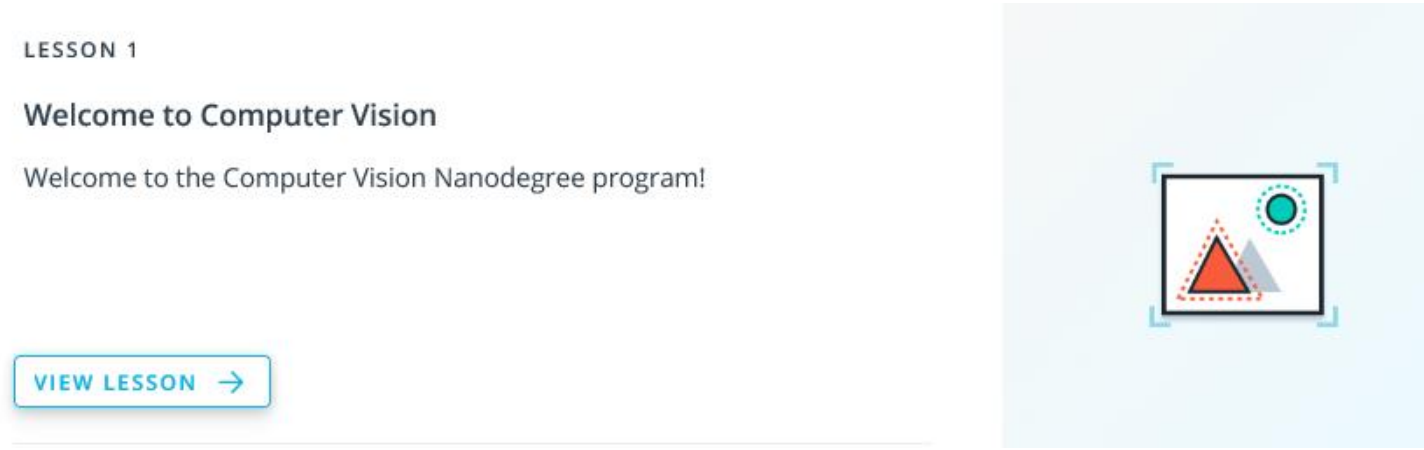
Welcome to the Computer Vision Nanodegree Program



https://www.youtube.com/watch?v=9s-gm2ZvODI&feature=emb_logo

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.

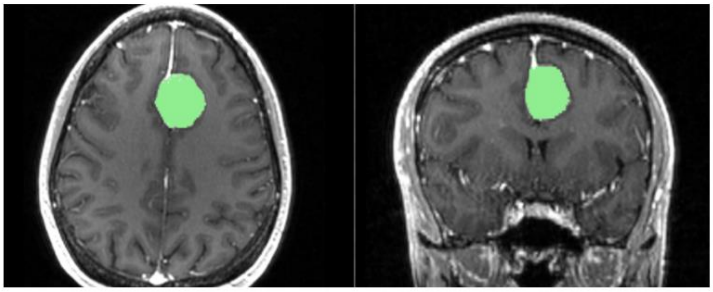
Self-Driving Car

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

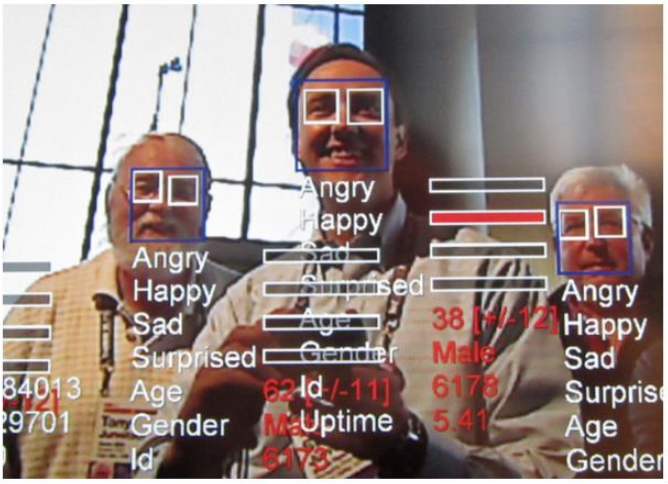
An AI system, using computer vision, can learn to recognize images of cancerous tissue and help with early detection and diagnoses.



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.

https://www.youtube.com/watch?v=_8be3GdqfqU&feature=emb_logo

Course Outline

https://www.youtube.com/watch?time_continue=7&v=INMNyJGB2DI&feature=emb_logo

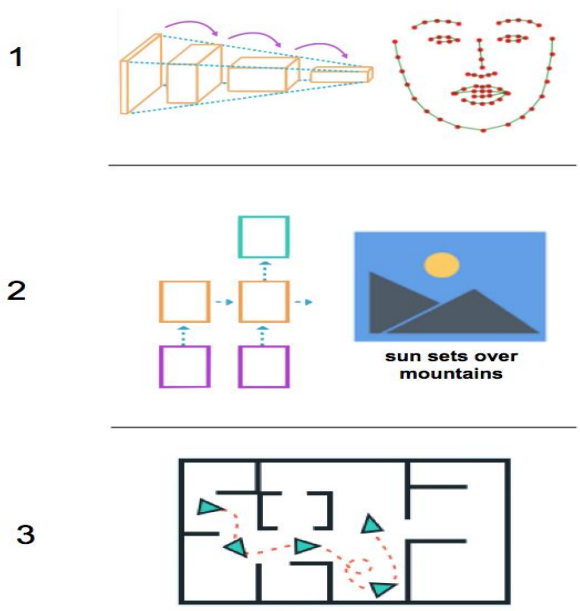
Computer Vision Course Overview

This Nanodegree program is broken into three main sections:

1. **Intro to Computer Vision**, which covers topics like image processing, feature extraction done manually or through training a convolutional neural network (CNN) using PyTorch.
2. **Advanced Computer Vision and Deep Learning**, which is all about advances in deep learning architectures like region-based CNN's, YOLO and single-shot detection algorithms, and CNN's used in combination with recurrent neural networks.
3. **Object Tracking and Localization**, which covers how a robot can move and sense the world around it, creating a visual representation of the world as it navigates.

Each of these three sections will have an associated project that allows you to demonstrate the skills you've learned in each part.

Projects



From top to bottom, representations of: 1. Facial Keypoint Detection, 2. Automatic Image Captioning, and 3. Localization and Mapping

You'll learn computer vision and deep learning techniques by getting to apply your skills to a variety of projects. The three project in this program are as follows:

1. Facial Keypoint Detection
2. Automatic Image Captioning
3. Simultaneous Localization and Mapping (SLAM).

You'll use a combination of computer vision and deep learning techniques to complete these projects; submitting each for review. By the end of the course, you'll have an impressive portfolio of applications.

Elective/Extracurricular Sections

At the bottom of the classroom navigation bar, you can see an Elective section with various lessons. These lessons are meant to **be review or supplement your learning** in the classroom, but the skills covered in this section will *not* be required for you to complete the projects in this course. For example, we might put content from Udacity's deep learning program in here as a way to review deep learning concepts. So, it is up to you to decide whether to watch this material or not.

As you go through this course, you will be prompted in text to look at specific elective sections. For example, as you start the "CNN and Feature Visualization" lesson, you will be prompted to consider watching the elective section "Review: Training a Neural Network."

Expertise

Throughout this course, you'll be learning from academic and industry experts. We've partnered with companies like Affectiva and NVIDIA to bring you information about the latest techniques and advances in data collection, computer vision, and deep learning architectures.

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

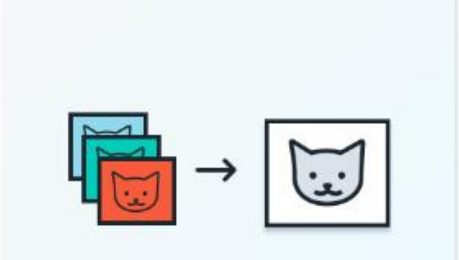
Register for the NVIDIA Developer Program to access the latest NVIDIA SDK tools and be the first to hear about NVIDIA product announcements. Learn more at developer.nvidia.com/developer-program .

https://www.youtube.com/watch?v=wm1aZZvF6Is&feature=emb_logo

Lesson 2: Image Representation and Classification

Image Representation & Classification

Learn how images are represented numerically and implement image processing techniques, such as color masking and binary classification.



Exercise Repository

Note that most exercise notebooks can be run locally on your computer, by following the directions in the **Github Exercise Repository**.

Learning Journey and Pace

This Image Representation Lesson is meant for people who have just started learning about image analysis. We want this program to be accessible to people who are just starting to learn about computer vision *and* to people who are interested in more advanced deep learning topics and applications like image classification and object tracking.

- So, **if you find this lesson to be a bit too easy, you are welcome to skip forward to the next lesson: Convolutional Filters and Edge Detection**.
- On the other hand, if you are just learning about computer vision or even if you want to review some foundational concepts, please proceed!

Happy learning!

Next

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.

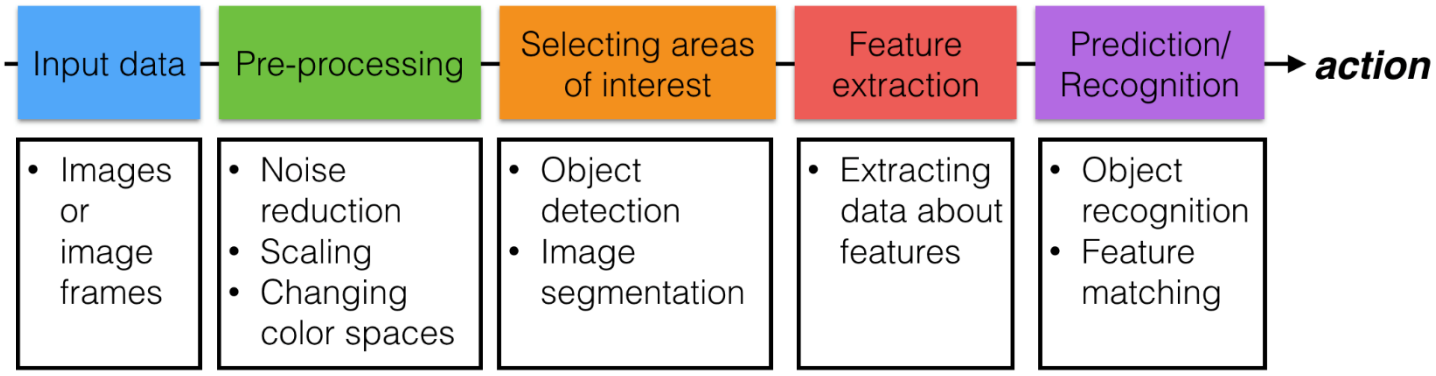
Emotional intelligence is the ability to understand and influence human emotion. For example, observing that someone looks sad based on their facial expression, body language, and what you know about them - then acting to comfort them or asking them if they want to talk, etc. For humans, this kind of 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**.

https://www.youtube.com/watch?v=D_LzJsJH5qk&feature=emb_logo

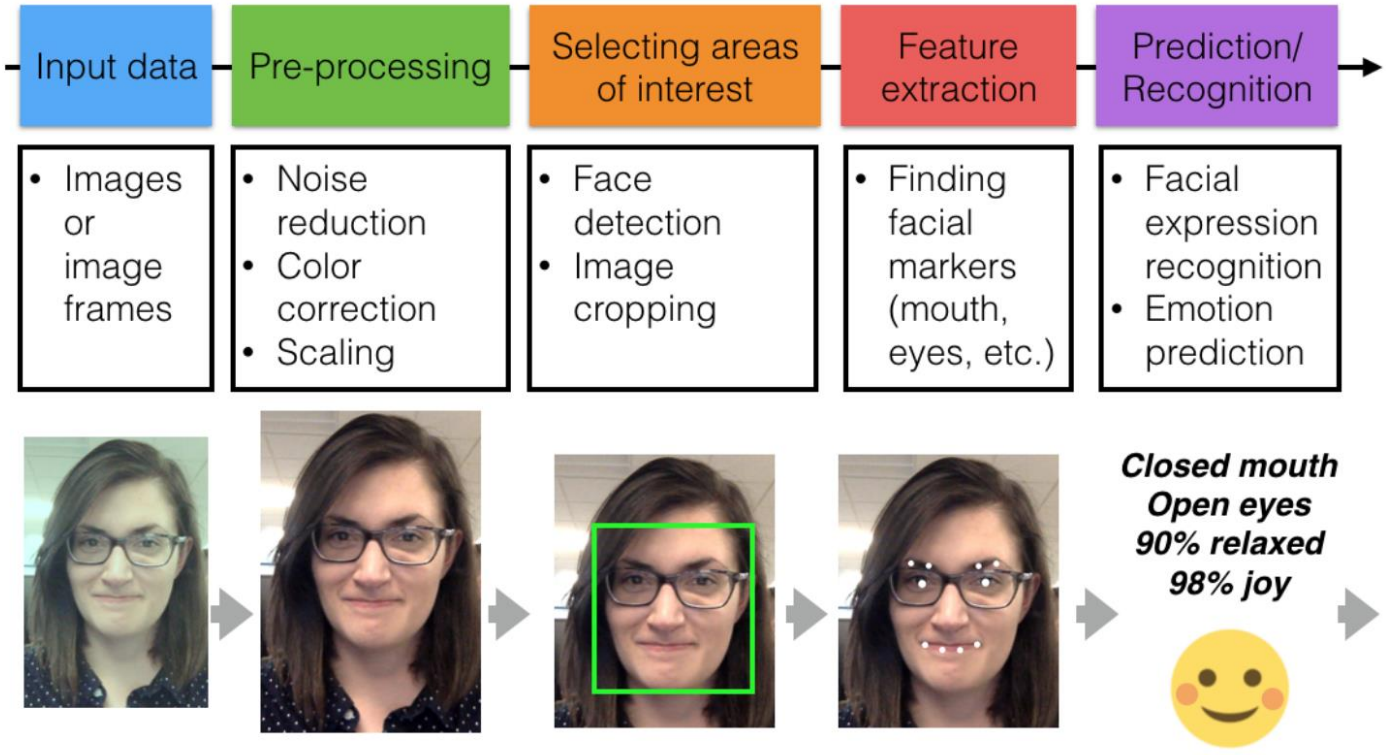
Computer Vision Pipeline

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.

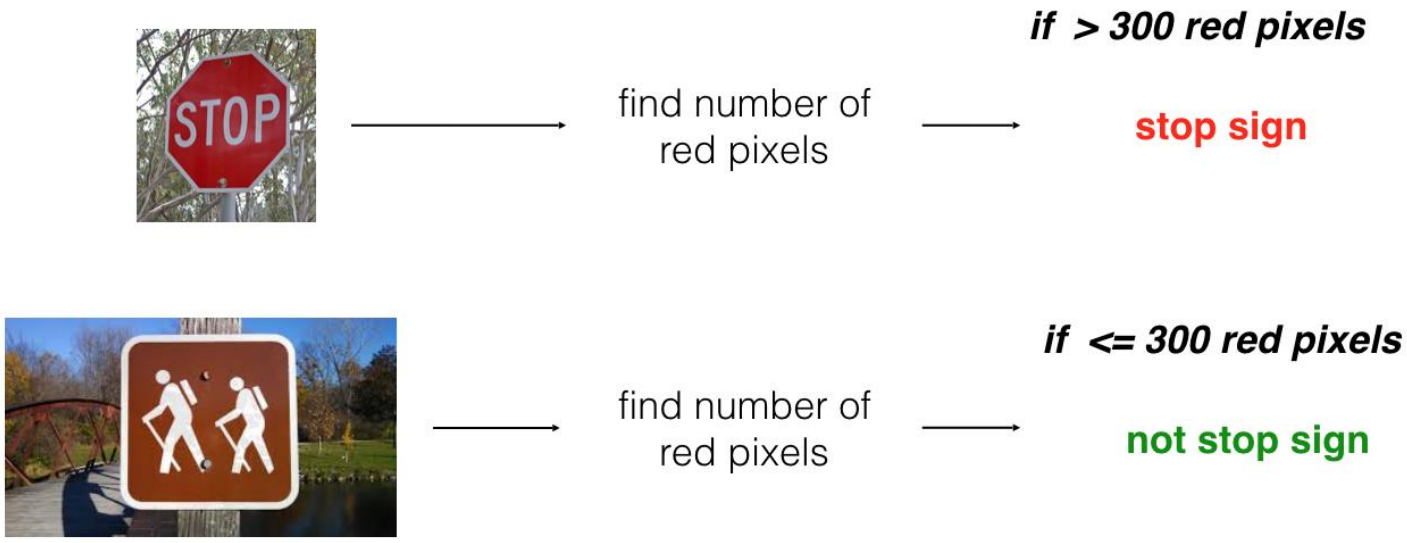


Facial recognition pipeline.

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.

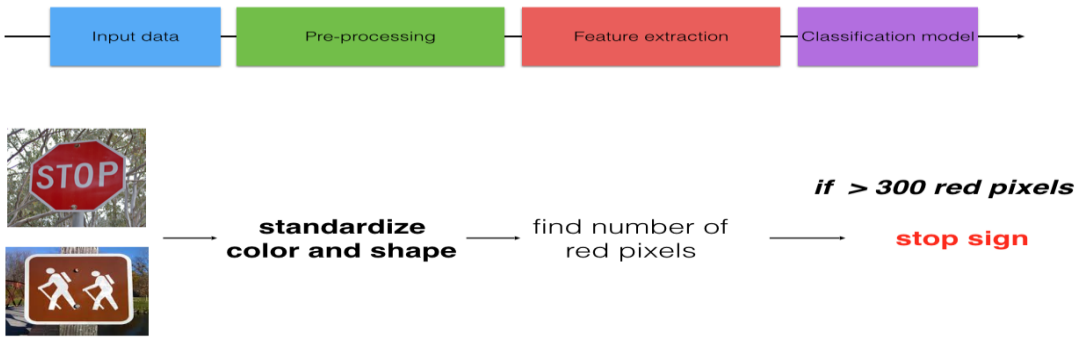
For example, imagine that you've created a simple algorithm to distinguish between stop signs and other traffic lights.



[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.

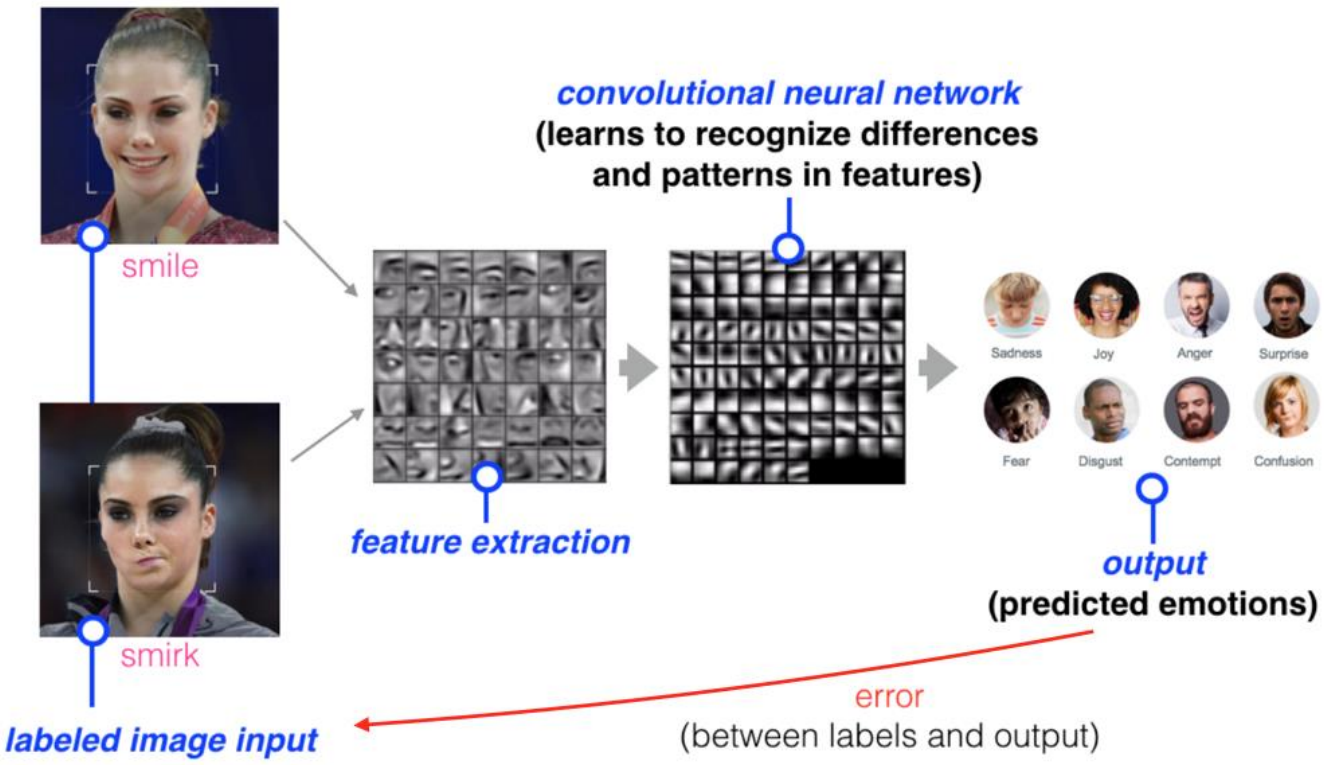


Training a Neural Network

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!

https://www.youtube.com/watch?v=m4GVfwVkj74&feature=emb_logo

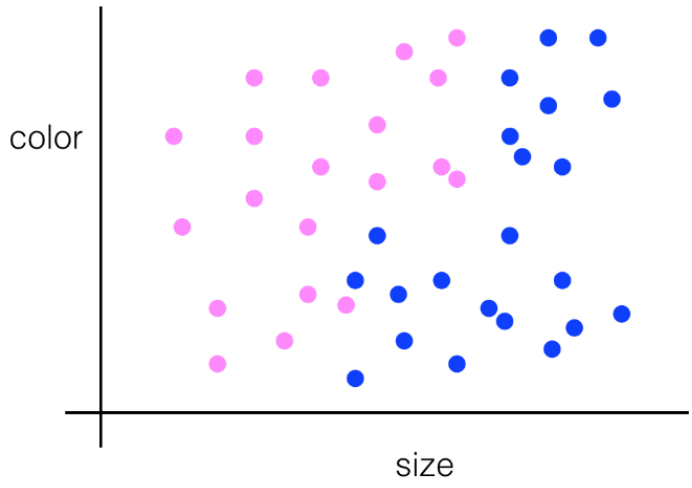
Machine Learning and Neural Networks

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.

