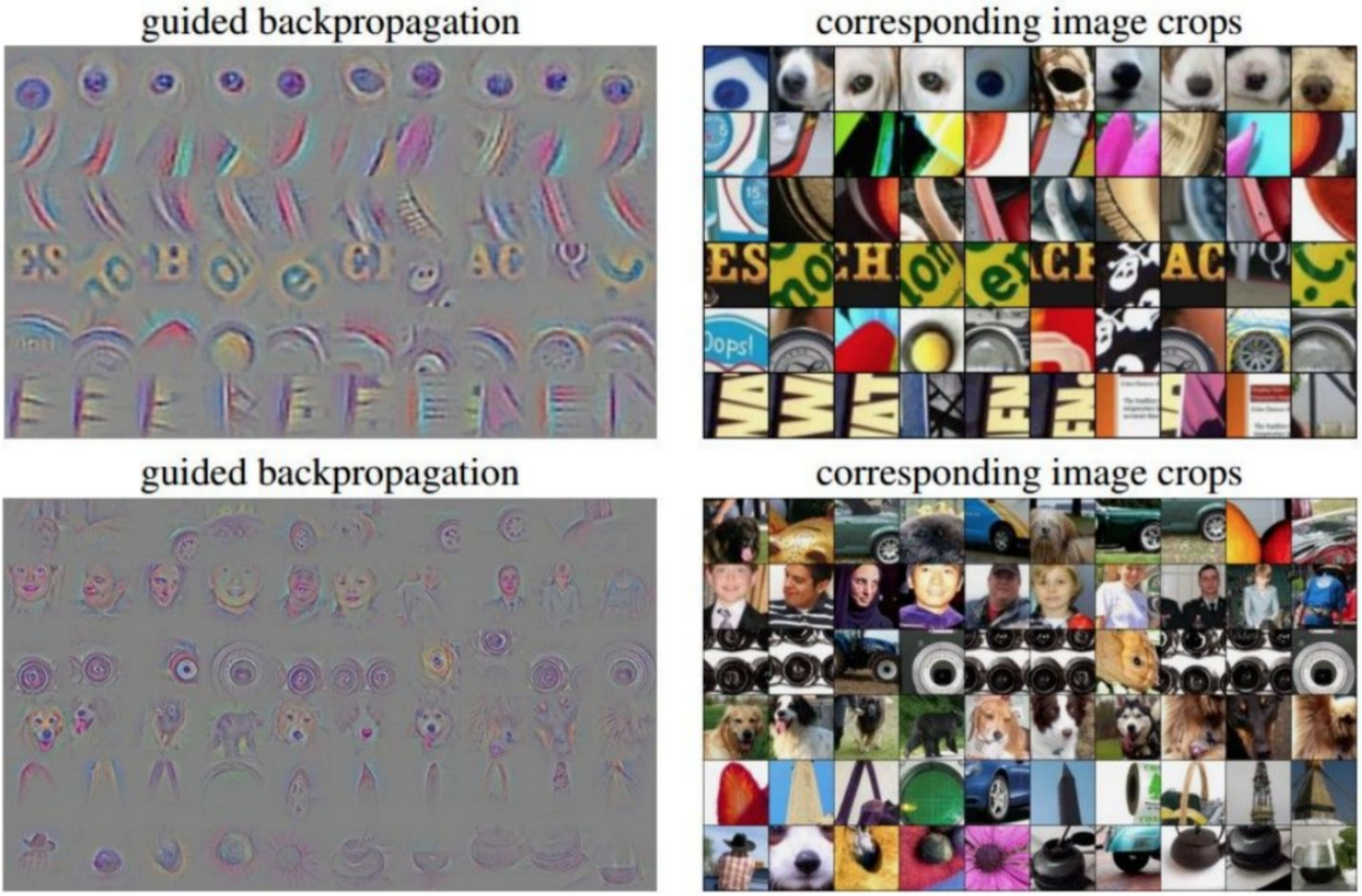
map overlay

Graph-based saliency map for a flower; the most salient (important) pixels have been identified as the flower-center and petals.

