**Computer Vision**

You’ve been building up a solid foundation of programming and data analysis skills and, in this section, I’m excited to introduce you to one more self-driving car tool: **computer vision**.

Computer vision is how machines, like self-driving cars, visually perceive the world and respond to it. Imagine that you are driving a car; you have to use your senses to notice pedestrians, other cars, bicyclists, and all the road and traffic markings around you.

This is similar to how a self-driving car “sees” the world. It gathers data through cameras and other sensors, and then uses that input to safely navigate and move through the world.

A car parked on the side of a road

Description automatically generated

Color coded image of how a self-driving car "sees" lanes and surrounding vehicles.

In this lesson, we’ll learn about common computer vision techniques that are used to analyze camera images and extract important information from them like the color or shape of different objects.

We’ll also briefly talk about machine learning and how it’s used in combination with computer vision to give machines a way to learn from data and recognize patterns in images.

## Computer Vision in Industry

To help us learn about computer vision techniques and applications, we have with us an industry expert: Tarin Ziyaee, the co-founder and Chief Technology Officer of [Voyage](https://voyage.auto/), a self-driving car company.

At Voyage, computer vision is used in a myriad of applications, for example, the recognition of what state a traffic light is in, or, the detection of lanes, and so on.

One cool thing about computer vision though, is that the techniques that you'll learn about, need not only be used with camera images, but also images created with other sensors. The techniques that you'll learn, will be useful for any data that has **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!

So, in addition to cameras, self-driving cars use sensors like radar and LiDAR, which use sound waves and lasers to gather data about a car’s surroundings.

**Sensors**

Traditionally, RADAR is used for long-range detections, while cameras are used for rich sensory inputs.

When it comes to sensor configurations for SDCs, we have to note the details therein; LiDARs and RADARs are what are known as **active sensors**. That is, they sense the environment based on transmissions of energy.

Cameras on the other hand, are **passive sensors**; they can only sense the environment based on energy (in this case, photons) already in the scene.

These sensor details have serious repercussions with respect to the types of algorithms we end up using to analyze this data. Computer vision has many powerful tools, but part of good design had to do not only with knowing what to do, but also what not to do when we use sensor data from multiple sources.

Now, as we mentioned, computer vision is a very powerful tool, but it should not necessarily be associated with *only* camera images. It's possible to also construct LiDAR images with a LiDAR sensor, thereby giving you measured-depth alongside well as classified pixels. Next you'll see some LiDAR sensor output and resulting data!

**LiDAR Sensor Data**

LiDAR stands for Light Detection and Ranging, and it is a type of sensor that uses light (a laser) to measure the distance between itself and objects that reflect light. It does this by sending a series of laser pulses out and measuring the time it takes for an object to reflect that light back to the sensor; the longer the reflection takes, the farther an object is from the sensor. In this sense, LiDAR is **spatially coherent** data, and can be used to create a visual world representation.

Below, you can see the output of a LiDAR that Voyage uses, it sends out pulses of light and detects the car's surroundings.