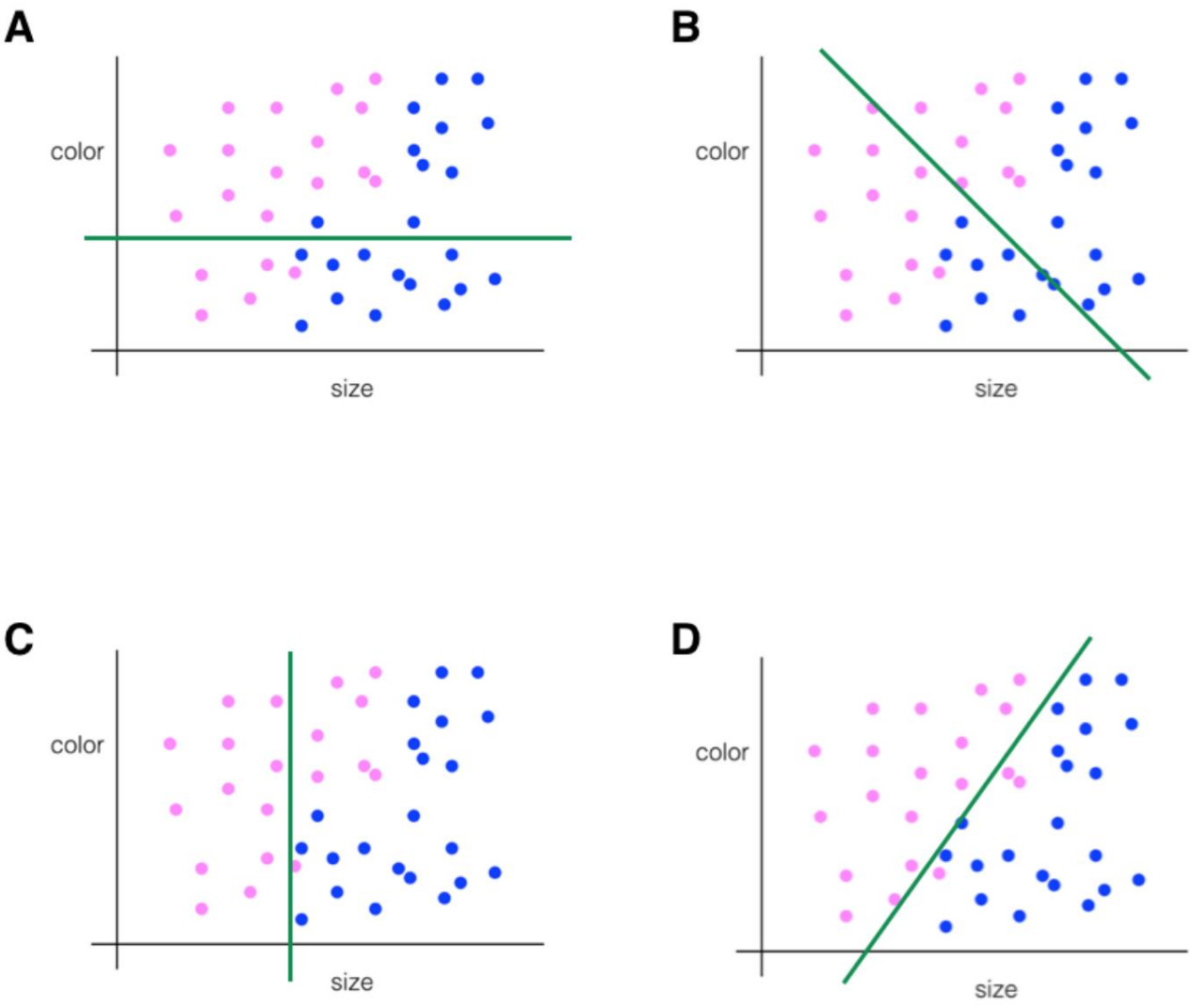
The example we gave earlier was sorting images of facial expressions into two categories: smile or smirk. A neural network might be able to learn to separate these expressions based on their different traits; a neural network can effectively learn how to draw a line that **separates** two kinds of data 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.



Quiz Question

Given the above choices, which line would you choose to best separate this data?

- A (horizontal line)
- B (diagonal line from top-left to bottom-right)

-

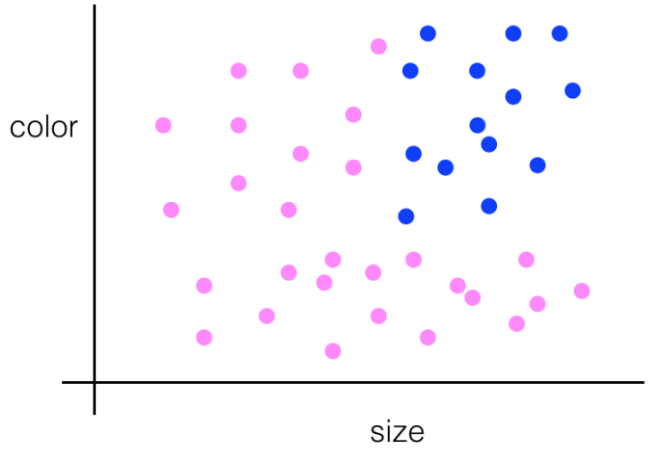
C (vertical line)

- D (diagonal line from top-right to bottom-left)

Submit

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.

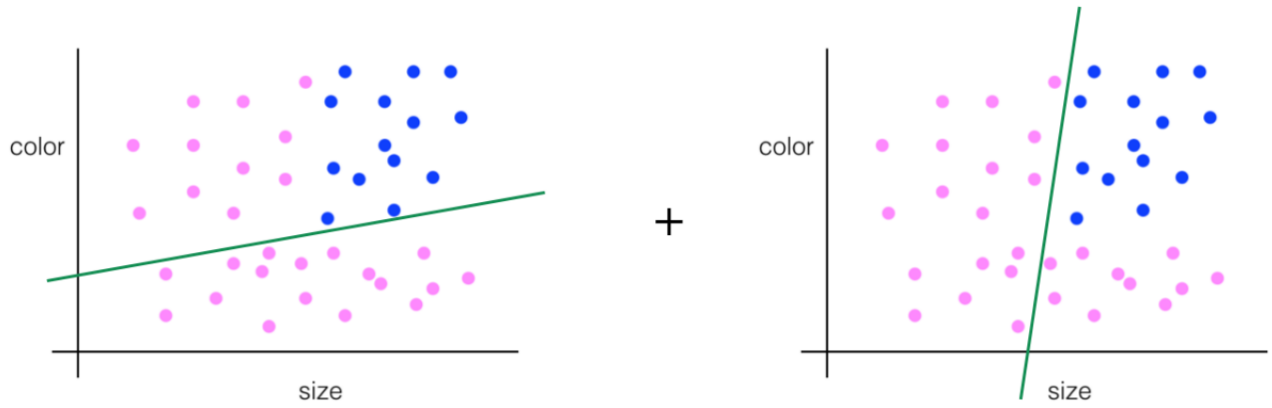
AffdexMe is currently available for download on multiple platforms.

If you'd like to try this out yourself, you can find the link to download the demo in the Supporting Materials section below!

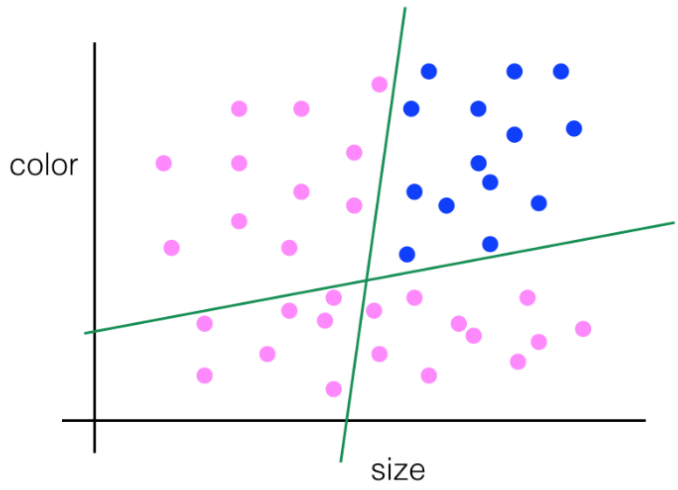
Supporting Materials

AffdexMe_desktop_demo

https://www.youtube.com/watch?v=dpFtXDqakvY&feature=emb_logo



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



Both lines, combined, clearly separate the car and bike data!

Image Formation

https://www.youtube.com/watch?time_continue=47&v=6ZVnQYzfpis&feature=emb_logo

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.

https://www.youtube.com/watch?v=RVNiaZuv6Ss&feature=emb_logo

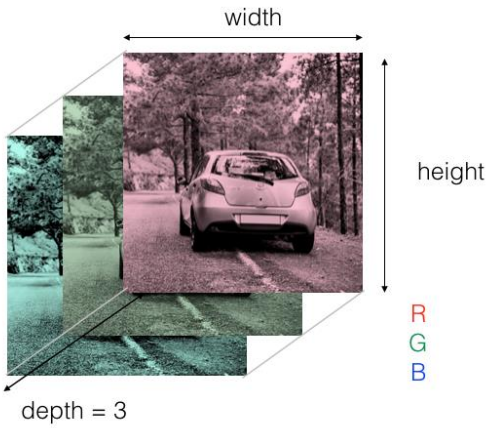
- Notebook: Images as Numerical Data

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.



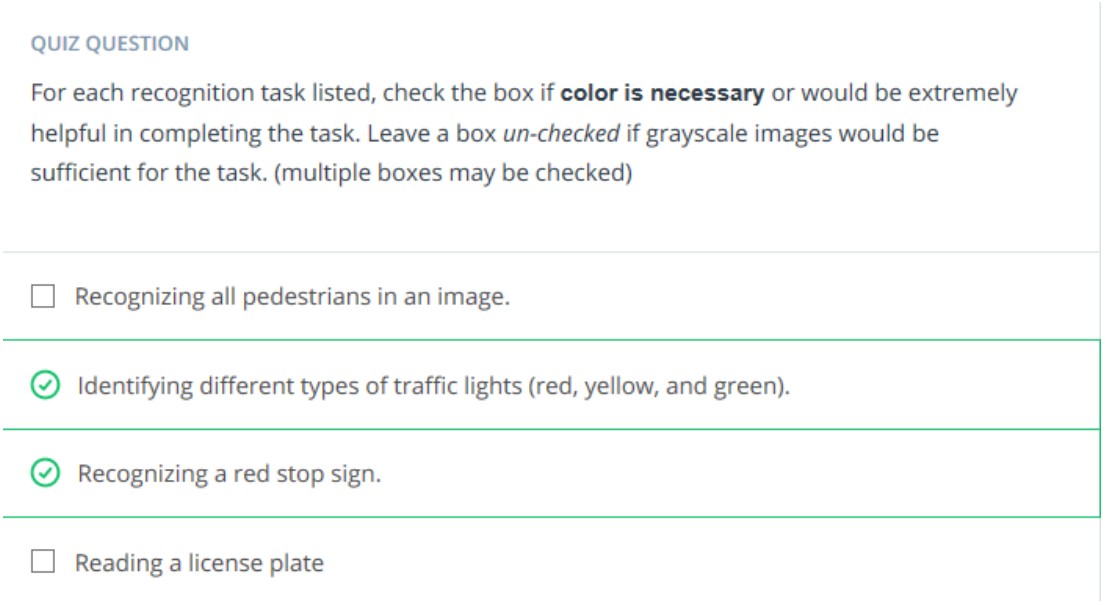
RGB layers of a car image.

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!

https://www.youtube.com/watch?v=-XbXiiGQ9gw&feature=emb_logo



- Notebook: Visualizing RGB Channels

Color Thresholds

https://www.youtube.com/watch?v=08ZlYZJaiUg&feature=emb_logo

Coding a Blue Screen

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**.

https://www.youtube.com/watch?v=jeeDryFxodk&feature=emb_logo

- Notebooks: Blue and green screen

Color Spaces and Transforms

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!

https://www.youtube.com/watch?v=B350aJVSsFc&feature=emb_logo

- Notebook: Color Conversion

Day and Night Classification

Now, you're on your way to being able to build a more complex computer vision application: an image classifier! You know how to analyze color and brightness in a given image, and that skill alone can help you distinguish between different images.

So, I'm going to give you a classification challenge: Classify two types of images, taken during either the day or at night (when the sun has set), and I want you to separate these images into two classes: day or night.

Two images of the same scene. One taken during the day (left) and one at night.

Visualizing the Data

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.

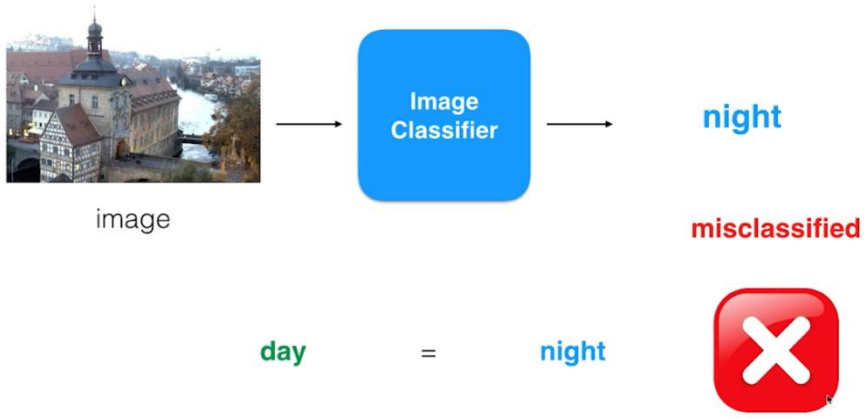
- Notebook: Load and visualize the data

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.

https://www.youtube.com/watch?v=FN96OM_JGyM&feature=emb_logo

Reflect

After visualizing the day and night images, what traits do you think distinguish the two classes?

Your reflection

I think the brightness matters, the value of the pixels should be nearer to 255 in order to be a day and it will be nearer to 0 for night.

Things to think about

There are many traits that distinguish a night from a day image. You may have thought about the sky: a day image has a bright and sometimes blue sky, and is generally brighter. Night images also often contain artificial lights, and so they have some small, very bright areas and a mostly dark background. All of these traits and more can help you classify these images!

QUESTION 2 OF 2

Say you have 500 day and night test images, and you send all of them through a classifier. What is the accuracy of this classifier if it *misclassifies* 80 images?

- 72%
- 80%
- ✓ 84%
- 92%

Features:

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.

https://www.youtube.com/watch?v=HshygbfQylA&feature=emb_logo

Standardizing Output

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]?

For the order [cat value, tiger value, hippopotamus value, dog value], what does a one-hot label of [0, 0, 0, 1] indicate?

---

it should indicate the dog value

- Notebook: Standardizing day and night images

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.

https://www.youtube.com/watch?v=oUlOS670uQg&feature=emb_logo

- Notebook: Averaging Brightness

Classification Task

Let's now complete our day and night classifier. After we extracted the average brightness value, we want to turn this feature into a predicted_label that classifies the image. Remember, we want to generate a numerical label, and again, since we have a binary dataset, I'll create a label that is a 1 if an image is predicted to be day and a 0 for images predicted to be night.

I can create a complete classifier by writing a function that takes in an image, extracts the brightness feature, and then checks if the average brightness is above some threshold X.

If it is, this classifier returns a 1 (day), and if it's not, this classifier returns a 0 (night)!

Next, you'll take a look at this notebook and get a chance to tweak the threshold parameter. Then, when you're able to generate predicted labels, you can compare them to the true labels, and check the accuracy of our model!

https://www.youtube.com/watch?v=LWD1M2vqXXo&feature=emb_logo

- Notebook: Classification

Evaluation Metrics

https://www.youtube.com/watch?v=fDN4D1QV674&feature=emb_logo

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!

- Note: Accuracy and misclassification

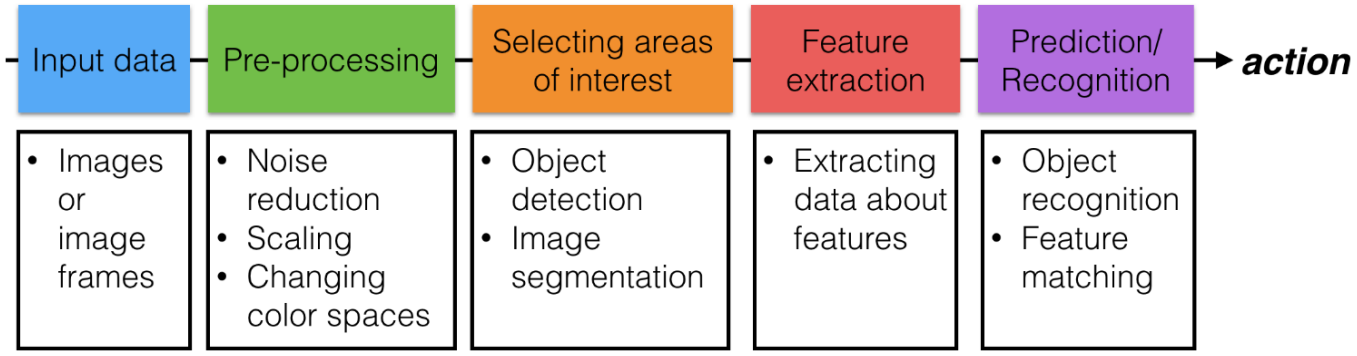
Review and the Computer Vision Pipeline

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!



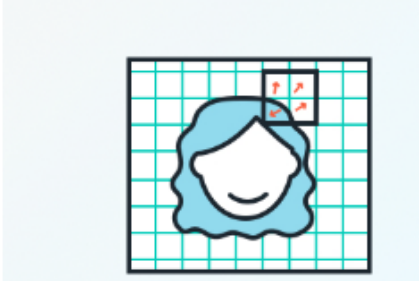
Computer Vision Pipeline.

Now, you're ready to build a more complex classifier, and learn more about feature extraction and deep learning architectures! Good luck and great work!!

Lesson 3: Convolutional Filters and edge detection

Convolutional Filters and Edge Detection

Learn about frequency in images and implement your own image filters for detecting edges and shapes in an image. Use a computer vision library to perform face detection.



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.

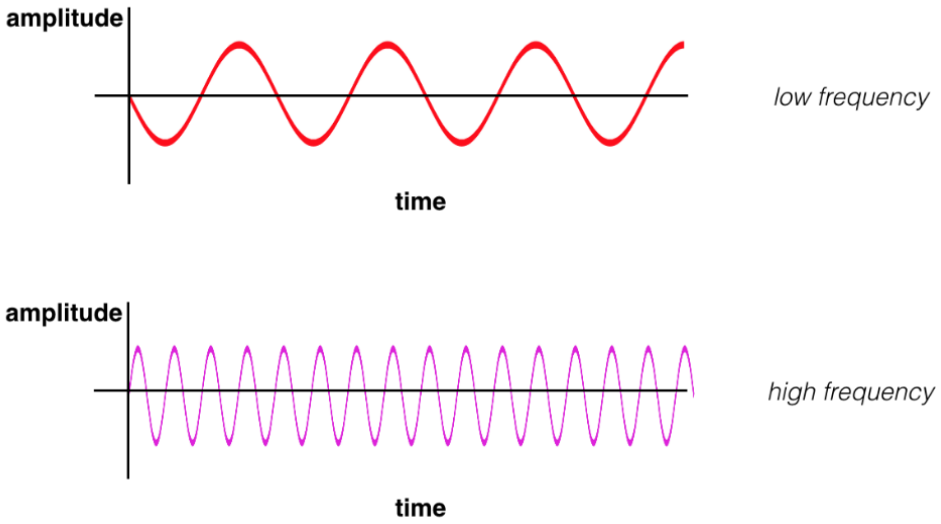
To detect these changes, you'll be using and creating specific image filters that look at groups of pixels and detect big changes in intensity in an image. These filters produce an output that shows these edges.