Guided Backpropagation

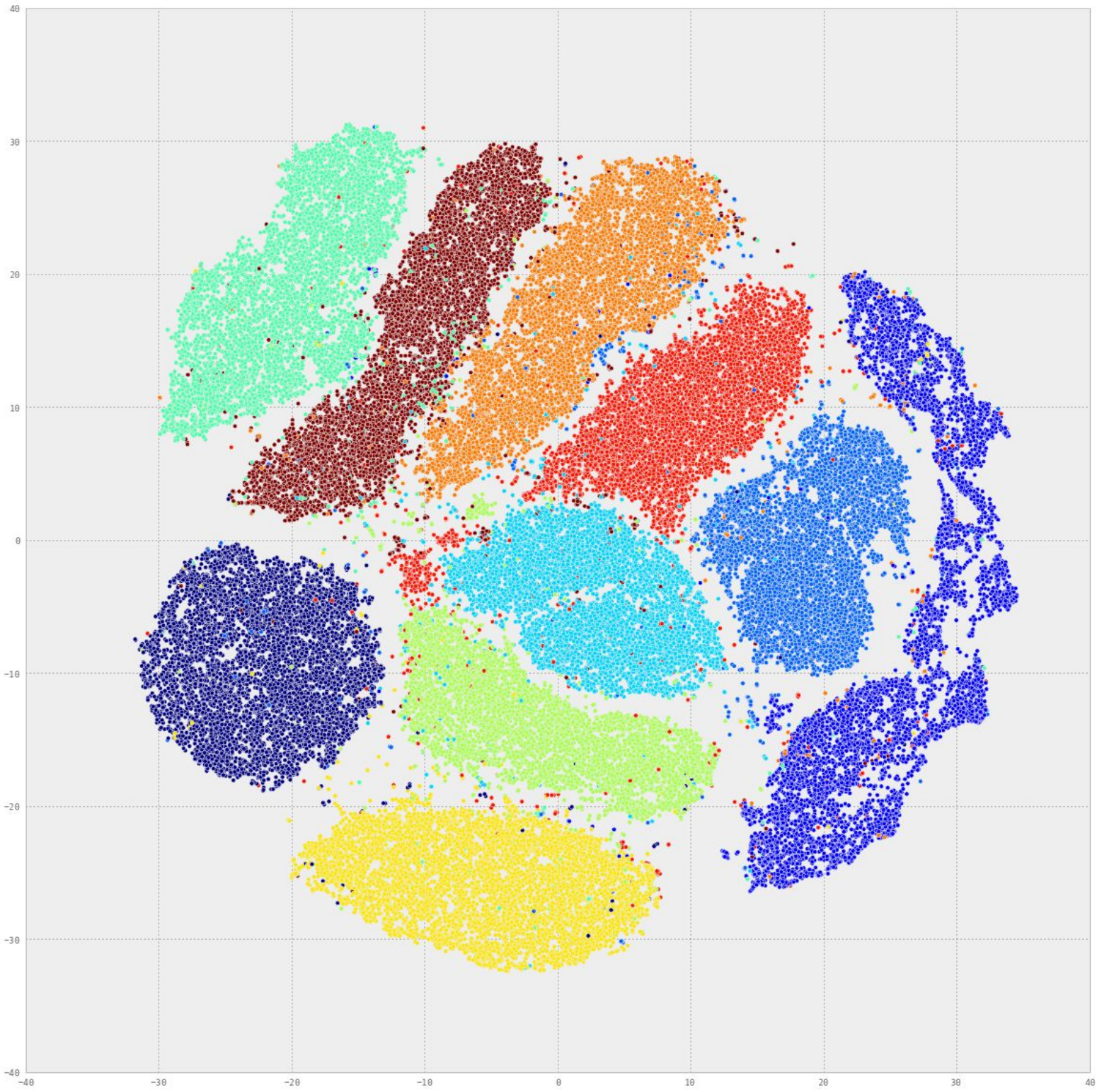
Similar to the process for constructing a saliency map, you can compute the gradients for mid level neurons in a network with respect to the input pixels. Guided backpropagation looks at each pixel in an input image, and asks: if we change it's pixel value slightly, how will the output of a particular neuron or layer in the network change. If the expected output change a lot, then the pixel that experienced a change, is important to that particular layer.

This is very similar to the backpropagation steps for measuring the error between an input and output and propagating it back through a network. Guided backpropagation tells us exactly which parts of the image patches, that we've looked at, activate a specific neuron/layer.



Examples of guided backpropagation, from

this paper.



t-SNE run on MNIST handwritten digit dataset. 10 clusters for 10 digits. You can see the generation code on Github.

t-SNE and practice with neural networks

If you are interested in learning more about neural networks, take a look at the **Elective Section: Text Sentiment Analysis**. Though this section is about text classification and not images or visual data, the instructor, Andrew Trask, goes through the creation of a neural network step-by-step, including setting training parameters and changing his model when he sees unexpected loss results.

He also provides an example of t-SNE visualization for the sentiment of different words, so you can actually see whether certain words are typically negative or positive, which is really interesting!

This elective section will be especially good practice for the upcoming section Advanced Computer Vision and Deep Learning, which covers RNN's for analyzing sequences of data (like sequences of text). So, if you don't want to visit this section now, you're encouraged to look at it later on.

Deep Dream

DeepDream takes in an input image and uses the features in a trained CNN to amplifying the existing, detected features in the input image! The process is as follows:

1. Choose an input image, and choose a convolutional layer in the network whose features you want to amplify (the first layer will amplify simple edges and later layers will amplify more complex features).
2. Compute the activation maps for the input image at your chosen layer.
3. Set the gradient of the chosen layer equal to the activations and and use this to compute the gradient image.
4. Update the input image and repeat!

In step 3, by setting the gradient in the layer equal to the activation, we're telling that layer to give more weight to the features in the activation map. So, if a layer detects corners, then the corners in an input image will be amplified, and you can see such corners in the upper-right sky of the mountain image, below. For any layer, changing the gradient to be equal to the activations in that layer will amplify the features in the given image that the layer is responding to the most.

DeepDream on an image of a mountain.

Style Transfer

Style transfer aims to separate the content of an image from its style. So, how does it do this?

Isolating content

When Convolutional Neural Networks are trained to recognize objects, further layers in the network extract features that distill information about the content of an image and discard any extraneous information. That is, as we go deeper into a CNN, the input image is transformed into feature maps that increasingly care about the content of the image rather than any detail about the texture and color of pixels (which is something close to style).

You may hear features, in later layers of a network, referred to as a "content representation" of an image.

Isolating style

To isolate the style of an input image, a feature space designed to capture texture information is used. This space essentially looks at the correlations between feature maps in each layer of a network; the correlations give us an

idea of texture and color information but leave out information about the arrangement of different objects in an image.

Combining style and content to create a new image

Style transfer takes in two images, and separates the content and style of each of those images. Then, to transfer the style of one image to another, it takes the content of the new image and applies the style of an another image (often a famous artwork).

The objects and shape arrangement of the new image is preserved, and the colors and textures (style) that make up the image are taken from another image. Below you can see an example of an image of a cat [content] being combined with the a Hokusai image of waves [style]. Effectively, style transfer renders the cat image in the style of the wave artwork.

If you'd like to try out Style Transfer on your own images, check out the **Elective Section: "Applications of Computer Vision and Deep Learning."**

Style transfer on an image of a cat and waves.