A picture containing dark, night, standing

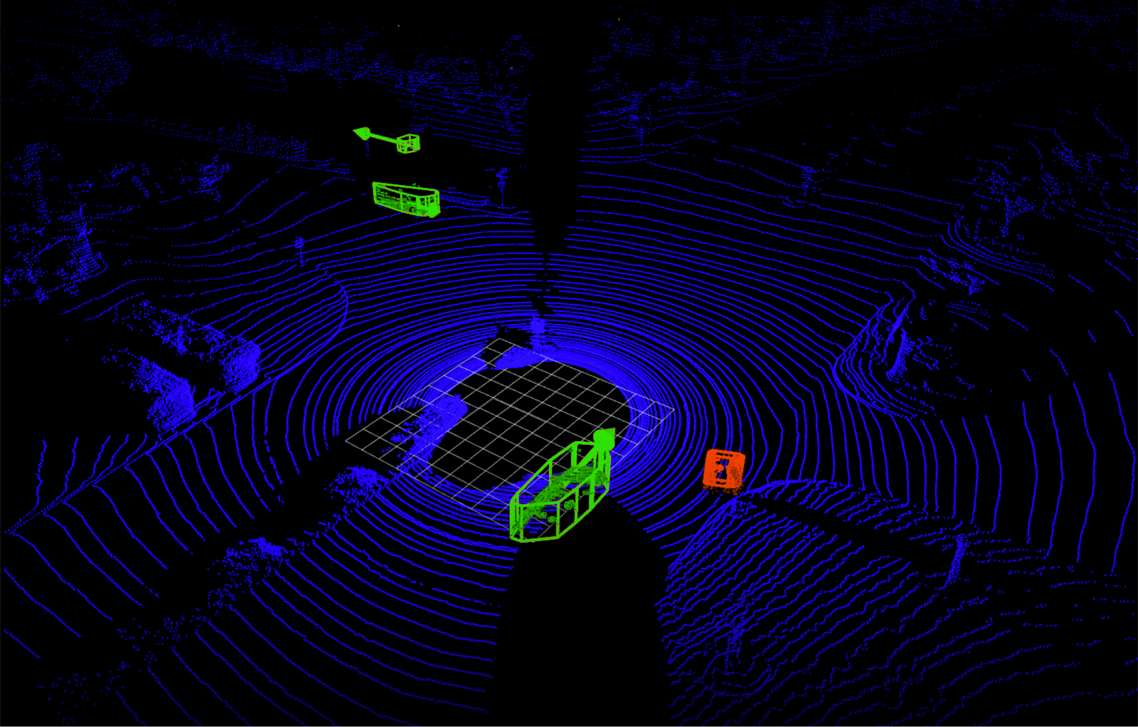
Description automatically generated

LiDAR sensor output. The blue indicates the detected, close-together points and the grid in the center is the space that the self-driving car occupies in the scene.

**Point Clouds**

Since LiDAR uses laser light, which sends out a thin light beam, the data it collects ends up being many single points also called **point clouds**. These point clouds can tell us a lot about an object like its shape and surface texture. By clustering points and analyzing them, this data provides enough information to classify an object or track a cluster of points over time!

Below, you can see the result of a classification algorithm performed on these points. The red are pedestrians and green indicates other cars.



That same LiDAR data, sent through a classifier that clusters the points together and classifies them as a human (red) or as a car wth some velocity (green).

**Visual World**

As you can see, LiDAR data provides enough spatial information to create a visual world representation, and in industry, computer vision techniques can be used to classify objects using not only camera images, but also point clouds and other types of spatially coherent data!

## Image Classification Pipeline

An image classifier is an algorithm that takes in an image as input and outputs a label or “class” that identifies that image. For example, a traffic sign classifier will look at different of roads and be able to identify whether that road contains humans, cars, bikes and so on. Distinguishing and classifying each image based on its contents.

There are many types of classifiers, used to recognize specific objects or even behaviors — like whether a person is walking or running — but they all involve a similar series of steps...

1. First, a computer receives visual **input** from an imaging device like a camera. This is typically captured as an image or a sequence of images.
2. Each image is then sent through some **pre-processing** steps whose purpose is to standardize each image. Common pre-processing steps include resizing an image, or rotating it, to change its shape or transforming the image from one color to another - like from color to grayscale. Only by standardizing each image, for example: making them the same size, can you then compare them and further analyze them in the same way.
3. Next, we **extract features**. Features are what help us define certain objects, and they are usually information about object shape or color. For example, some features that distinguish a car from a bicycle are that a car is usually a much larger shape and that it has 4 wheels instead of two. The shape and wheels would be distinguishing features for a car. And we’ll talk more about features later in this lesson.
4. And then, finally, these features are fed into a **classification model**! This step looks at any features from the previous step and predicts whether, say, this image is of a car or a pedestrian or a bike, and so on.

Image classification pipeline and specific pipeline example applied to classifying an image of a car. A screenshot of a cell phone

Description automatically generated

You’ll be programming each of these classification steps manually so that you really understand each step.