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

https://www.youtube.com/watch?v=f3H2EtiZLOQ&feature=emb_logo

Frequency in images

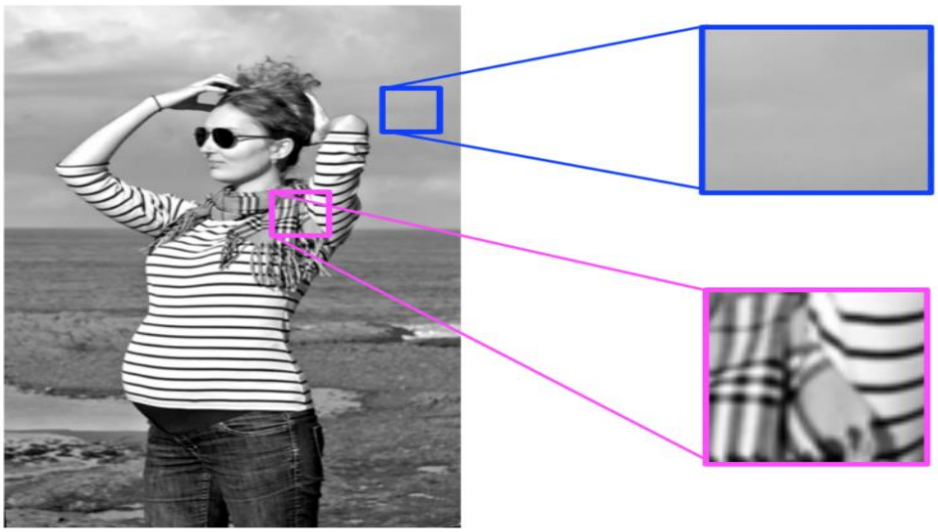
We have an intuition of what frequency means when it comes to sound. High-frequency is a high pitched noise, like a bird chirp or violin. And low frequency sounds are low pitch, like a deep voice or a bass drum. For sound, frequency actually refers to how fast a sound wave is oscillating; oscillations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

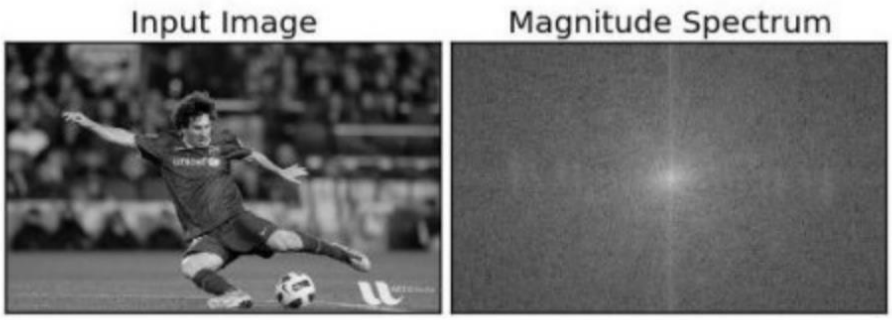
Most images have both high-frequency and low-frequency components. In the image above, on the scarf and striped shirt, we have a high-frequency image pattern; this part changes very rapidly from one brightness to another. Higher up in this same image, we see parts of the sky and background that change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

- Notebook: Fourier Transform

Image of left-leaning diagonal stripes of varying widths.

QUIZ QUESTION

For the input image of diagonal stripes, which of the options A-D, do you think is the most likely Fourier transform?

- A
- B
- C
- ✓ D

High-pass Filters

Edge Handling

Kernel convolution relies on centering a pixel and looking at it's surrounding neighbors. So, what do you do if there are no surrounding pixels like on an image corner or edge? Well, there are a number of ways to process the edges, which are listed below. It's most common to use padding, cropping, or extension. In extension, the border pixels of an image are copied and extended far enough to result in a filtered image of the same size as the original image.

Extend The nearest border pixels are conceptually extended as far as necessary to provide values for the convolution. Corner pixels are extended in 90° wedges. Other edge pixels are extended in lines.

Padding The image is padded with a border of 0's, black pixels.

Crop Any pixel in the output image which would require values from beyond the edge is skipped. This method can result in the output image being slightly smaller, with the edges having been cropped.

https://www.youtube.com/watch?time_continue=1&v=OpcFn_H2V-Q&feature=emb_logo

a)

-1	-1	-1
-1	8	-1
-1	-1	-1

b)

-1	0	1
-2	0	2
-1	0	1

c)

0	-1	0
-2	6	-2
0	-1	0

d)

-1	-2	-1
0	0	0
1	2	1

Four different kernels

QUIZ QUESTION

Of the four kernels pictured above, which would be best for finding and enhancing **horizontal** edges and lines in an image?

- a
- b
- c
- ✓ d

Creating a Filter

https://www.youtube.com/watch?time_continue=6&v=VEsdTRBH3D8&feature=emb_logo

Gradients and Sobel Filters

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 xxx or yyy direction. The operators for SobelxSobel_xSobelx and SobelySobel_ySobely, 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

xxx **vs.** yyy

In the above images, you can see that the gradients taken in both the xxx and the yyy directions detect the edges of the brain and pick up other edges. Taking the gradient in the xxx direction emphasizes edges closer to vertical. Alternatively, taking the gradient in the yyy 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 xxx **and** yyy directions, the magnitude is the square root of the sum of the squares.

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

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

abs_sobelxy=(sobelx)2+(sobely)2 = \sqrt{(sobel_x)^2+(sobel_y)^2}=(sobelx)2+(sobely)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 yyy gradient divided by the xxx gradient:

tan−1(sobely/sobelx)tan^{-1}{(sobel_y/sobel_x)}tan−1(sobely/sobelx).

- Notebook: finding edges

Low-pass Filters

https://www.youtube.com/watch?time_continue=4&v=jy5Be9vf2rk&feature=emb_logo

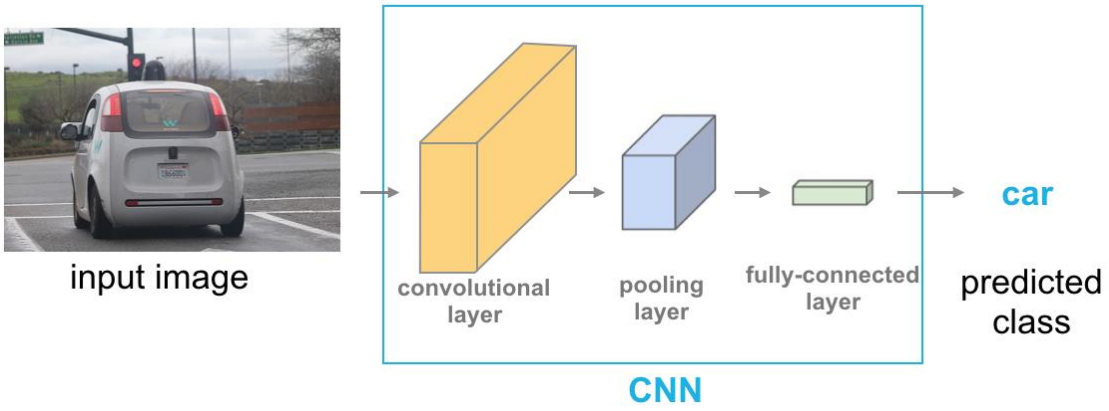
Gaussian Blur

https://www.youtube.com/watch?time_continue=5&v=1tlPonqIOrU&feature=emb_logo

- Notebook: Gaussian Blur
- Notebook: Fourier transformation of filters

The Importance of Filters

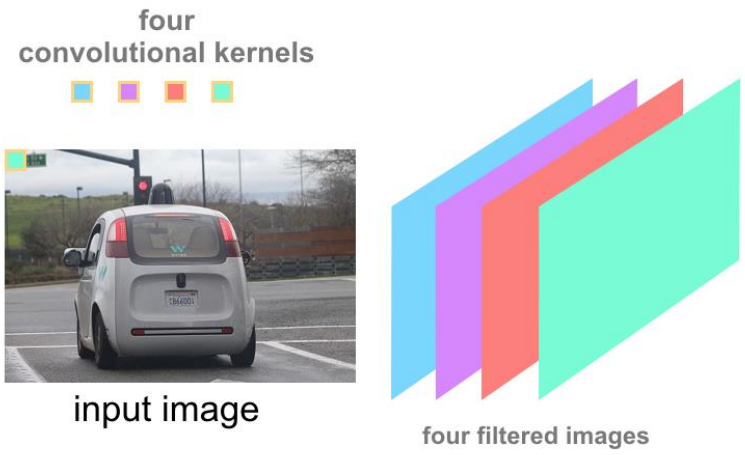
What you've just learned about different types of filters will be really important as you progress through this course, especially when you get to Convolutional Neural Networks (CNNs). CNNs are a kind of deep learning model that can learn to do things like image classification and object recognition. 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.

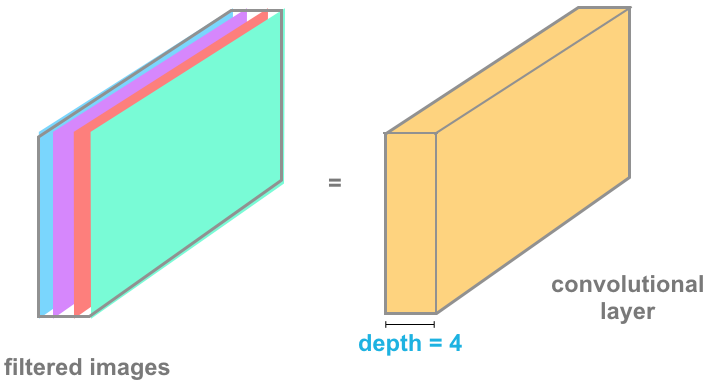
Convolutional Layer

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 object contain valuable information about the shape of an object.

Canny Edge Detector

https://www.youtube.com/watch?time_continue=11&v=Nm1H2xMMS3I&feature=emb_logo

Shape Detection

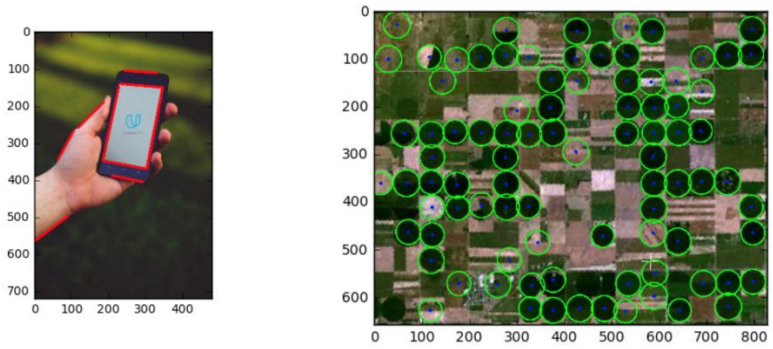
Edge Detection

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.

Hough Transform

https://www.youtube.com/watch?time_continue=3&v=RIOzsF1_xuo&feature=emb_logo

Let's test your intuition about the Hough transform, and look at this image of a dots in a square pattern. What will this image look like after applying a Hough transform?

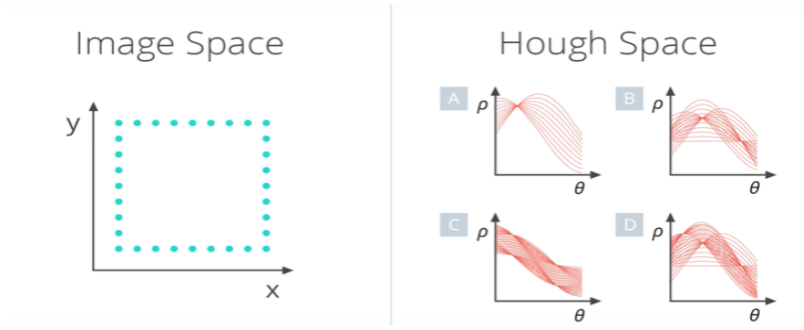


Image space and potential Hough space transforms

QUIZ QUESTION

What will this image look like in Hough space? Choose the correct plot.

- A
- B
- C

Hough Line Detection

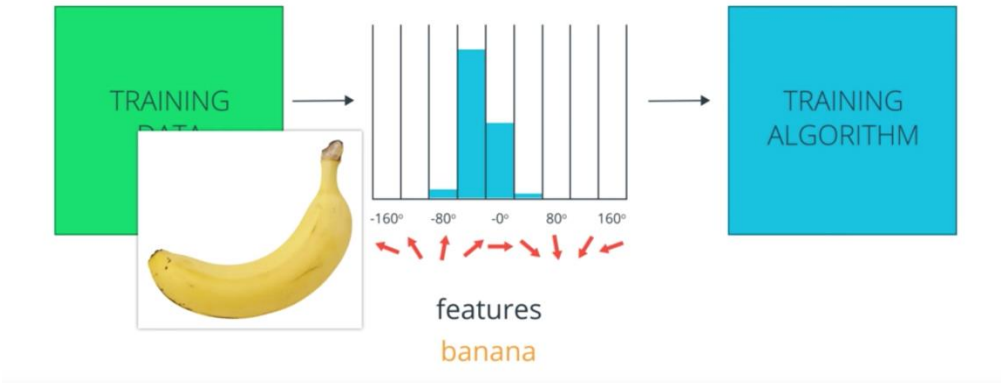
https://www.youtube.com/watch?time_continue=6&v=PVIUkQKbrUw&feature=emb_logo

- Notebook: Hough Detections

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.

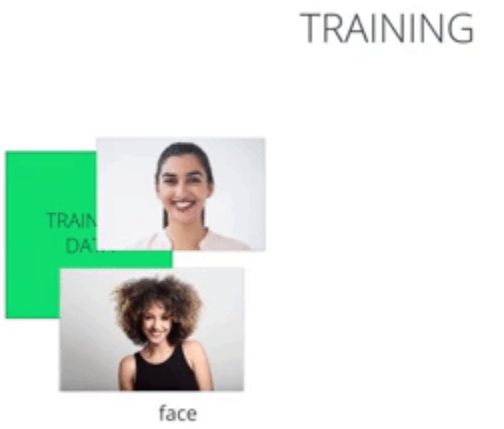


Training data (an image of a banana) going through some feature extraction and classification steps.

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.

Next, let's learn more about this algorithm and implement a face detector in code!

Haar Cascades

https://www.youtube.com/watch?time_continue=2&v=vALIpxVfKRc&feature=emb_logo

- Notebook: Haar cascade facial detection

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.

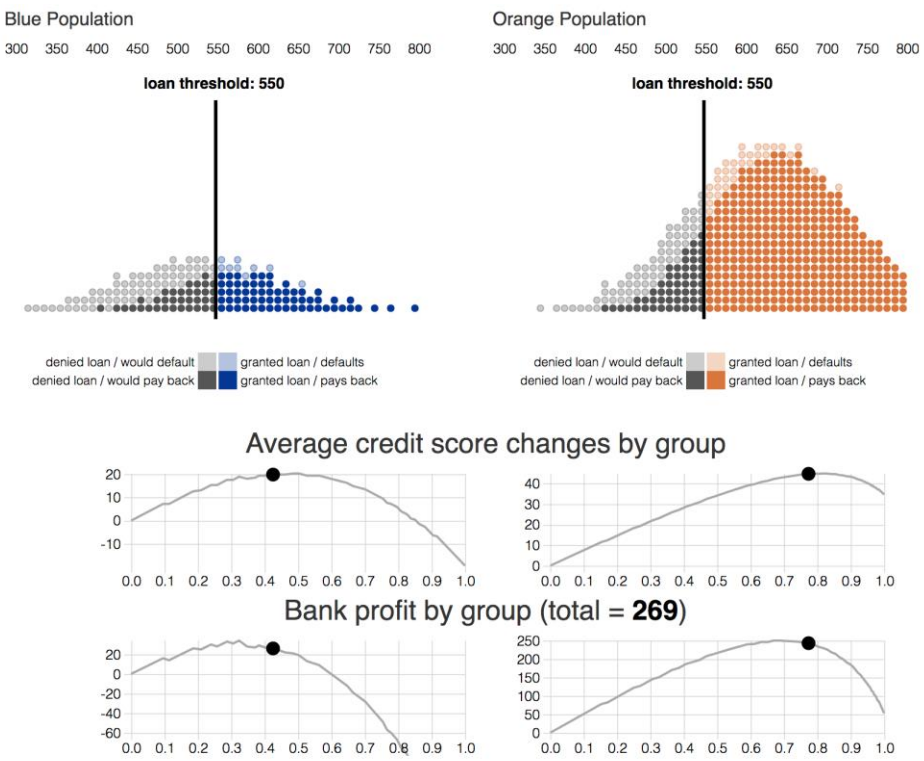
The computer scientist, **Joy Buolamwini**, based out of the MIT Media Lab, has studied bias in decision-making algorithms, and her work has revealed some of the extent of this problem. One study looked at the error rates of facial recognition programs for women by shades of skin color; results pictured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

Supporting Materials

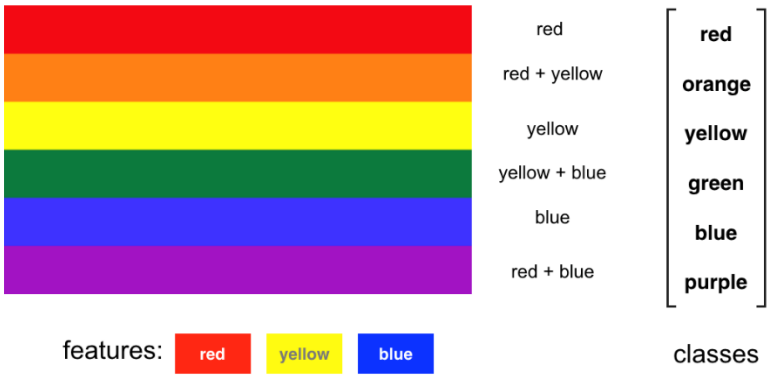
Gender Shades: Intersectional Accuracy Disparities in Commercial Gender Classification

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.

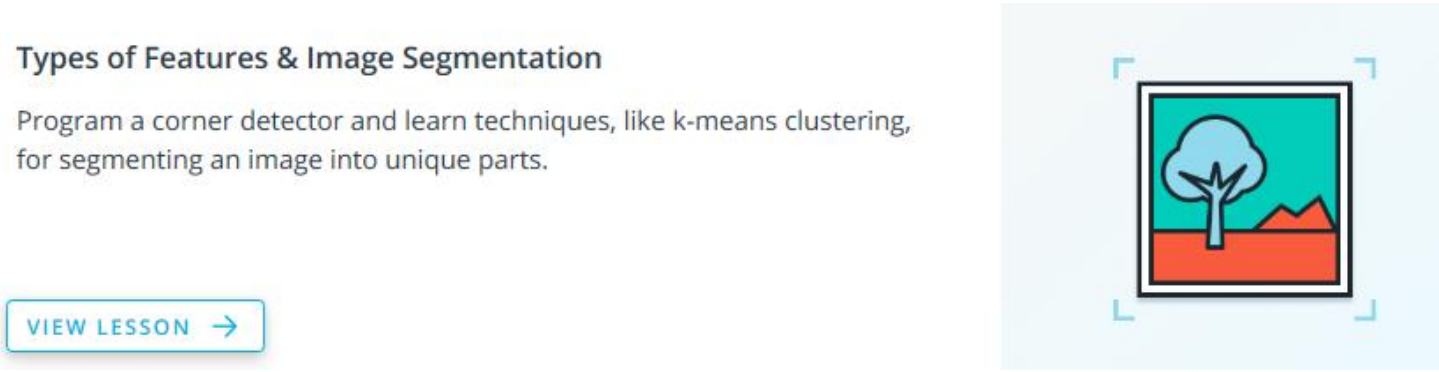
As you continue learning, keep in mind that selecting the right feature is an important computer vision task.

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.

Lesson 3: Types of features and Image Segmentation



Types of Features

https://www.youtube.com/watch?time_continue=1&v=cJHro29nzgg&feature=emb_logo

Corner Detectors

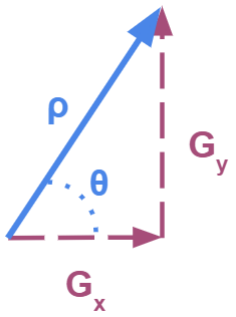
https://www.youtube.com/watch?v=jemzDq07MEI&feature=emb_logo

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**.