And by the end of this lesson, you’ll have all the skills you’ll need to complete the final project: building a traffic light classifier, which takes in images of traffic lights and separates them into three classes: red, yellow, or green lights.

## Image Classification Pipeline

A standard classification pipeline uses the following steps to classify a given image.

A screenshot of a cell phone

Description automatically generated

## Classification Techniques

You’ve just seen a complete classification pipeline. For an image classifier, starting with some input images, computer vision techniques are used to process those images, and extract features, like distinguishing colors or shapes in that image.

Then, a classifier looks at these features and outputs a class, which is a label that describes the image.

**A classifier should predict that images with similar shapes or colors have the same class.**

We usually tell a classification model what to look for. For example, say we are looking at a bunch of images and we want to classify them into two classes: car and not-car! To classify a car, we might write a program that looks for the different parts of a car: wheels, lights, windows, and so on, and then if those things are found, we’ll classify an image as a car. We decide what traits are important to look for.

## Machine Learning

However, there's another way to create a classifier and that’s with machine learning.

Machine learning allows a computer to figure out things on its own by giving it lots of examples. So, instead of telling a model what traits to look for, with machine learning, we’d just give it lots of images of cars and not-cars and let it learn to recognize traits that differentiate them! It can learn to recognize wheels and windows and which classification algorithm is best for accurately classifying any given image as car or not-car!

Now, you might be wondering: how exactly does a model like this learn to classify different images?

Next, we’ll look at how machine learning techniques can actually be trained to classify sets of images.

## Separating Data

When we talk about machine learning, you’ll often hear the terms “deep learning” and “neural networks” come up. This might conjure up an image of a brain or some strange graphic of layers of mathematical formulas and data. But at their core, all of these learning techniques are about separating data into different classes.

The image that this conjures in my head is of a child playing on a beach. The child sees some blue and yellow shells in the sand.

Then, someone says to the child: “put these shells into groups and draw a line between them.”

Without me telling you anything else, how would you group these shells?

A close up of graphics

Description automatically generated

Image of yellow and blue shells

You might separate them by color and shape and draw a line between these two types of shells. Now, for a computer, the scenario might be that we give it individual images of shells, and just like a child, a neural network can learn how to separate these images of shells based on similarities or differences in the given examples.

A close up of a flower

Description automatically generated

Separated shells

After this separation step, if a network sees a new image, one it hasn’t seen before, it sorts and classifies it based on which side of the line it falls!

Realistically, data is often a lot more complex than this, but neural networks just layer separation on top of separation layer to create more complex boundaries and group all kinds of data!

## Training a Model

Earlier, I gave a simple example of a classification model that classifies all images as either **car** or **not-car**. And I said that using machine learning techniques, you can give a model lots of examples of cars and not-cars until it learns to recognize them. But, how exactly does this work?

Similar to how humans learn, a model has to learn by its successes and mistakes, and we often call this the training phase. At the beginning of a training phase, a classification model typically performs very badly.

The model will look at an image, try to classify it as a car or not-car and monitor the errors that it makes. For example, if a model mistakenly classifies a car as not-car, it will learn from this mistake, tweak its classification parameters and improve its performance each time it sees more images.

After many iterations, the model converges on the right set of parameters and the error rate becomes low. That’s when we consider the model to be trained.

A training flow is pictured below. This is a [convolutional neural network](http://cs231n.github.io/convolutional-networks/) that learns to recognize and distinguish between images of cars and not-cars.

A screenshot of a cell phone

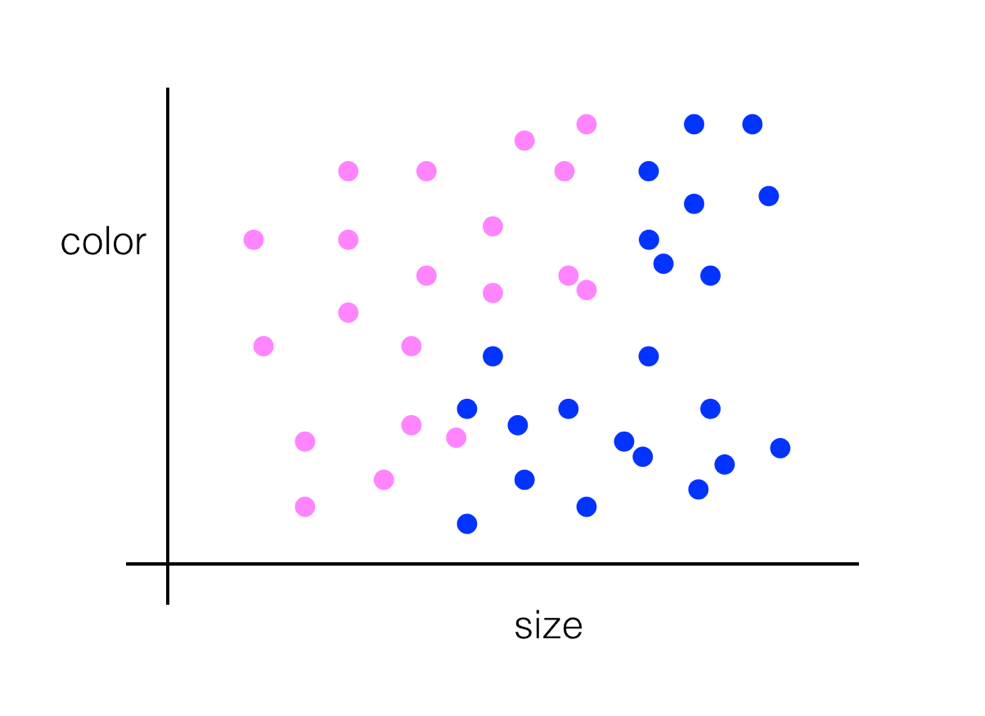
Description automatically generated

A convolutional neural network will adjust its pattern recognition algorithm until it learns to accurately classify a set of images.

Now, this is a very high-level view of how to train any classification model. And the details will vary based on the type of model you use and the training algorithm you choose.

## 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 images of cars and of bikes.

A close up of a map

Description automatically generated

## Layers of Separation

What if the data looked like this?

A close up of a logo

Description automatically generated

Pink and blue dots representing images of cars and of bikes.

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.

A picture containing table, clock, group, computer

Description automatically generated

A picture containing photo, sitting, different, full

Description automatically generated

Separated data.

## 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.

A close up of a street

Description automatically generated

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!

**Pre-processing**

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.

Really common pre-processing steps include:

1. Changing how an image looks spatially, by using geometric transforms which can scale an image, rotate it, or even change how far away an object appears, and
2. Changing color schemes, like choosing to use grayscale images over color images.

## Color Masking

We know about how color images are represented numerically, and next, I want to show you how color can be used in image analysis and transformation. We’ll start by learning how to use information about the colors in an image to isolate a particular area in an image. This is easiest to understand in an example.

We'll be selecting an area of interest using a color threshold. Color thresholds are used in a number of applications including extensively in computer graphics and video; a common use is with a **green screen**.

A green screen is used to layer two images or video streams based on identifying and replacing a large green area.

A picture containing drawing, car

Description automatically generated

Car on a green screen background.

So, how does all this work?

Well, the first step is to isolate the green background, and then replace that green area with an image of your choosing. Next, let's see how to do this programmatically with the image shown above!

## OpenCV

Before we jump into coding our green screen example, you may want to follow along by downloading a library, which we’ll use throughout this course: OpenCV!

**Installing OpenCV is completely optional for this course, since you can complete your work in provided notebooks.**

OpenCV is a **computer vision and machine learning software library** that includes many common image analysis algorithms that will help us build custom, intelligent computer vision applications. To start with, this includes tools that help us process images and select areas of interest! The library is widely used in academic and industrial applications; from [**their site**](http://opencv.org/about.html), OpenCV includes an impressive list of users: “Along with well-established companies like Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda, Toyota that employ the library, there are many startups such as Applied Minds, VideoSurf, and Zeitera, that make extensive use of OpenCV.”