- Notebook: find the corners

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)
```

erosion

original

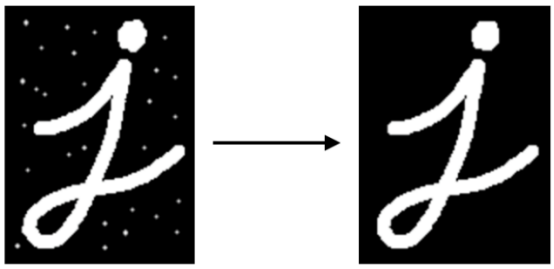
The letter "j": (left) erosion, (middle) original image, (right) dilation

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

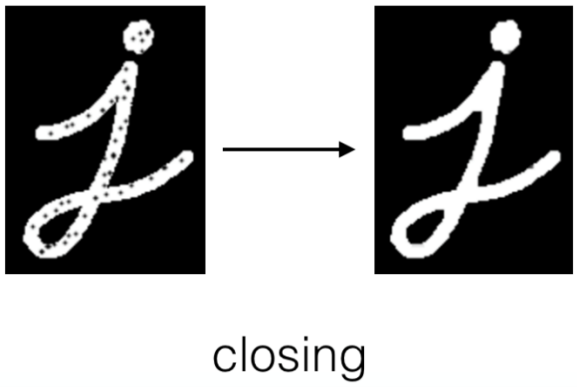
Opening

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

Closing

Many of these operations try to extract better (less noisy) information abo

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.

Image Contours

https://www.youtube.com/watch?time_continue=27&v=Wcbrl7Wr_kU&feature=emb_logo

- Notebook: Find contors and features

Solution: Solution notebook

K-means Clustering

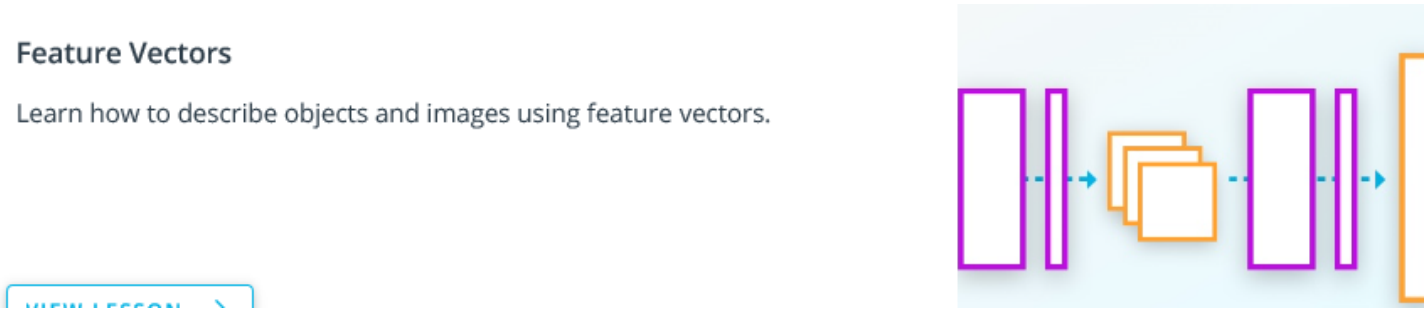
https://www.youtube.com/watch?time_continue=1&v=Cf_LSDCEBzk&feature=emb_logo

K-means Implementation

https://www.youtube.com/watch?time_continue=5&v=poKlg-aB4rU&feature=emb_logo

- Notebook: K-means Clustering

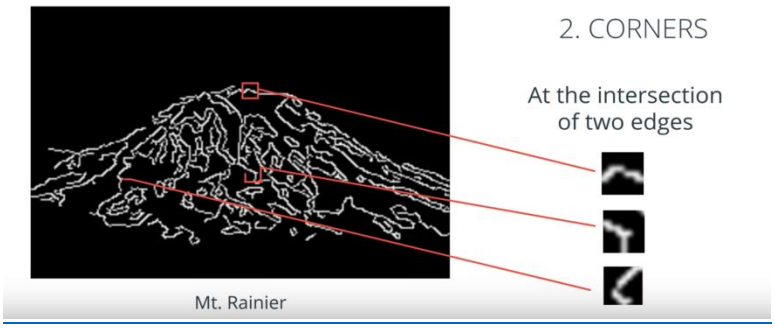
Lesson 4: Feature Vectors



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.

Feature Vectors

https://www.youtube.com/watch?v=-PF1_MITrOw&feature=emb_logo

Real-Time Feature Detection

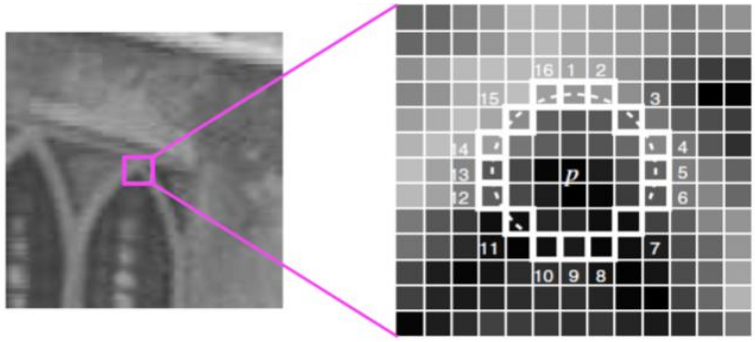
https://www.youtube.com/watch?time_continue=4&v=zPxylrXf-Gs&feature=emb_logo

Introduction to ORB

https://www.youtube.com/watch?time_continue=57&v=WN37zcMhMas&feature=emb_logo

FAST

https://www.youtube.com/watch?v=DCHAc6fjcVM&feature=emb_logo



Zoomed in patch around an arch in a building. Original image taken from OpenCV's documentation.

QUIZ QUESTION

Do you think the pixel, p, above, will be identified as a keypoint by the FAST algorithm?

Yes

BRIEF

https://www.youtube.com/watch?time_continue=21&v=EKIPEPpRciw&feature=emb_logo

Scale and Rotation Invariance

https://www.youtube.com/watch?time_continue=4&v=2k3T6rfjvx0&feature=emb_logo

- Notebook: Image Pyramids

Feature Matching

https://www.youtube.com/watch?time_continue=20&v=RH05Wnl1-2A&feature=emb_logo

ORB in Video

https://www.youtube.com/watch?time_continue=2&v=Vzs6B1dFQC0&feature=emb_logo

- Notebook: Implementing ORB

HOG

https://www.youtube.com/watch?v=dqe9zGtxoNM&feature=emb_logo

- Notebook: Implementing HOG

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.
Good work so far!

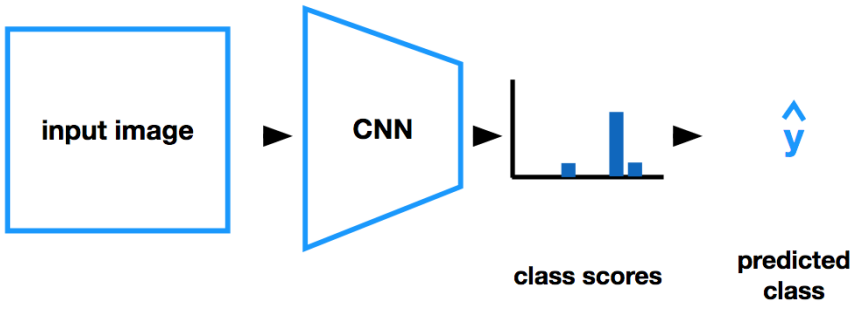
Lesson 5: CNN Layers & Feature Visualization

CNN Layers and Feature Visualization

Define and train your own convolution neural network for clothing recognition. Use feature visualization techniques to see what a network has learned.

CNN Structure

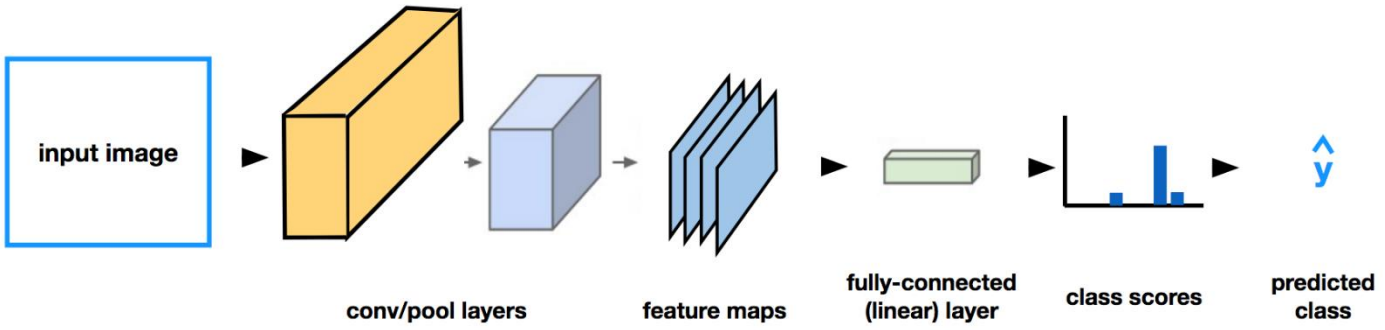
A classification CNN takes in an input image and outputs a distribution of class scores, from which we can find the most likely class for a given image. As you go through this lesson, you may find it useful to consult **this blog post**, which describes the image classification pipeline and the layers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

https://www.youtube.com/watch?v=hT6zBYCuAfw&feature=emb_logo

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**.

Lesson Outline and Data

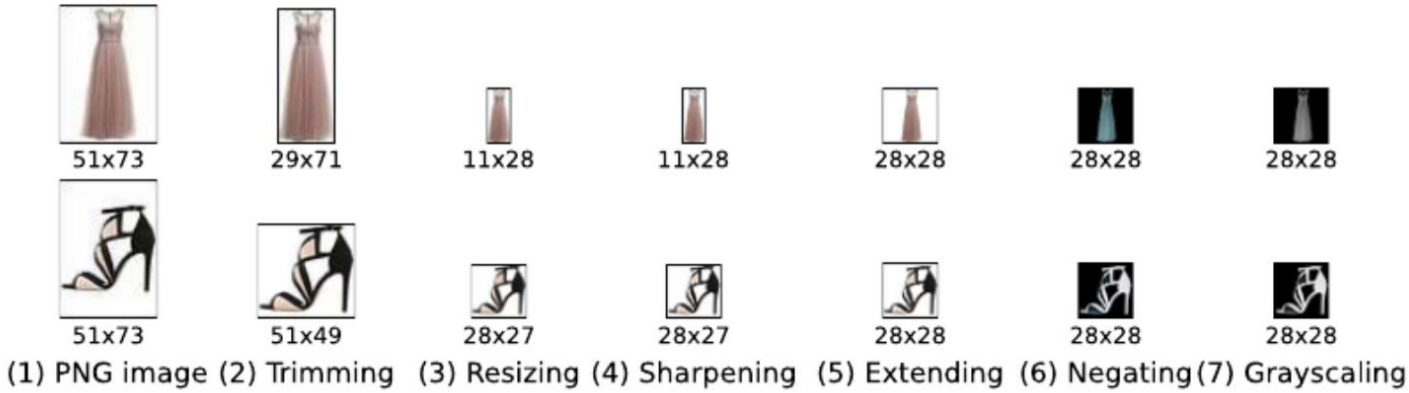
https://www.youtube.com/watch?v=jPr-5aZA6NE&feature=emb_logo

Check out the FashionMNIST dataset, **here**.

[Animation of FashionMNIST clothing images, grouped by class in 3D space. (Original image from the dataset repository)]

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.]

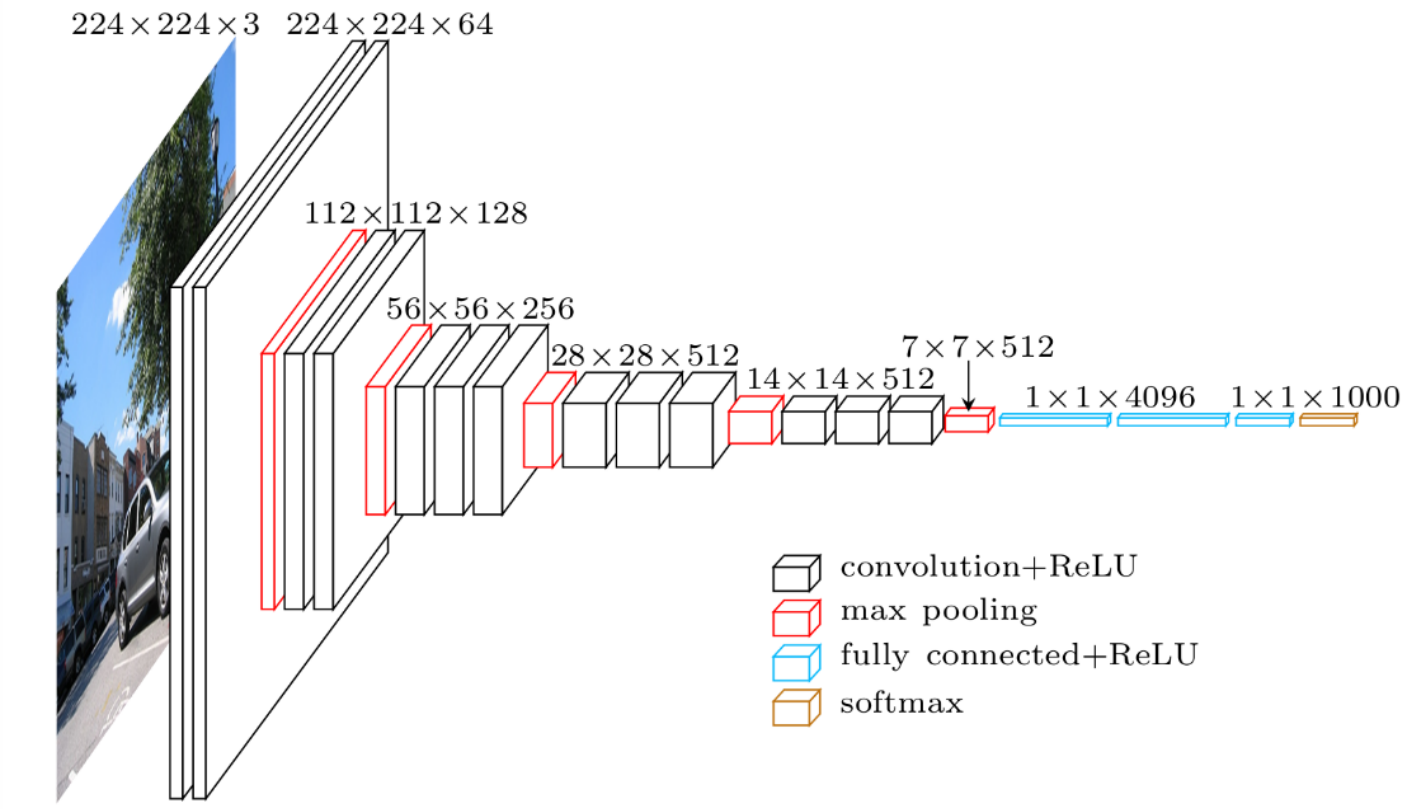
A note on Workspaces

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 Alexis

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.

Alexis Cook

Convolutional Layers

https://www.youtube.com/watch?time_continue=8&v=LX-yVob3c28&feature=emb_logo

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 convol
ution 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:

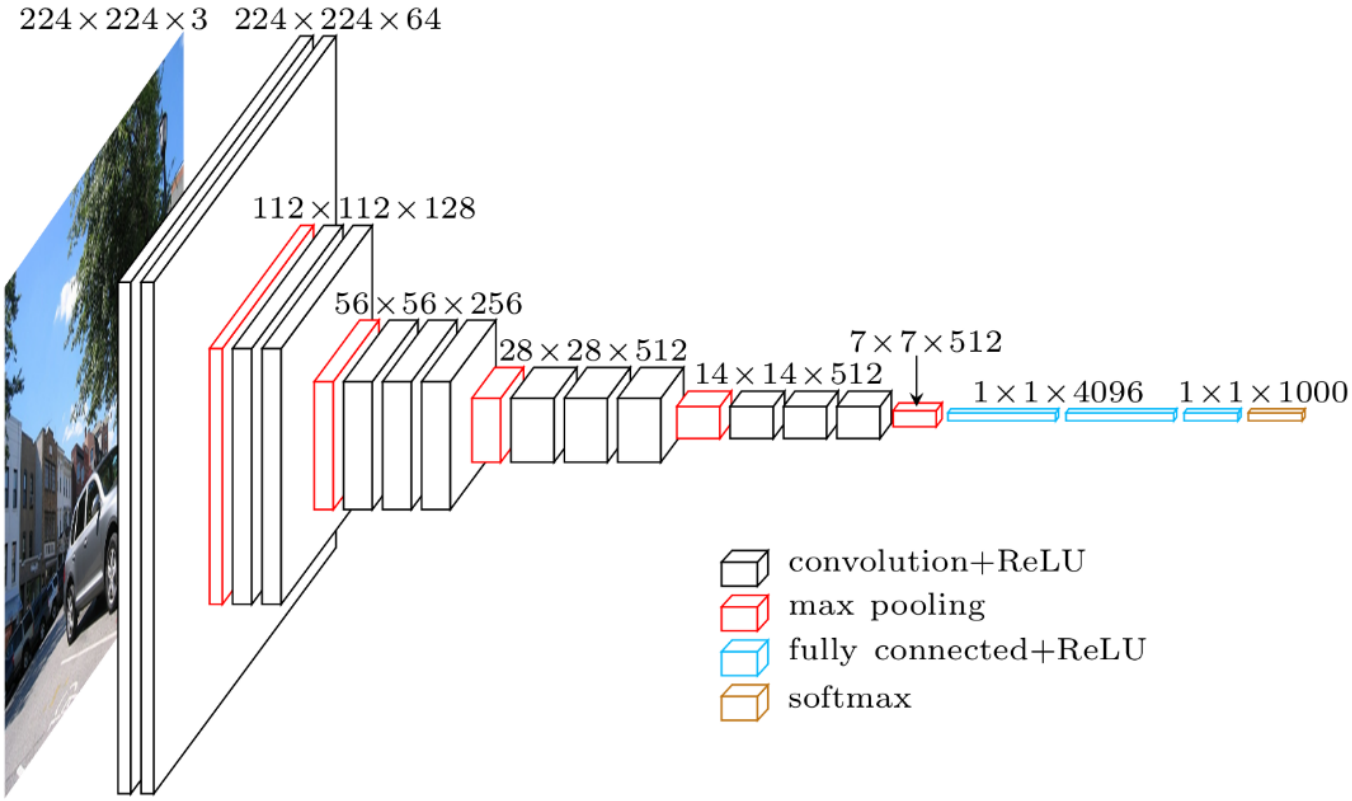
```
x = self.pool(F.relu(self.conv1(x)))
```

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.

- Notebook: Visualizing a convolutional Layer

VGG-16 Architecture

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

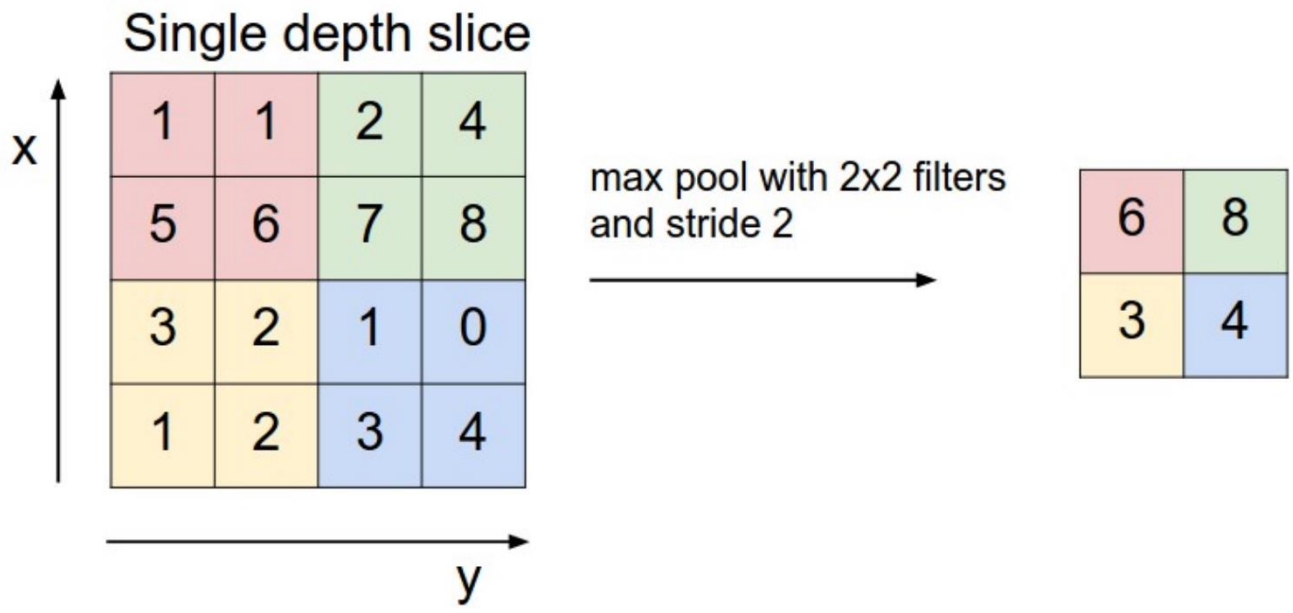


VGG-16 architecture

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

Next, let's learn more about how these pooling layers work.

Pooling Layers

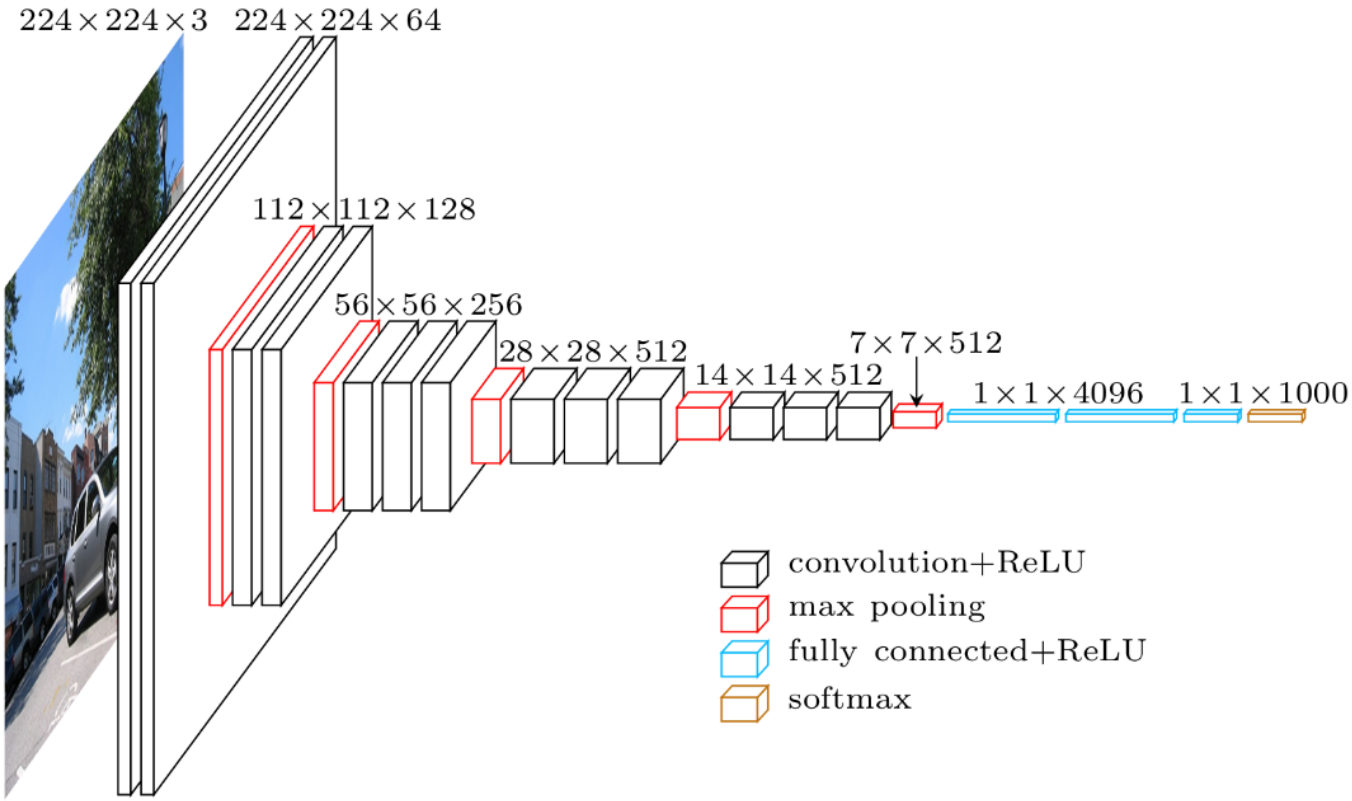
https://www.youtube.com/watch?time_continue=7&v=OkkIZNs7Cyc&feature=emb_logo

- Notebook: visualizing a pooling layer

Fully-Connected Layers, VGG-16

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!

- Notebook: Visualizing FashionMNIST

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!

- **Notebook: Fashion MNIST training excersie**
- Notebok: **Fashion MNIST solution 1**

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.

For a recap of what dropout does, check out Luis's video below.

https://www.youtube.com/watch?v=Ty6K6YiGdBs&feature=emb_logo

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!

Check out the video below for a review of how momentum works, mathematically.

https://www.youtube.com/watch?v=r-rYz_PEWC8&feature=emb_logo

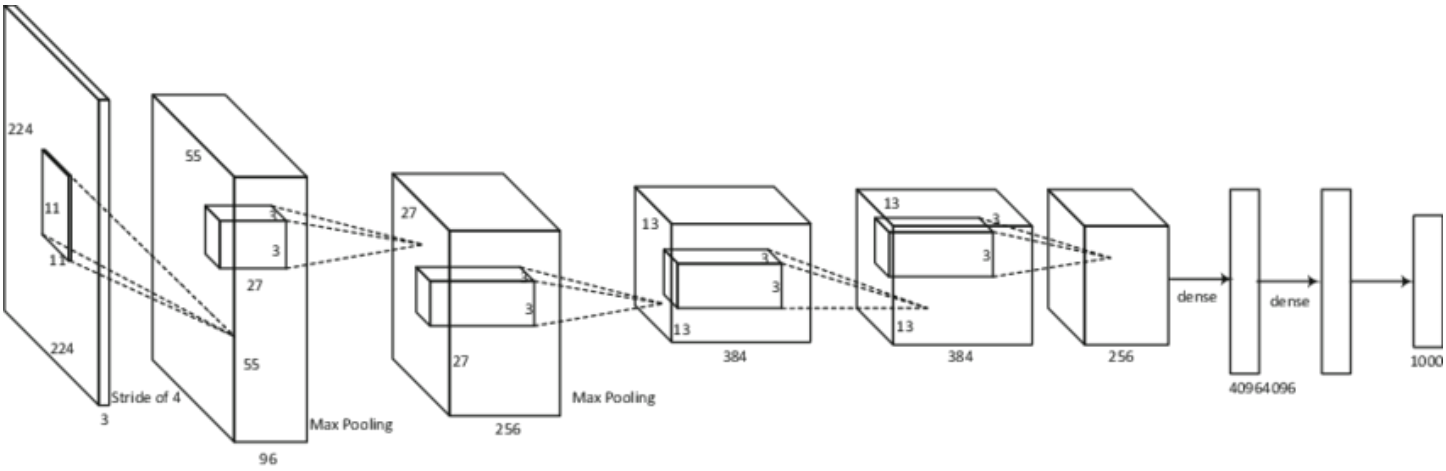
- Notebook: Fashion MNIST Solution 2

Network Structure

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!

Feature Visualization

https://www.youtube.com/watch?time_continue=4&v=xwGa7RFg1EQ&feature=emb_logo

Feature Maps

https://www.youtube.com/watch?time_continue=2&v=oRhsJHHWtu8&feature=emb_logo