And we’ll be getting a lot of experience with this library as we progress!

### Installation via Terminal

I will be using a Jupyter notebook to walk through code examples and I’ll assume you’ll be working in the same environment.

To install OpenCV for use in a notebook, open a terminal window (aka a command prompt window for Windows users) and use conda to install the latest version (v3) using the following command:

conda install opencv3

You should then get an installation prompt asking to proceed (y/n)? Select yes to proceed with the installation.

Troubleshooting Note: If the above installation command did not work for your computer, try accessing opencv3 through a specific source and use one of the the following commands:

conda install -c conda-forge opencv=3.2.0

or

conda install -c menpo opencv3=3.2.0

A picture containing indoor, bird

Description automatically generated

Screenshot of installation command in my (Cezanne's) terminal window.

### Verify Installation

And that’s it! To verify that you have OpenCV correctly installed, let’s check in a python environment. Again in your terminal window, type the following:

First, let’s get into our python execution environment by typing python

Then, let’s see if we have OpenCV installed by typing import cv2. if this command executes without any errors, you’ve installed it correctly!

Finally, to check your version of OpenCV, type a print statement print (cv2.\_\_version\_\_) which should print out the latest stable version.

**Now you're ready to code in your own environment!**

Let's get back to the green screen example :)

The code should read cv2.inRange() and the complete notebook is in the next section.

## Green Screen

To create a color mask, I’ll create a color threshold, I’ll define lower and upper bounds for the color I want to isolate: green.

*# Define our color selection boundaries in RGB values*

lower\_green = np.array([0,180,0])

upper\_green = np.array([100,255,100])

Next, I’ll use the color thresholds I just created to create an **image mask**.

*# Define the masked area*

mask = cv2.inRange(image, lower\_green, upper\_green)

Then, to mask the image, I’ll make an image copy called masked\_image, of our original image to manipulate without changing the original image. To select the green screen area, is by asking for the part of the image that overlaps with the part of the mask that is white or not black (!= 0).

*# Mask the image to let the car show through*

masked\_image = np.copy(image)

masked\_image[mask != 0] = [0, 0, 0]

Then, when we display our masked\_image, we can see that the car area is the only area that shows through; the green screen background is gone!

A close up of a car

Description automatically generated

Masked background!

Next, it’ll be up to you to practice masking and add a background! You should get an image that looks similar to the flying car below.

A fighter jet flying through a cloudy blue sky

Description automatically generated

Green screen replaced with an image of the sky.

## Color Spaces

A **color space** is a specific organization of colors; color spaces provide a way to categorize colors and represent them in digital images.

**RGB** is red-green-blue color space. You can think of this as a 3D space, in this case a cube, where any color can be represented by a 3D coordinate of R, G, and B values. For example, white has the coordinate (255, 255, 255), which has the maximum value for red, green, and blue.

A picture containing toy

Description automatically generated

RGB color space

There are many other ways to represent the colors in an image besides just composed of red, green, and blue values.