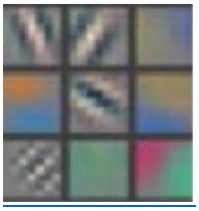
First Convolutional Layer

https://www.youtube.com/watch?time_continue=39&v=hIHDMWVSfsM&feature=emb_logo

Visualizing CNNs

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

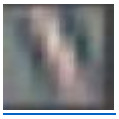
The CNN we will look at is trained on ImageNet as described in this paper by Zeiler and Fergus. In the images below (from the same paper), we'll see *what* each layer in this network detects and see *how* each layer detects more and more complex ideas.



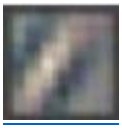
Example patterns that cause activations in the first layer of the network. These range from simple diagonal lines (top left) to green blobs (bottom middle).

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.

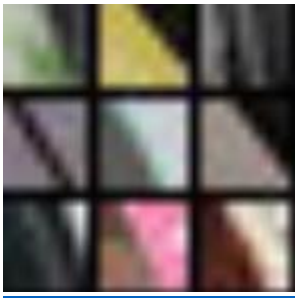


As visualized here, the first layer of the CNN can recognize -45 degree lines.



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

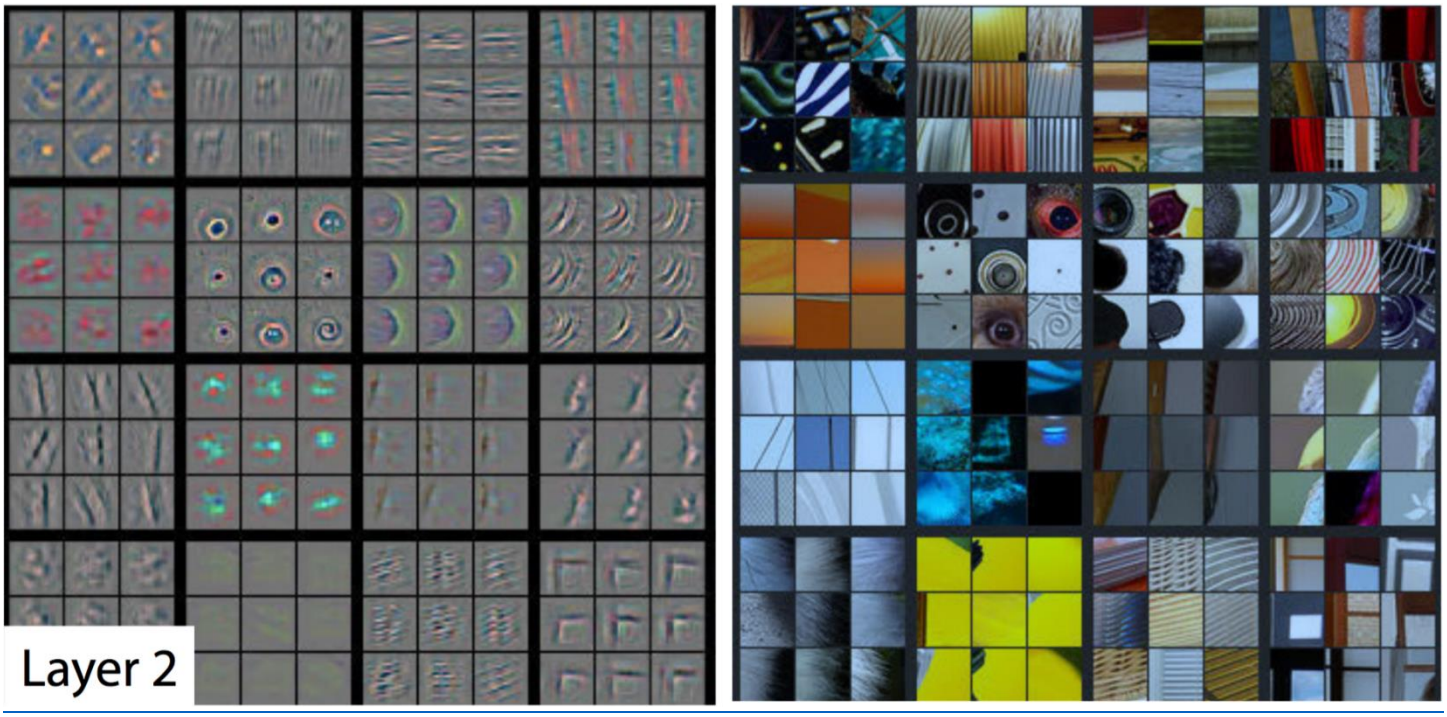
Let's now see some example images that cause such activations. The below grid of images all activated the -45 degree line. Notice how they are all selected despite the fact that they have different colors, gradients, and patterns.



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



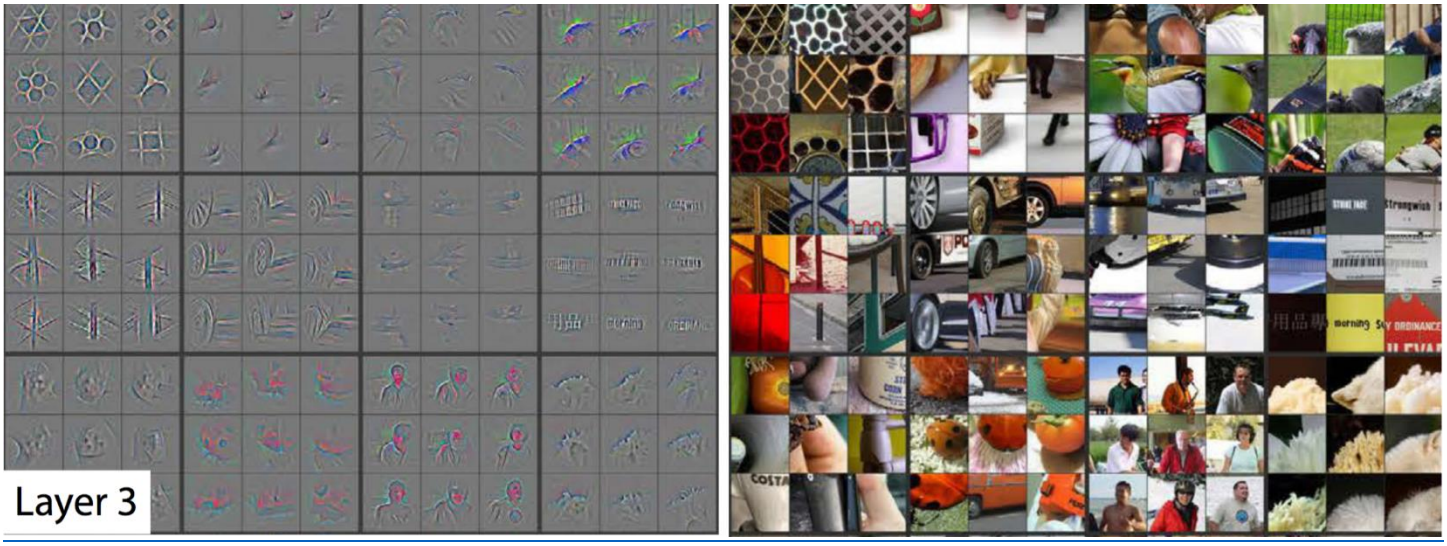
A visualization of the second layer in the CNN. Notice how we are picking up more complex ideas like circles and stripes. The gray grid on the left represents how this layer of the CNN activates (or "what it sees") based on the corresponding images from the grid on the right.

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

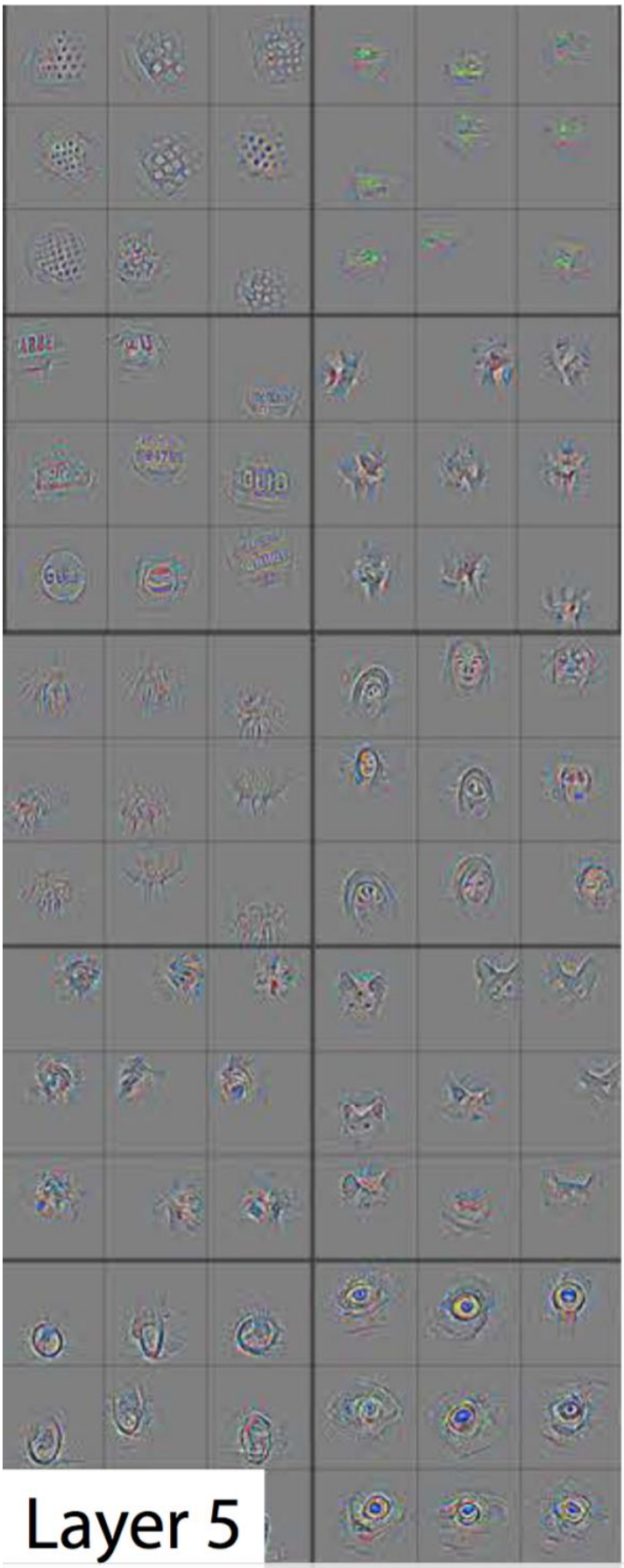


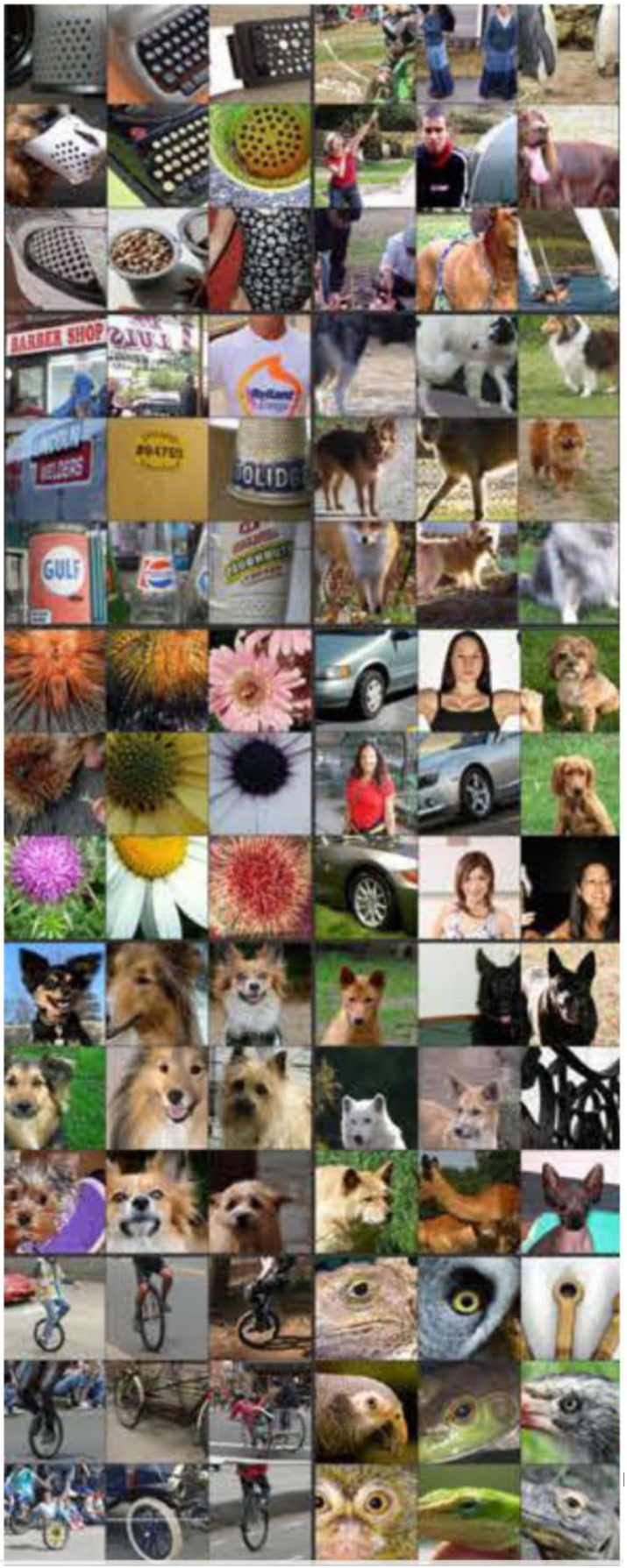
A visualization of the third layer in the CNN. The gray grid on the left represents how this layer of the CNN activates (or "what it sees") based on the corresponding images from the grid on the right.

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


BARBER SHOP
GULF

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.

Visualizing Activations

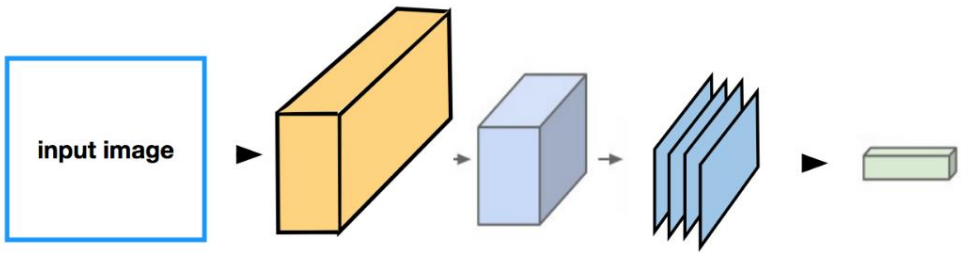
https://www.youtube.com/watch?time_continue=7&v=CJLNTOXqt3I&feature=emb_logo

- Notebook: Feature Viz for FashionMNIST

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

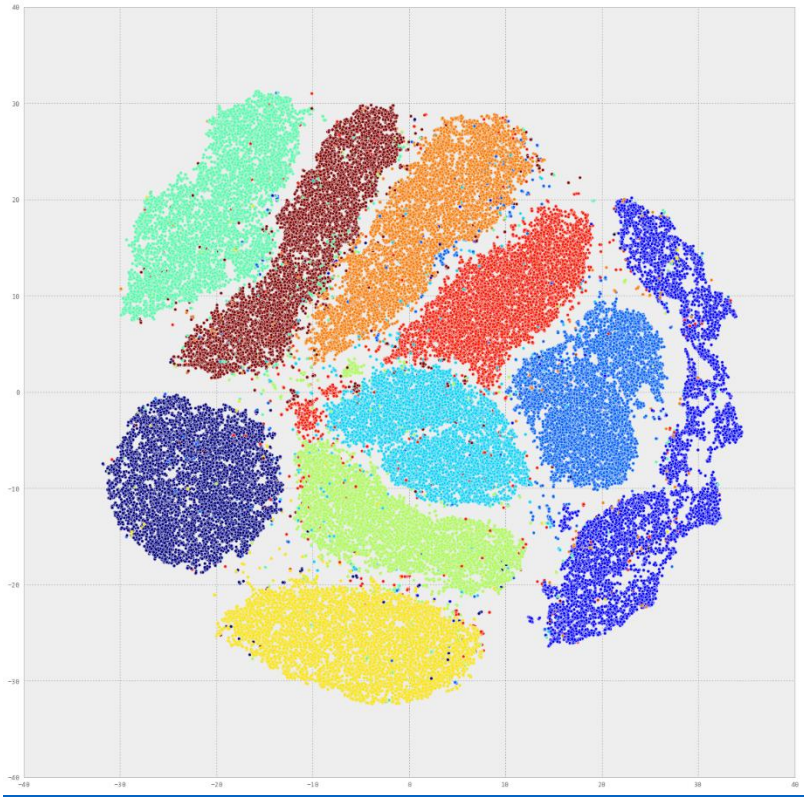
One is PCA, principal component analysis, which takes a high dimensional vector and compresses it down to two dimensions. It does this by looking at the feature space and creating two variables (x, y) that are functions of these features; these two variables want to be as different as possible, which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



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.

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!

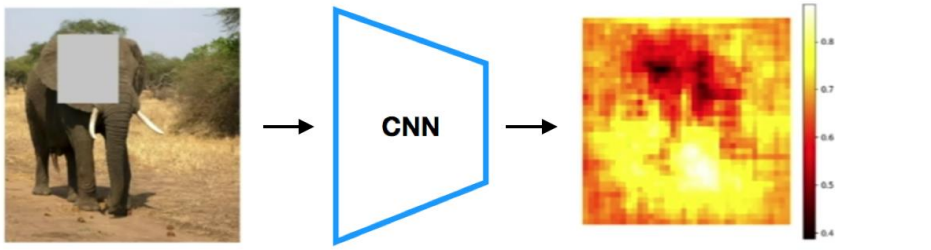
Occlusion Experiments

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.

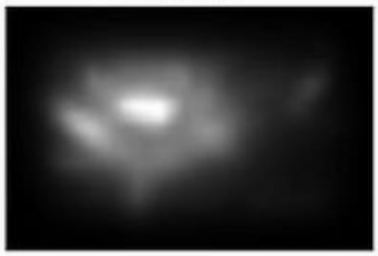
Saliency maps aim to show these important pictures by computing the gradient of the class score with respect to the image pixels. A gradient is a measure of change, and so, the gradient of the class score with respect to the image pixels is a measure of how much a class score for an image changes if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

Measuring change

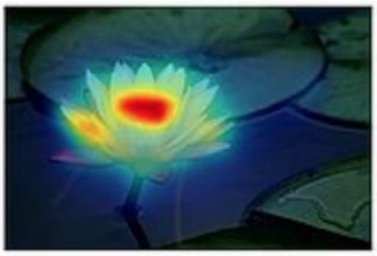
A saliency map tells us, for each pixel in an input image, if we change it's value slightly (by *dp*), how the class output will change. If the class scores change a lot, then the pixel that experienced a change, dp, is important in the classification task.

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



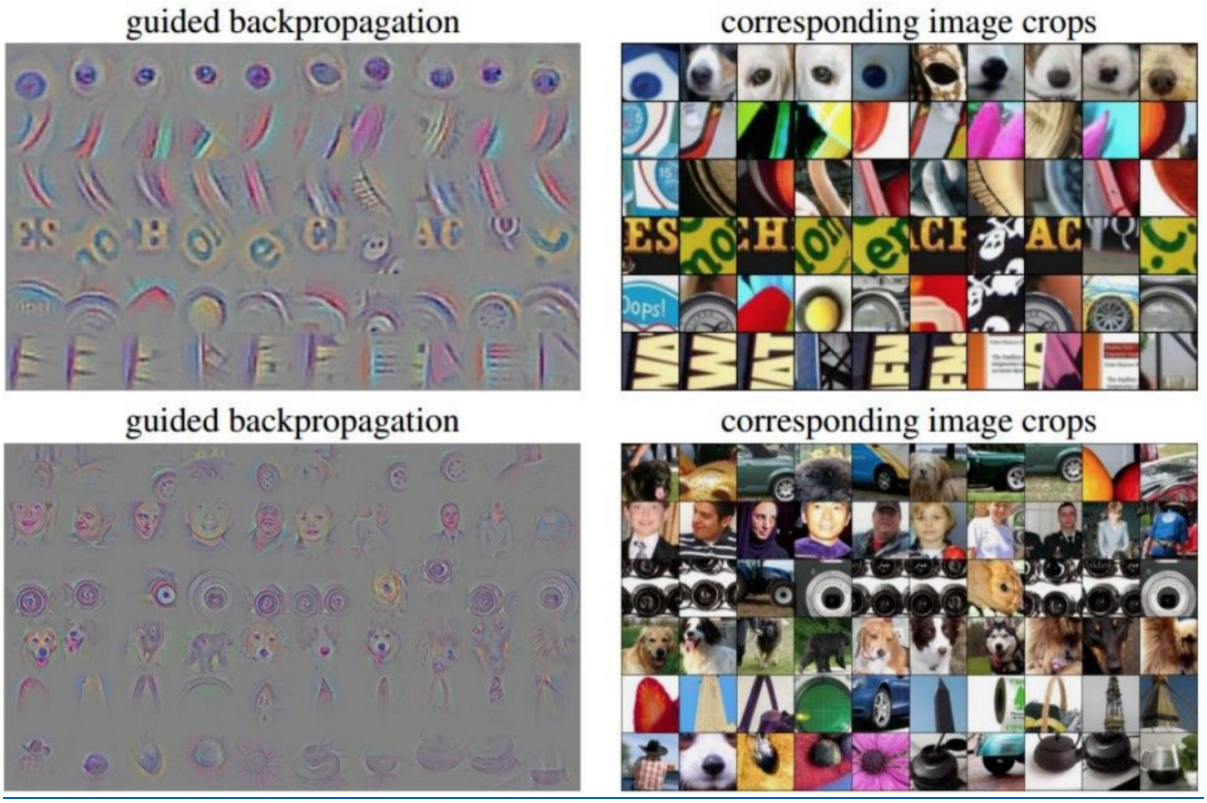
map overlay

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.

Supporting Materials

Visualizing Conv Nets

Guided Backprop Network Simplicity

Summary of Feature Viz

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 wave

The Project

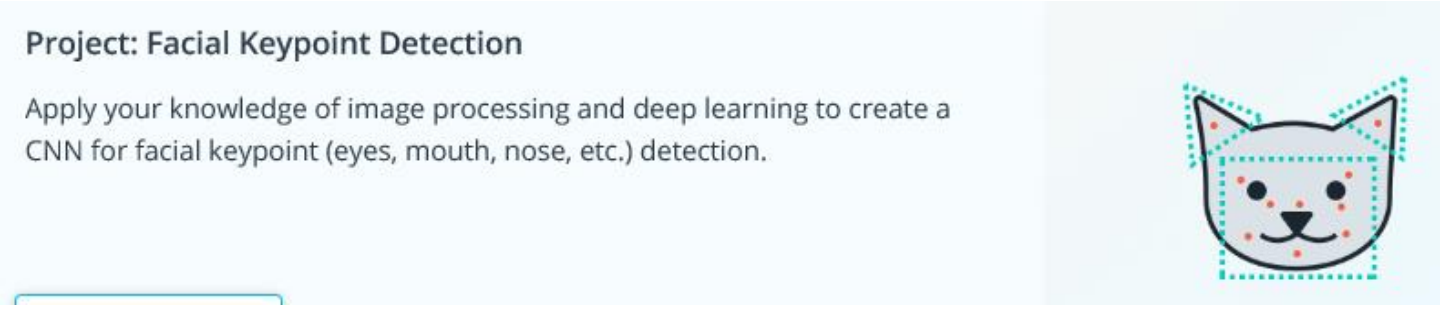
Next, you'll be tasked with completing an end-to-end facial keypoint detection system that learns to locate the eyes, mouth, and edges of a face in an image. This is an interesting regression challenge, and will test your knowledge of CNN architectures!

[Facial keypoint detection, example structure.]

Further Resources and a mini-project (optional)

If you complete the project, and are looking for another challenge to solve with computer vision and deep learning, check out the **Elective Section: Skin Cancer Detection** in which Sebastian Thrun walks you through his use of CNNs for detecting skin cancer. This is an ongoing filed of research and it is extremely challenging to create a *good* diagnostic model.

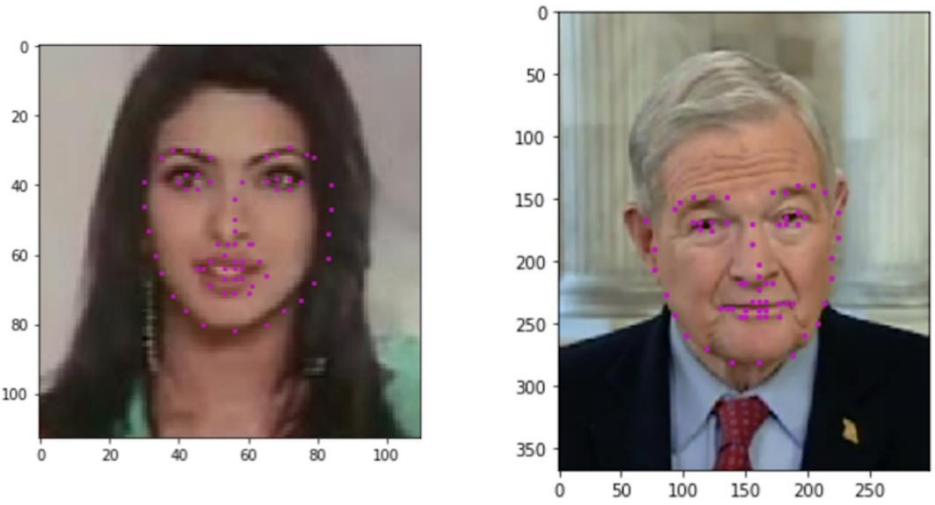
Project: Facial Keypoint Detection



Project Overview

In this project, you'll combine your knowledge of computer vision techniques and deep learning architectures to build a facial keypoint detection system that takes in any image with faces, and predicts the location of 68 distinguishing keypoints on each face!

Facial keypoints include points around the eyes, nose, and mouth on a face and are used in many applications. These applications include: facial tracking, facial pose recognition, facial filters, and emotion recognition. Your completed code should be able to look at any image, detect faces, and predict the locations of facial keypoints on each face. Some examples of these keypoints are pictured below.