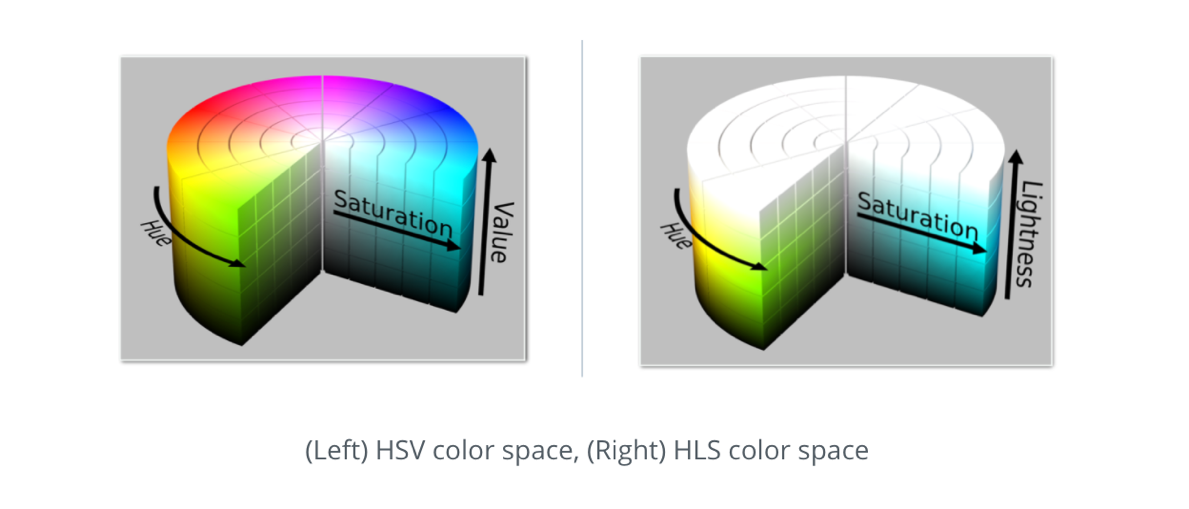
There is also **HSV** color space (hue, saturation, and value), and **HLS** space (hue, lightness, and saturation). These are some of the most commonly used color spaces in image analysis.

To get some intuition about these color spaces, you can generally think of **Hue** as the value that represents color independent of any change in brightness. So if you imagine a basic red paint color, then add some white to it or some black to make that color lighter or darker -- the underlying color remains the same and the hue for all of these colors will be the same.

## HSV and HLS

On the other hand, **Lightness** and **Value** represent different ways to measure the relative lightness or darkness of a color. For example, a dark red will have a similar hue but much lower value for lightness than a light red. **Saturation** also plays a part in this; saturation is a measurement of colorfulness. So, as colors get lighter and closer to white, they have a lower saturation value, whereas colors that are the most intense, like a bright primary color (imagine a bright red, blue, or yellow), have a high saturation value. You can get a better idea of these values by looking at the 3D color spaces pictured below.

Most of these different color spaces were either inspired by the human vision system, and/or developed for efficient use in television screen displays and computer graphics. You can read more about the history and the derivation of HLS and HSV color spaces [**here**](https://en.wikipedia.org/wiki/HSL_and_HSV).



## HSV Conversion

In the code example, I used HSV space to help detect a green screen background under different lighting conditions. OpenCV provides a function hsv = cv2.cvtColor(im, cv2.COLOR\_RGB2HSV) that converts images from one color space to another.

After this conversion, I plotted the individual color channels; it was easy to see that the Hue channel remained fairly constant under different lighting conditions.

A close up of a computer

Description automatically generated

H, S, and V channels for the green screen car image.

Next, I used the same masking code as before only using a **Hue channel** color mask, next you'll see how this looks in a notebook!

## Day and Night Classification

You’re on your way to being able to build classifier! You know how to analyze color and brightness in a given image, and that skill alone can help you distinguish between different traffic signs.

Now, I’m going to give you a classification challenge. What if I asked you to classify two types of images, taken during the day or at night when the sun has set, and I want you to separate these images into two classes: day or night.

A picture containing building, outdoor, cat, photo

Description automatically generated

Examples of day and night images of the same scene.

This is actually an important classification challenge for self-driving cars. These cars need to know what kind of conditions they are driving in, so that they can safely navigate a road at any time of day, and so that they can still recognize other vehicles and surrounding objects whether it is dark or light outside!

We’ll walk through each classification step together, but what do you think would be the first step in creating a classification model for day and night images?

Before you can classify any set of images, you have to look at them! **Visualizing the image data you’re working with is the first step in identifying any patterns in image data and being able to make predictions about the data!**

So, we’ll first load in this image data and learn a bit about the images we’ll be working with.

### 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 screenshot of a cell phone

Description automatically generated

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?

In order of [cat value, tiger value, hippopotamus value, dog value], that label indicates that it's an image of a tiger! Let's do one more example, what about the label [0, 0, 0, 1]?

**Average Brightness**

Here were the steps we took to extract the average brightness of an image.

1. Convert the image to HSV color space (the Value channel is an approximation for brightness)
2. Sum up all the values of the pixels in the Value channel
3. Divide that brightness sum by the area of the image, which is just the width times the height.

This gave us one value: the average brightness or the average Value of that image.

In the next notebook, make sure to look at a variety of day and night images and see if you can think of an average brightness value that will separate the images into their respective classes!

The next step will be to feed this data into a classifier. A classifier might be as simple as a conditional statement that checks if the average brightness is above some threshold, then this image is labeled as 1 (day) and if not, it’s labeled as 0 (night).

On your own, you can choose to create more features that help distinguish these images from one another, and we’ll soon learn about testing the accuracy of a model like this.

## What is a Feature?

A feature can easily be thought of as a "summarizer" of something. So features of images, are really just nice and concise summarizers of image data. Furthermore, just as how images are really just a collection of numbers in an array, features are also just another collection of numbers in an array, although usually, they are much smaller than images.

So what does this mean? Let’s be even simpler: Forget images...consider humans! An individual person has many facets about themselves and it may be hard to describe someone in totality.

What are, however, some "compact" features that we may extract about a person? Compact, because we want these features to describe something about this person, but we want this description to be a summary of what is relevant.

For example, if we wanted to place boxers into their weight class, we may want to do feature extraction on each fighter, and we would extract a two dimensional feature: height and weight (both of which are used to determine a weight class).

Those are "features" in this sense, because they nicely **ignore the irrelevant**; they describe a person’s weight and height, features that are useful for placing the boxer into their proper weight class, and they also ignore things like skin color, or say, hair length, etc. So, in this sense, you can think of **feature extraction** as a way to extract relevant information, while also smartly ignoring the irrelevant. A good feature is very succinct.

Features are distinct and measurable pieces of information in an image. And we’ll go through examples of features and how to detect them. One of the breakthroughs in computer vision came from being able to automatically come up with features that are good. However you can also do this on your own, manually.

**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. For example, say I wanted to classify a stop 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 avoid crashing into a car (a very important avoidance case!). I'll want to classify the object as a car, or at least recognize the car's boundaries, which are determined by shape. Specifically, I'll want to identify the edges of the vehicle, so that I can track the car and avoid it. Color is not very useful in this case, but shape is critical.

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

## 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.

## High-pass Filters

High-pass filters detect big changes in intensity over a small area, and patterns of intensity can be best seen in a grayscale image.

A car stopped at a traffic light sitting on the side of a road

Description automatically generated

Grayscale car.

The filters I’ll be talking about are in the form of matrices, often called **convolution kernels**, which are just grids of numbers that modify an image. Here is a [**resource**](http://setosa.io/ev/image-kernels/) if you'd like to see a wider variety of kernel types in action. Below is an example of a high-pass kernel that does edge detection. It’s a 3x3 kernel, whose elements all sum to zero.

It’s important that, for edge detection, all of the elements sum to 0 because edge filters compute the difference or change between neighboring pixels; they are an approximation for the **derivative** of an image over space.

A picture containing clock

Description automatically generated

High-pass kernel.

### Convolution

During kernel convolution, the 3x3 kernel is slid over every pixel in the original, grayscale image. The weights in the kernel are multiplied pair-wise around a center pixel, and then added up. This sum becomes the value of a pixel in a new, filtered, output image.

This operation is at the center of convolutional neural networks, which use multiple kernels to extract shape-based features and identify patterns that can accurately classify sets of images. These neural networks are trained on large sets of labelled data, and they learn the most effective kernel weights; the weights that help characterize each image correctly.

A screenshot of a cell phone

Description automatically generated

Calculating one output pixel value (175) while performing convolution.

To handle the edges of images, where the filter cannot exactly overlap, a variety of techniques are used. One of the most common is to extend the edge pixel values of the image out by one and use that to perform a convolution. Another is to pad the image with zeroes, though this creates a darker border in the resulting, filtered image.