Facial keypoints displayed on two images, each of which contains a single face.

Project Instructions

The project will be broken up into a few main parts in four Python notebooks, **only Notebooks 2 and 3 (and the models.py file) will be graded**:

Notebook 1 : Loading and Visualizing the Facial Keypoint Data

Notebook 2 : Defining and Training a Convolutional Neural Network (CNN) to Predict Facial Keypoints

Notebook 3 : Facial Keypoint Detection Using Haar Cascades and your Trained CNN

Notebook 4 : Fun Filters and Keypoint Uses

You can find these notebooks in the Udacity workspace that appears in the concept titled **Project: Facial Keypoint Detection**. This workspace provides a Jupyter notebook server directly in your browser.

Note: This project does not require the use of GPU, so this project does not include instructions for GPU setup.

You can also choose to complete this project in your own local repository (and may use GPU, there), and you can find all of the project files in this **GitHub repository**. Note that while you are *allowed* to complete this project on your local computer, you are **strongly** encouraged to complete the project from the workspace.

Note: There is a final, 5th notebook that is there to guide you once you are ready to submit your project!

Evaluation

Your project will be reviewed by a Udacity reviewer against the Facial Keypoint Detection project **rubric**. Review this rubric thoroughly, and self-evaluate your project before submission. All criteria found in the rubric must meet specifications for you to pass.

Ready to submit your project?

Zipping project files, manually

There are instructions in a 5th, final notebook, for compressing your project and work into a zip file, which you can download and submit. **This is the recommended submission approach.**

Once you've completed your project, at the end of this lesson you'll see a "Submit Project" button; click it and follow the instructions to submit!

If you decide to submit directly from the workspace please make sure:

1. You do not have any image files in your Jupyter notebook home directory
2. You have deleted any large model checkpoints in the home directory

When workspaces are submitted, everything in the home directory is compressed and submitted in a zip file. Deleting large files that are not graded, will make sure that your submission will not be too large!

The Workspace

Next, you'll load the project workspace and follow detailed instructions there to complete this project! The workspace may take a minute or two to load the facial image/keypoint data.

If you want to ask questions about this project or share advice, post in the Study Groups or Knowledge.

Best Practices

Follow the best practices outlined below to avoid common issues with Workspaces.

- **Keep your home folder small**

Your home folder (including subfolders) must be less than 2GB or you may lose data when your session terminates. You may use directories outside of the home folder for more space, but only the contents of the home folder are persisted between sessions and submitted with your project.

NOTE: Your home folder (including subfolders) must be less than 25 megabytes to submit as a project. If the site becomes unresponsive when you try to submit your project, it is likely that your home folder is too large. You can check the size of your home folder by opening a terminal and running the command du -h . | tail -1 You can use ls to list the files in your terminal and rm to remove unwanted files. (Search for both commands online to find example usage.)

- **What's the "home folder"?**

"Home folder" refers to the directory where users files are stored (compared to locations where system files are stored, for example). (Ref. Wikipedia: **home directory**) In Workspaces, the home folder is /home/workspace. Any files in this folder or any subfolder are part of your home folder contents, which means they're saved between sessions and transferred automatically when you switch between CPU/GPU mode.

The folder /tmp is *not* in the home folder; files in any folder outside your home folder are **not** persisted between sessions or transferred between CPU/GPU mode. You can create a folder outside the home folder using the command mkdir from a terminal. For example you could create a temporary folder to store data using mkdir -p /data to create a folder at the root directory. You will need to recreate the folder and recreate any data inside every time you start a new Workspace session.

- **Keeping your connection alive during long processes**

Workspaces automatically disconnect when the connection is inactive for about 30 minutes, which includes inactivity while deep learning models are training. You can use the workspace_utils.py module **here** to keep your connection alive during training. The module provides a context manager and an iterator wrapper—see example use below.

NOTE: The script sometimes raises a connection error if the request is opened too frequently; just restart the jupyter kernel & run the cells again to reset the error.

NOTE: These scripts will keep your connection alive while the training process is *running*, but the workspace will still disconnect 30 minutes after the last notebook cell finishes. Modify the notebook cells to **save your work** at the end of the last cell or else you'll lose all progress when the workspace terminates.

Example using context manager:

```
from workspace_utils import active_session


with active_session():
    # do long-running work here
```

Example using iterator wrapper:

```
from workspace_utils import keep_awake


for i in keep_awake(range(5)):
```

 # do iteration with lots of work here

- **Manage your GPU time**

It is important to avoid wasting GPU time in Workspace projects that have GPU acceleration enabled. The benefits of GPU acceleration are most useful when evaluating deep learning models—especially during *training*. In most cases, you can build and test your model (including data pre-processing, defining model architecture, etc.) in CPU mode, then activate GPU mode to accelerate training.

- **Handling "Out of Memory" errors**

This issue isn't specific to Workspaces, but rather it is an apparent issue between Pytorch & Jupyter, where Jupyter reports "out of memory" after a cell crashes. Jupyter holds references to active objects as long as the kernel is running—including objects created before an error is raised. This can cause Jupyter to persist large objects in memory long after they are no longer required. The only known solution so far is to reset the kernel and run the notebook cells again.

Supporting Materials

workspace_utils.py

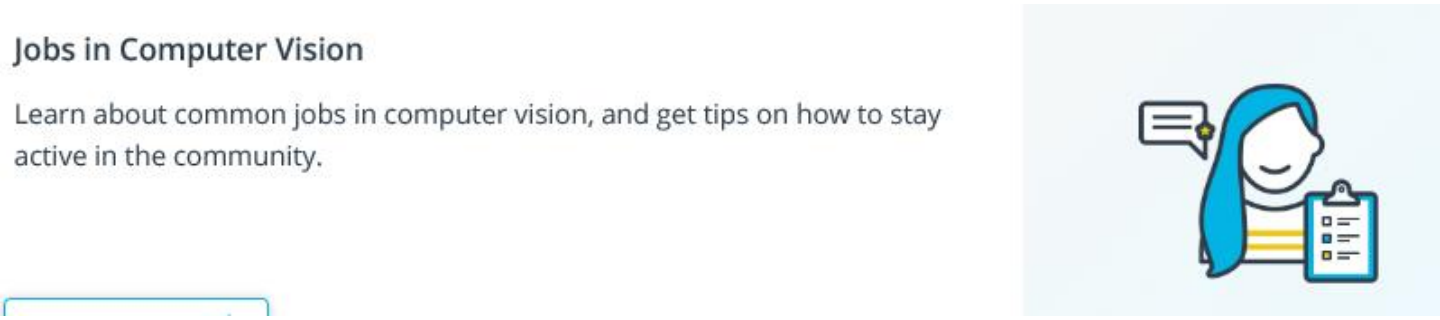
Meets Specifications

You have done a fantastic job completing the facial keypoint detection project. The facial keypoint detection is a well known machine learning challenge. You might want to check the resource below to further improve your model.

1. Facial keypoints detection using Neural Network.
2. Facial Keypoints Detection: An Effort to Top the Kaggle Leaderboard.

Keep up the great work.
Stay safe, stay healthy!

Lesson 6: Jobs in Computer Vision



6 Questions About Computer Vision Jobs

1. How do I break into the computer vision industry?

Computer vision and related fields in AI are so fast-moving. Find opportunities by staying on top of new developments.

We recommend a 3-part strategy to finding opportunities in computer vision.

1. **Stay updated on thought leadership.** Start every morning reading a 10-minute blog post on **Medium** or by scrolling your **Twitter feed**. Authors tend to link to larger research papers for further insights and learnings.
2. **See how computer vision fits into common image analysis and machine learning roles**, ranging from researcher to image analyst to software engineer. Do job searches based on skills (such as feature extraction or SLAM), rather than job title. You will find more opportunities.
3. **Create your own computer vision application and share it!** Job seekers ask Udacity Careers all the time on how to talk about their skills and experiences - better than talking is showing. Develop your own app or product (it can be a small, personal project!), put the work online (ie: upload your code on GitHub), and then write a blog post documenting the development process and purpose of the project. You can pursue both software and hardware projects.
    - Software ideas: Demonstrate basic image processing skills or proficiency with **OpenCV**. Run deep learning models with **AWS Deeplens**. You can also get inspired by what other people are doing - right now on Medium, many people are blogging about their **TensorFlow** projects.
    - Hardware ideas: **Raspberry Pi models** are small and affordable and Google released the **AIY Vision Kit** in 2017. Share what you create by writing a blog about the experience and become a part of the larger community by hashtagging #AIYProjects.

2. What are the key knowledge/skills that are useful to work on computer vision as an engineer?

In the Computer Vision Nanodegree program, you will gain insights into and practice these core computer vision skills:

- Implement image processing techniques like color and geometric transforms
- Create and apply convolutional filters to images to smooth them or detect object edges

- Apply feature extraction methods like Histogram of Oriented Gradients (HOG) and ORB to detect features like the eyes and mouth on a face
- Implement k-means clustering and contour detection to find areas with similar traits in an image
- Define and train a convolutional neural network (CNN) to classify different items of clothing
- Build a CNN that aims to solve a regression task: detecting facial keypoints; this skill can be extended to the task of emotion recognition
- Explore how region-based CNNs and formulations like YOLO lead to faster object recognition
- Define and train a recurrent neural network (RNN) to process and generate sequences of text
- Use long short-term memory cells to create an RNN that can produce captions given an image feature vector
- Train your own algorithmic models to predict and understand visual scenes
- Use methods like optical flow and Bayesian filters to track an object's motion
- Use SLAM to simultaneously localize an autonomous vehicle and create a map of the landmarks that surround it using only sensor measurements, collected over time

To become a better job candidate, dive deeper into the follow concepts, beyond the Nanodegree program.

- Knowledge of or applications in deep learning and neural networks [learning resource - not Udacity]
- Programming competency in C++ (a useful language for working with hardware). Find C++ programming lessons in the Extracurricular section of your Nanodegree Classroom!
- Simultaneous Location and Mapping (SLAM)
- PyTorch and/or TensorFlow

Although the above is by no means an exhaustive list, taking the time to pick up any of these these technologies will only strengthen your chances of moving forward in an application process.

3. Could I apply for deep learning jobs?

Companies investing in cutting-edge technologies often look for candidates with both deep learning and computer vision knowledge. For example, **Athelas** uses computer vision with deep learning to count white blood cells.

A deep learning engineer may be asked to focus on designing and changing the architecture of a CNN or RNN so that it is optimized for training for a variety of tasks, such as voice recognition or generating fake images. Computer vision and deep learning engineers tend to focus on deep learning as it is applied to feature extraction and object recognition with a CNN; some tasks and skills that fall in both realms are pre-processing image data, and training a CNN to detect meaningful visual features. A computer vision & deep learning engineer is expected to work with visual data like images, video, and LiDAR point clouds (laser data).

4. What are some job titles in computer vision?

Tip: Remember that there are so many roles available related to computer vision. Instead of searching for job titles, consider finding top AI startups and companies and looking at their overall jobs page.

Many companies, when advertising for roles requiring computer vision skills, use substitute terms. You may find these terms in job postings, rather than "computer vision" itself. Remember to read the job listing carefully. Common key terms akin to computer vision are:

- Image Recognition
- 3D Image Analysis
- Image Processing
- Deep Learning

Here are **common titles** of jobs in computer vision:

- Machine Learning Engineer
- Image Processing Engineer
- Computer Vision Engineer
- Researcher, Computer Vision
- Machine Learning Researcher, Computer Vision
- Applied Research Scientist
- AI Vision Image Processing Engineer
- Software Engineer, Vision
- Data Scientist
- Deep Learning Engineer - Computer Vision

5. What do typical computer vision engineers do in their day-to-day?

Although you will find jobs, especially in Silicon Valley, labelled "computer vision engineer", you are most likely to find other roles (such as "applied researcher" or "data scientist") that ask for computer vision skills. Do **Boolean searches** that combine some of your background with computer vision skills. Many smaller companies not yet established in the AI field may be looking for a generalist first, to pave the way for their data team.

Some common work duties you will find in job postings are:

- Design and build machine intelligence features
- Develop machine learning algorithms related to computer vision, such as object detection/recognition and image retrieval

- Deploy analytics models in production and evaluating their scalability
- Code in C++ and Python
- Collaborate with other data and engineering teams on hardware, software architecture hardware and quality assurance
- Improve deep-learning-based image segmentation and object detection pipelines
- Identify core requirements for camera sensors

6. Okay, I know all of this now. What should I do next?

Right now, you are learning the foundational skills for a career in computer vision. Focus on your learning, and imagine yourself in different roles. We recommend performing a **job search** to see what's out there.

We'll help you get started. We searched for "computer vision" jobs worldwide on LinkedIn. Click the button below to see the search results. Change your search terms and apply filters. Once you have an idea of which jobs or companies you may be interested in, read about the companies or find personal blogs by people working in those roles. You can even reach out to people for **informational interviews**.

Go to LinkedIn Search

After you spend some time looking at various jobs available to you, come back to the Classroom to continue.

Real World Applications of Computer Vision

Self-Driving Cars and Spatially Coherent Data

Computer vision is used for vehicle and pedestrian recognition and tracking (to determine their speed and predict movement). Check out **this blog** from David Silver on how computer vision works for self-driving cars.

Medical Image Analysis and Diagnosis

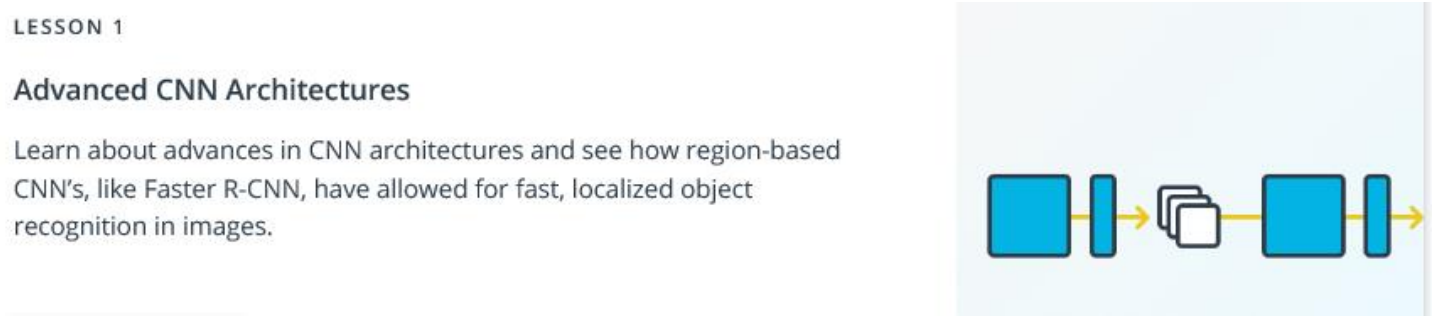
In addition to founding Udacity and starting the Google X and Self-Driving Car labs at Google, Sebastian Thrun is also a professor of computer science and an extensively published artificial intelligence scientist. In early 2017, his team at Stanford produced a report on how deep learning--which is tied intrinsically with computer vision--can be used to identify different types of skin cancers at an accuracy level comparable to that of dermatologists.

In the Extracurricular section of your classroom, you'll find a **course** taught by Sebastian himself that takes you through the research his team did to identify melanoma lesions, and what implications this research has for the future of medicine. The course material should take you between two and three hours, and at the conclusion, you'll have the opportunity to write an implementation of your own.

While you technically don't need to take the course to graduate, understanding and being able to discuss applications of computer vision in the real world can really impress a hiring manager during your interview or technical interview. It can also open the door to job opportunities in tech-adjacent fields--in this case, medical technology.

Advanced Computer Vision & Deep Learning

Lesson 1: Advanced CNN Architectures



CNN's and Scene Understanding

https://www.youtube.com/watch?time_continue=6&v=_iRqSOsTBQU&feature=emb_logo

More than Classification

https://www.youtube.com/watch?time_continue=4&v=vBE5KvvAYzg&feature=emb_logo

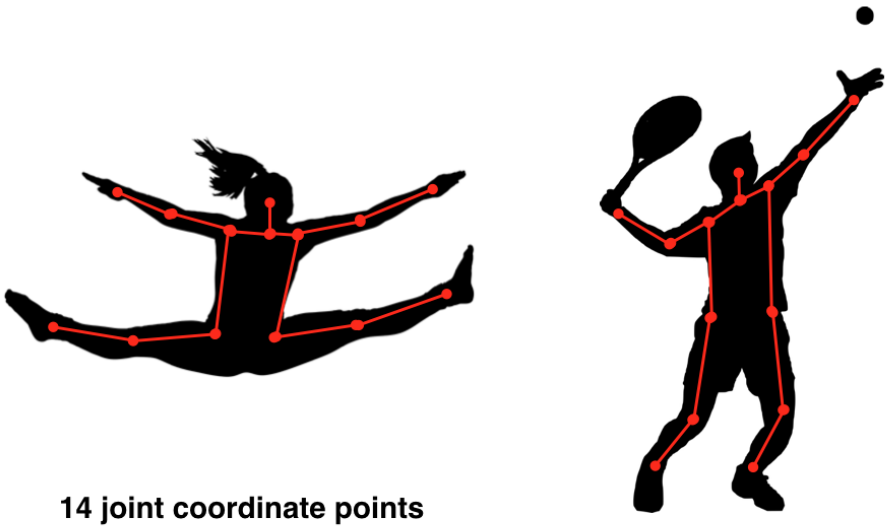
Classification and Localization

https://www.youtube.com/watch?time_continue=4&v=UqNg9d6cKQU&feature=emb_logo

Beyond Bounding & Regression

Beyond Bounding Boxes

To predict bounding boxes, we train a model to take an image as input and output coordinate values: (x, y, w, h). This kind of model can be extended to *any* problem that has coordinate values as outputs! One such example is **human pose estimation**.



[Huan pose estimation points.]

In the above example, we see that the pose of a human body can be estimated by tracking 14 points along the joints of a body.

Weighted Loss Functions

You may be wondering: how can we train a network with two different outputs (a class and a bounding box) and different losses for those outputs?

We know that, in this case, we use categorical cross entropy to calculate the loss for our predicted and true classes, and we use a regression loss (something like smooth L1 loss) to compare predicted and true bounding boxes. But, we have to train our whole network using one loss, so how can we combine these?

There are a couple of ways to train on multiple loss functions, and in practice, we often use a weighted sum of classification *and* regression losses (ex. 0.5*cross_entropy_loss + 0.5*L1_loss); the result is a single error value with which we can do backpropagation. This does introduce a hyperparameter: the loss weights. We want to weight each loss so that these losses are balanced and combined effectively, and in research we see that another regularization term is often introduced to help decide on the weight values that best combine these losses.

Look at the documentation for MSE Loss. For a ground truth coordinate `(2, 5)` and a predicted coordinate `(2.5, 3)`, what is the MSE loss between these points? (You may assume default values for averaging.)

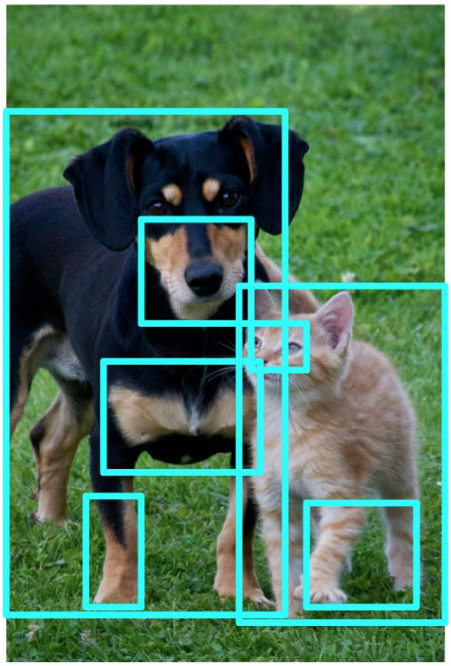
2.125 RESET

QUESTION 2 OF 2

Look at the documentation for Smooth L1 Loss. For a ground truth coordinate `(2, 5)` and a predicted coordinate `(2.5, 3)`, what is the smooth L1 loss between these points?

- 0.0125
- 0.55
- ✓ 0.8125

Region Proposals



[Suggested region proposals for an image of a cat and dog.]

Reflect

Consider the above image, how do you think you would select the best proposed regions; what criteria do good regions have?

Your reflection

should have some edges and a different pattern.

Things to think about

The regions we want to analyze are those with complete objects in them. We want to get rid of regions that contain image background or only a portion of an object. So, two common approaches are suggested: 1. identify similar regions using feature extraction or a clustering algorithm like k-means, as you've already seen; these methods should identify any areas of interest. 2. Add another layer to our model that performs a binary classification on these regions and labels them: object or not-object; this gives us the ability to discard any non-object regions!

https://www.youtube.com/watch?v=HLwpr7h3rPY&feature=emb_logo

R-CNN

https://www.youtube.com/watch?time_continue=3&v=EchapZJMTYU&feature=emb_logo

Fast R-CNN

RoI Pooling

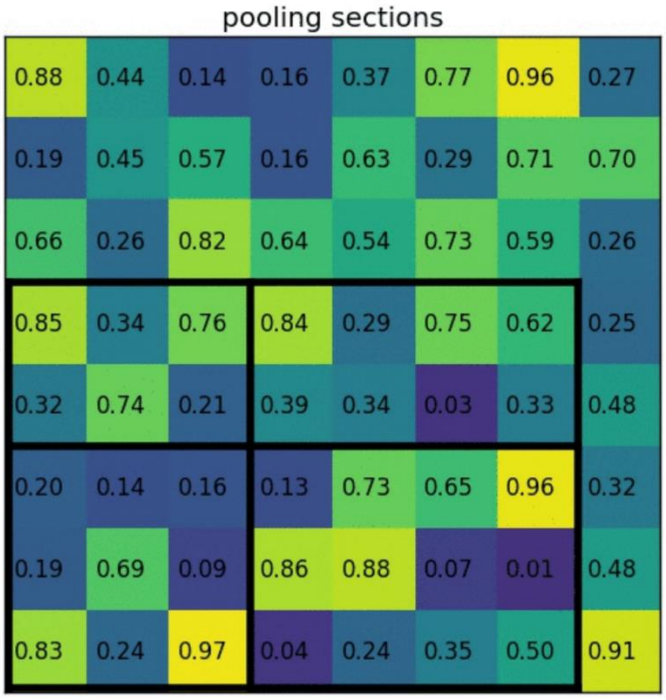
To warp regions of interest into a consistent size for further analysis, some networks use RoI pooling. RoI pooling is an additional layer in our network that takes in a rectangular region of any size, performs a maxpooling operation on that region in pieces such that the output is a fixed shape. Below is an example of a region with some pixel values being broken up into pieces which pooling will be applied to; a section with the values:

[[0.85, 0.34, 0.76],

[0.32, 0.74, 0.21]]

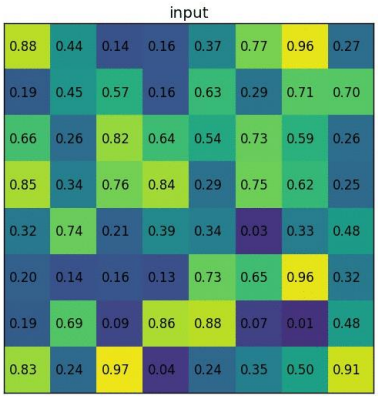
Will become a single max value after pooling: 0.85. After applying this to an image in these pieces, you can see how any rectangular region can be forced into a smaller, square representation.

https://www.youtube.com/watch?v=6FOBZ9OgWlY&feature=emb_logo



An example of pooling sections, credit to this informational resource on RoI pooling [by Tomasz Grel].

You can see the complete process from input image to region to reduced, maxpooled region, below.



Credit to this informational resource on RoI pooling.

Speed

Fast R-CNN is about 10 times as fast to train as an R-CNN because it only creates convolutional layers once for a given image and then performs further analysis on the layer. Fast R-CNN also takes a shorter time to test on a new image! It's test time is dominated by the time it takes to create region proposals.

Region Proposal Network

You may be wondering: how exactly are the RoI's generated in the region proposal portion of the Faster R-CNN architecture?

The region proposal network (RPN) works in Faster R-CNN in a way that is similar to YOLO object detection, which you'll learn about in the next lesson. The RPN looks at the output of the last convolutional layer, a produced feature map, and takes a sliding window approach to possible-object detection. It slides a small (typically 3x3) window over the feature map, then for *each* window the RPN:

1. Uses a set of defined anchor boxes, which are boxes of a defined aspect ratio (wide and short or tall and thin, for example) to generate multiple possible RoI's, each of these is considered a region proposal.
2. For each proposal, this network produces a probability, Pc, that classifies the region as an object (or not) and a set of bounding box coordinates for that object.
3. Regions with too low a probability of being an object, say Pc < 0.5, are discarded.

Training the Region Proposal Network

Since, in this case, there are no ground truth regions, how do you train the region proposal network?

The idea is, for any region, you can check to see if it overlaps with any of the ground truth objects. That is, for a region, if we classify that region as an object or not-object, which class will it fall into? For a region proposal that does cover some portion of an object, we should say that there is a high probability that this region has an object init and that region should be kept; if the likelihood of an object being in a region is too low, that region should be discarded.

I'd recommend **this blog post** if you'd like to learn more about region selection.

Speed Bottleneck

Now, for all of these networks *including* Faster R-CNN, we've aimed to improve the speed of our object detection models by reducing the time it takes to generate and decide on region proposals. You might be wondering: is there a way to get rid of this proposal step entirely? And in the next section we'll see a method that does not rely on region proposals to work!

https://www.youtube.com/watch?v=ySh_Q3KTTBY&feature=emb_logo

Faster R-CNN Implementation

If you'd like to look at an implementation of this network in code, you can find a peer-reviewed version, at **this Github repo**.

Detection With and Without Proposals

https://www.youtube.com/watch?time_continue=8&v=IMnt1HFu_nc&feature=emb_logo

Lesson 2: YOLO

YOLO

Learn about the YOLO (You Only Look Once) multi-object detection model and work with a YOLO implementation.

Introduction: https://www.youtube.com/watch?time_continue=1&v=uyefSrHZesY&feature=emb_logo

YOLO Output

https://www.youtube.com/watch?v=MyOuuwk0qC4&feature=emb_logo

Sliding Windows, Revisited

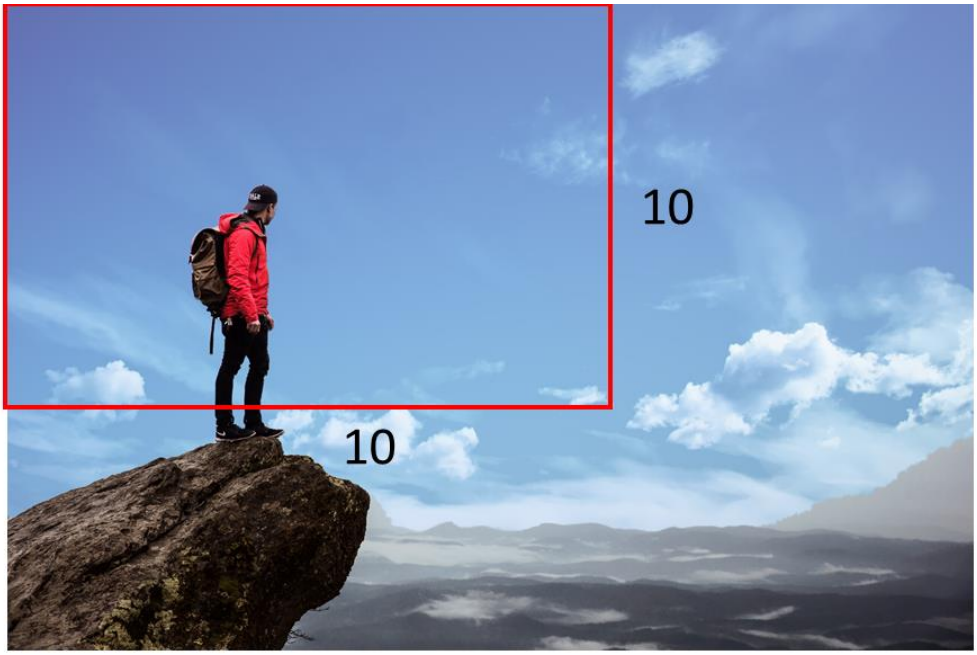
https://www.youtube.com/watch?time_continue=6&v=8qYqqibIz90&feature=emb_logo

A Convolutional Approach to Sliding Windows

Let's assume we have a 16 x 16 x 3 image, like the one shown below. This means the image has a size of 16 by 16 pixels and has 3 channels, corresponding to RGB.

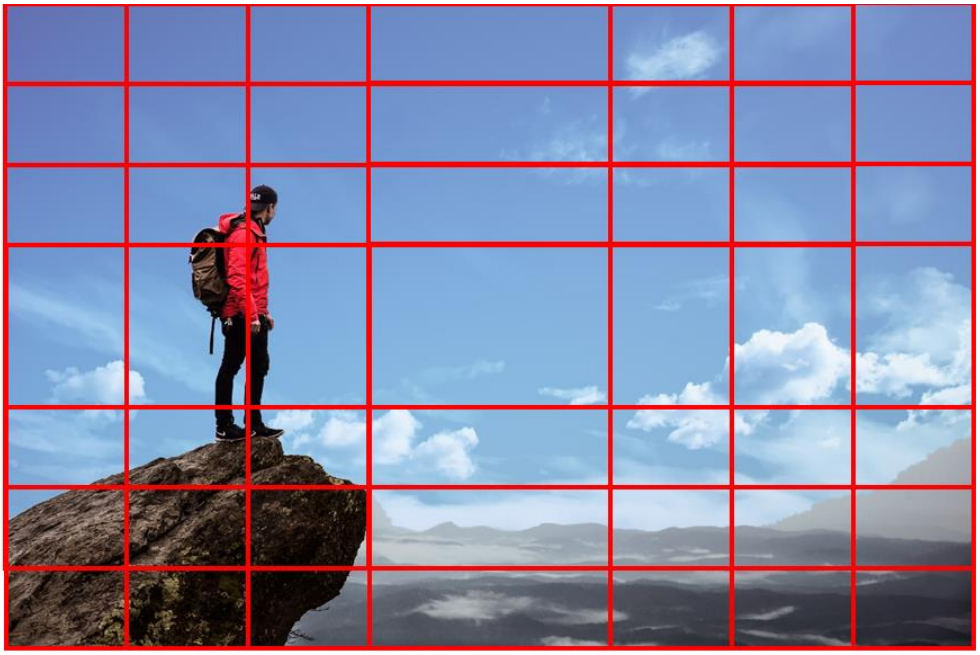
16 x 16 x 3

Let’s now select a window size of 10 x 10 pixels as shown below:



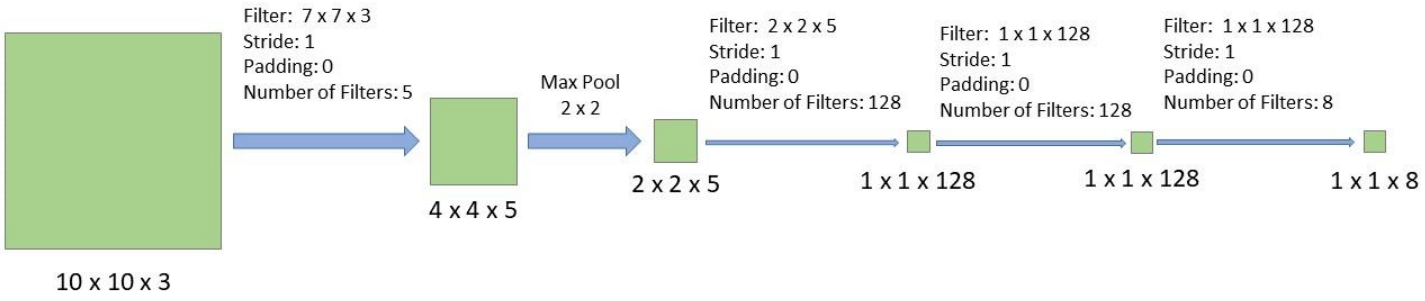
16 x 16 x 3

If we use a stride of 2 pixels, it will take 16 windows to cover the entire image, as we can see below.



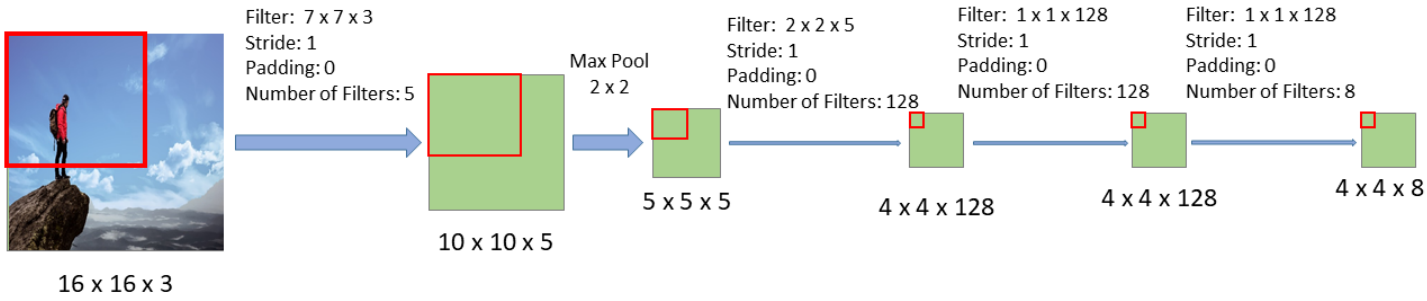
16 x 16 x 3

In the original Sliding Windows approach, each of these 16 windows will have to be passed individually through a CNN. Let's assume that CNN has the following architecture:



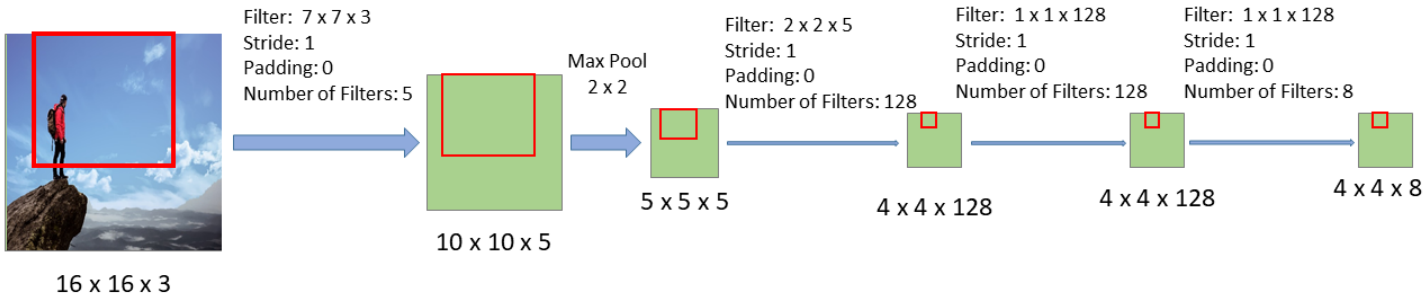
The CNN takes as input a 10 x 10 x 3 image, then it applies 5, 7 x 7 x 3 filters, then it uses a 2 x 2 Max pooling layer, then is has 128, 2 x 2 x 5 filters, then is has 128, 1 x 1 x 128 filters, and finally it has 8, 1 x 1 x 128 filters that represents a softmax output.

What will happen if we change the input of the above CNN from 10 x 10 x 3, to 16 x 16 x 3? The result is shown below:

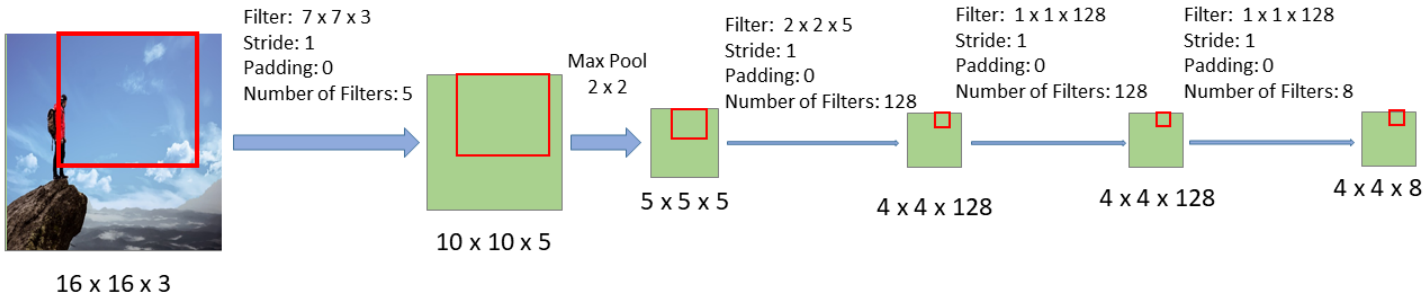


As we can see, this CNN architecture is the same as the one shown before except that it takes as input a 16 x 16 x 3 image. The sizes of each layer change because the input image is larger, but the same filters as before have been applied.

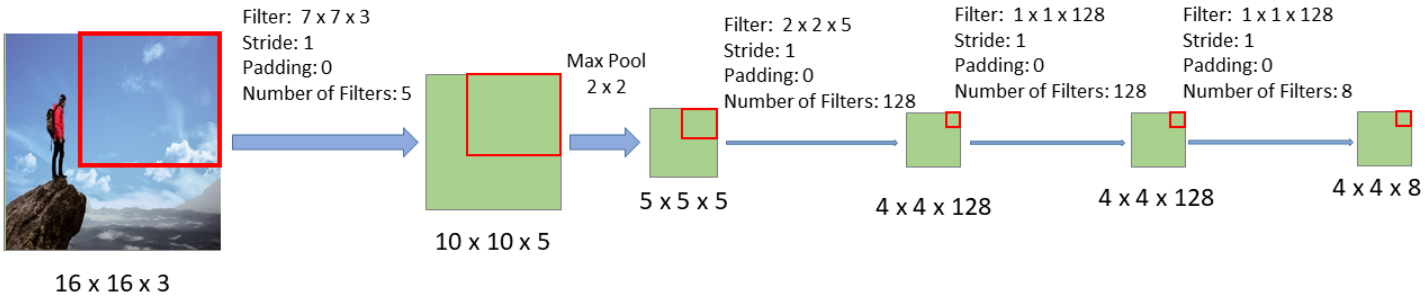
If we follow the region of the image that corresponds to the first window through this new CNN, we see that the result is the upper-left corner of the last layer (*see image above*). Similarly, if we follow the section of the image that corresponds to the second window through this new CNN, we see the corresponding result in the last layer:



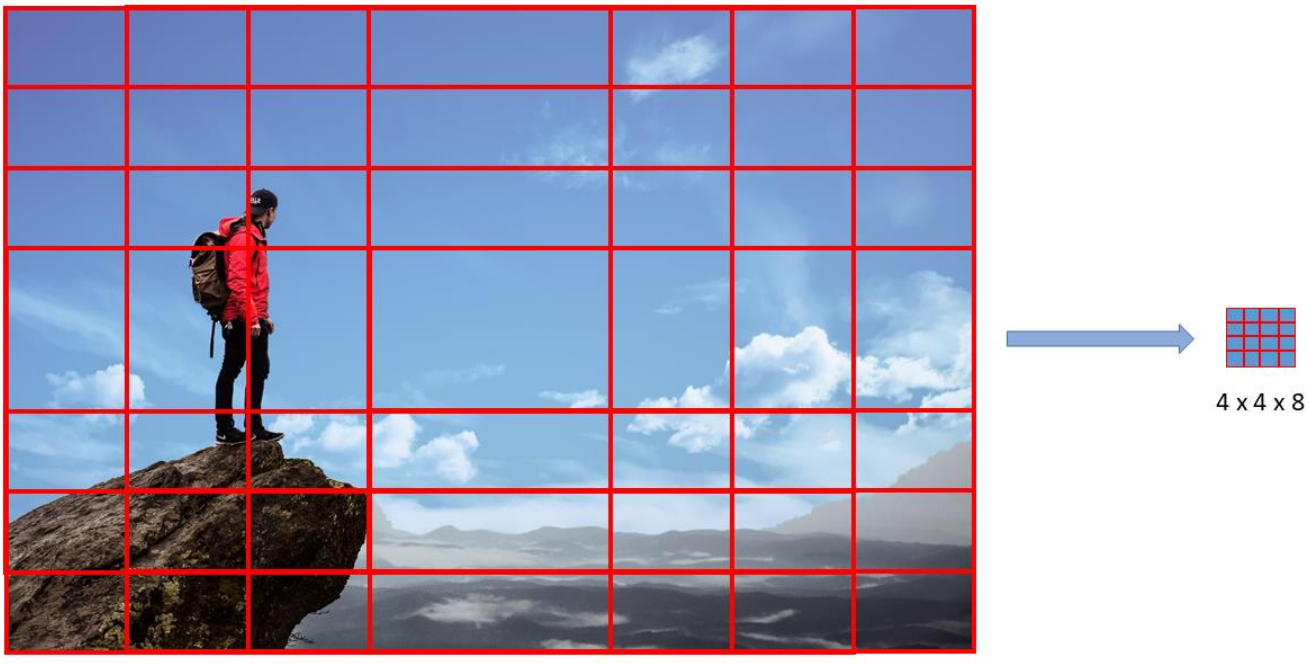
Likewise, if we follow the section of the image that corresponds to the third window through this new CNN, we see the corresponding result in the last layer, as shown in the image below:



Finally, if we follow the section of the image that corresponds to the fourth window through this new CNN, we see the corresponding result in the last layer, as shown in the image below:



In fact, if we follow all the windows through the CNN we see that all the 16 windows are contained within the last layer of this new CNN. Therefore, passing the 16 windows individually through the old CNN is exactly the same as passing the whole image only once through this new CNN.



This is how you can apply sliding windows with a CNN. This technique makes the whole process much more efficient. However, this technique has a downside: the position of the bounding boxes is not going to be very accurate. The reason is that it is quite unlikely that a given size window and stride will be able to match the objects in the images perfectly. In order to increase the accuracy of the bounding boxes, YOLO uses a grid instead of sliding windows, in addition to two other techniques, known as Intersection Over Union and Non-Maximal Suppression.

The combination of the above techniques is part of the reason the YOLO algorithm works so well. Before diving into how YOLO puts all these techniques together, we will look first at each technique individually.

Using a Grid

https://www.youtube.com/watch?time_continue=12&v=OmgR35Go79Y&feature=emb_logo

Training on a Grid

https://www.youtube.com/watch?time_continue=4&v=uhefpakvXh8&feature=emb_logo

Generating Bounding Boxes

https://www.youtube.com/watch?time_continue=10&v=TGfPX-XcyOs&feature=emb_logo

What do you think will happen once a network, like the CNN we've been talking about, sees a new, test image with an object in it? First, our network has to break this new, test image into a grid.

An image of a woman working broken into a grid of cells; 5 of which are labelled: A, B, C, D, and E.

QUIZ QUESTION

For the above image, which cell do you think will predict the bounding box for the person in that image? (You may select multiple cells if you think more than one will detect the person and produce a bounding box.)

A

Too Many Boxes

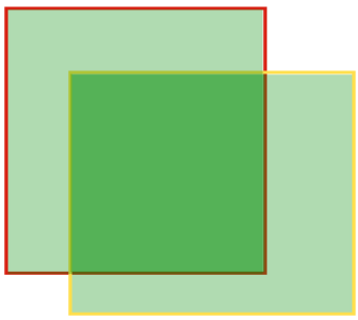
https://www.youtube.com/watch?time_continue=3&v=nYDWsFdFnQ8&feature=emb_logo

Intersection over Union (IoU)

https://www.youtube.com/watch?time_continue=3&v=ieKEHlEjIsY&feature=emb_logo

Intersection over Union (IoU)

We know the IoU is given by the area of: intersection/union. The next couple questions will test your intuition about what values IoU can take as it compares two bounding boxes.



Two bounding boxes. Intersection in dark green and union in light green.

QUESTION 1 OF 2

Imagine you're comparing a ground truth box to a predicted box. If you want your predicted box to be as close to this ground truth box as possible, what would you want the IoU to be?

- 0
- 0.5
- ✓ 1

QUESTION 2 OF 2

What is the lowest value IoU can have?

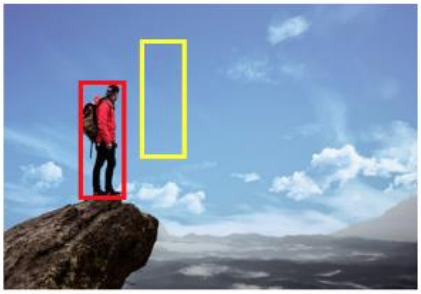
- -2
- -1
- ✓ 0

IoU Values

The IoU between two bounding boxes will always have a value between 0 and 1 because, as two boxes drift apart, their intersection approaches 0, but if two bounding boxes overlap *perfectly* their IoU will be 1. So, the higher the IOU the more overlap there is between the two bounding boxes!

In the next video we will see how Non-Maximal suppression uses the IOU to only choose the best bounding box.

$$IOU = \frac{2000}{2000} = 1$$



$$IOU = \frac{0}{4000} = 0$$

Examples of maximum and minimum IoU values between two boxes.

Non-Maximal Suppression

https://www.youtube.com/watch?time_continue=24&v=TE6M29Jo9hk&feature=emb_logo

Anchor Boxes

https://www.youtube.com/watch?time_continue=3&v=IzILYgVb76g&feature=emb_logo

YOLO Algorithm

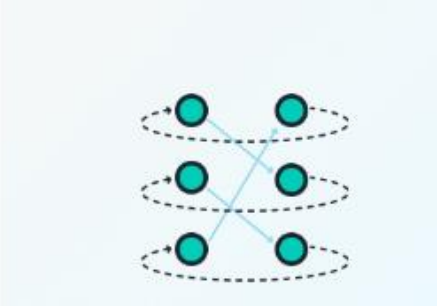
https://www.youtube.com/watch?time_continue=1&v=ZbQzCHQ8YEo&feature=emb_logo

- Notebook: YOLO Implementation

Lesson 3: RNN's

RNN's

Explore how memory can be incorporated into a deep learning model using recurrent neural networks (RNNs). Learn how RNNs can learn from and generate ordered sequences of data.



Recurrent Neural Networks

So far, we've been looking at convolutional neural networks and models that allows us to analyze the spatial information in a given input image. CNN's excel in tasks that rely on finding spatial and visible patterns in training data.

In this and the next couple lessons, we'll be reviewing RNN's or recurrent neural networks. These networks give us a way to incorporate **memory** into our neural networks, and will be critical in analyzing sequential data. RNN's are most often associated with text processing and text generation because of the way sentences are structured as a sequence of words, but they are also useful in a number of computer vision applications, as well!

RNN's in Computer Vision

At the end of this lesson, you will be tasked with creating an automatic image captioning model that takes in an image as input and outputs a *sequence* of words, describing that image. Image captions are used to create accessible content and in a number of other cases where one may want to read about the contents of an image. This model will include a CNN component for finding spatial patterns in the input image *and* and RNN component that will be responsible for generative descriptive text!

RNN's are also sometimes used to analyze sequences of images; this can be useful in captioning video, as well as video classification, gesture recognition, and object tracking; all of these tasks see as input a *sequence* of image frames.

Sketch RNN

One of my favorite use cases for RNN's in computer vision tasks is in generating drawings. **Sketch RNN (demo here)** is a program that learns to complete a drawing, once you give it something (a line or circle, etc.) to start!

Sketch RNN example output. Left, Mona Lisa. Right, pineapple.

It's interesting to think of drawing as a sequential act, but it is! This network takes a starting line or squiggle and then, having trained on a number of types of sketches, does it's best to complete the drawing based on your input squiggle.

Next, you'll learn all about how RNN's are structured and how they can be trained! This section is taught by Ortal, who has a PhD in Computer Engineering and has been a professor and researcher in the fields of applied cryptography and embedded systems.

Supporting Materials

Video classification methods

RNN Introduction

Hi! I am Ortal, your instructor for this lesson!

In this lesson we will learn about **Recurrent Neural Networks (RNNs)**.

The neural network architectures you've seen so far were trained using the current inputs only. We did not consider previous inputs when generating the current output. In other words, our systems did not have any **memory** elements. RNNs address this very basic and important issue by using **memory** (i.e. past inputs to the network) when producing the current output.

https://www.youtube.com/watch?v=AIQEqg6F38A&feature=emb_logo

A bit of history

How did the theory behind RNN evolve? Where were we a few years ago and where are we now?

As mentioned in this video, RNNs have a key flaw, as capturing relationships that span more than 8 or 10 steps back is practically impossible. This flaw stems from the "**vanishing gradient**" problem in which the contribution of information decays geometrically over time.

What does this mean?

As you may recall, while training our network we use **backpropagation**. In the backpropagation process we adjust our weight matrices with the use of a **gradient**. In the process, gradients are calculated by continuous multiplications of derivatives. The value of these derivatives may be so small, that these continuous multiplications may cause the gradient to practically "vanish".

LSTM is one option to overcome the Vanishing Gradient problem in RNNs.

Please use these resources if you would like to read more about the **Vanishing Gradient** problem or understand further the concept of a **Geometric Series** and how its values may exponentially decrease.

If you are still curious, for more information on the important milestones mentioned here, please take a peek at the following links:

- **TDNN**
- Here is the original **Elman Network** publication from 1990. This link is provided here as it's a significant milestone in the world on RNNs. To simplify things a bit, you can take a look at the following **additional info**.
- In this **LSTM** link you will find the original paper written by **Sepp Hochreiter** and **Jürgen Schmidhuber**. Don't get into all the details just yet. We will cover all of this later!

As mentioned in the video, Long Short-Term Memory Cells (LSTMs) and Gated Recurrent Units (GRUs) give a solution to the vanishing gradient problem, by helping us apply networks that have temporal dependencies. In this lesson we will focus on RNNs and continue with LSTMs. We will not be focusing on GRUs. More information about GRUs can be found in the following **blog**. Focus on the overview titled: **GRUs**.

https://www.youtube.com/watch?v=HbxAnYUfRnc&feature=emb_logo

Applications

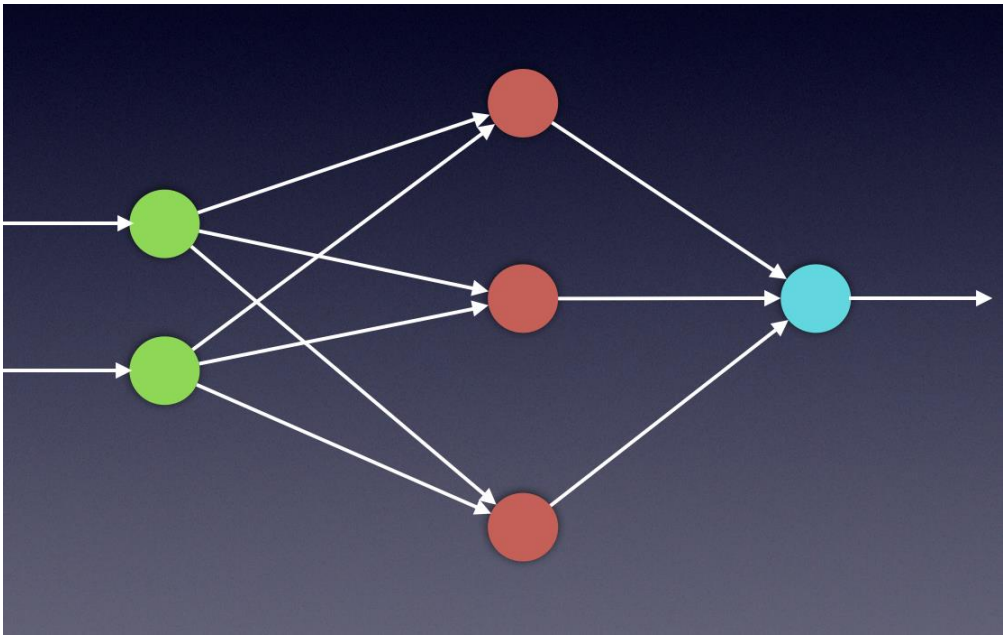
https://www.youtube.com/watch?v=6JbTNARuKII&feature=emb_logo

The world's leading tech companies are all using RNNs, particularly LSTMs, in their applications. Let's take a look at a few.

There are so many interesting applications, let's look at a few more!

- Are you into gaming and bots? Check out the **DotA 2 bot by Open AI**
- How about **automatically adding sounds to silent movies?**
- Here is a cool tool for **automatic handwriting generation**
- Amazon's voice to text using high quality speech recognition, **Amazon Lex**.
- Facebook uses RNN and LSTM technologies for **building language models**
- Netflix also uses RNN models - **here is an interesting read**

Feedforward Neural Network - A Reminder



The mathematical calculations needed for training RNN systems are fascinating. To deeply understand the process, we first need to feel confident with the vanilla FFNN system. We need to thoroughly understand the feedforward process, as well as the backpropagation process used in the training phases of such systems. The next few videos will cover these topics, which you are already familiar with. We will address the feedforward process as well as backpropagation, using specific examples. These examples will serve as extra content to help further understand RNNs later in this lesson.

The following couple of videos will give you a brief overview of the **Feedforward Neural Network (FFNN)**.

https://www.youtube.com/watch?v=_vrp2lZjXf0&feature=emb_logo

OK, you can take a small break now. We will continue with FFNN when you come back!
As mentioned before, when working with neural networks we have 2 primary phases:

https://www.youtube.com/watch?v=FfPjaGcZODc&feature=emb_logo

Training
and

Evaluation.
During the **training** phase, we take the data set (also called the *training set*), which includes many pairs of inputs and their corresponding targets (outputs). Our goal is to find a set of weights that would best map the inputs to the desired outputs. In the **evaluation** phase, we use the network that was created in the training phase, apply our new inputs and expect to obtain the desired outputs.
The training phase will include two steps:

Feedforward
and

Backpropagation
We will repeat these steps as many times as we need until we decide that our system has reached the best set of weights, giving us the best possible outputs.

The next two videos will focus on the feedforward process.
You will notice that in these videos I use subscripts as well as superscript as a numeric notation for the weight matrix.

For example:

- W_k is weight matrix k
- W_{ij}^k is the ij element of weight matrix k

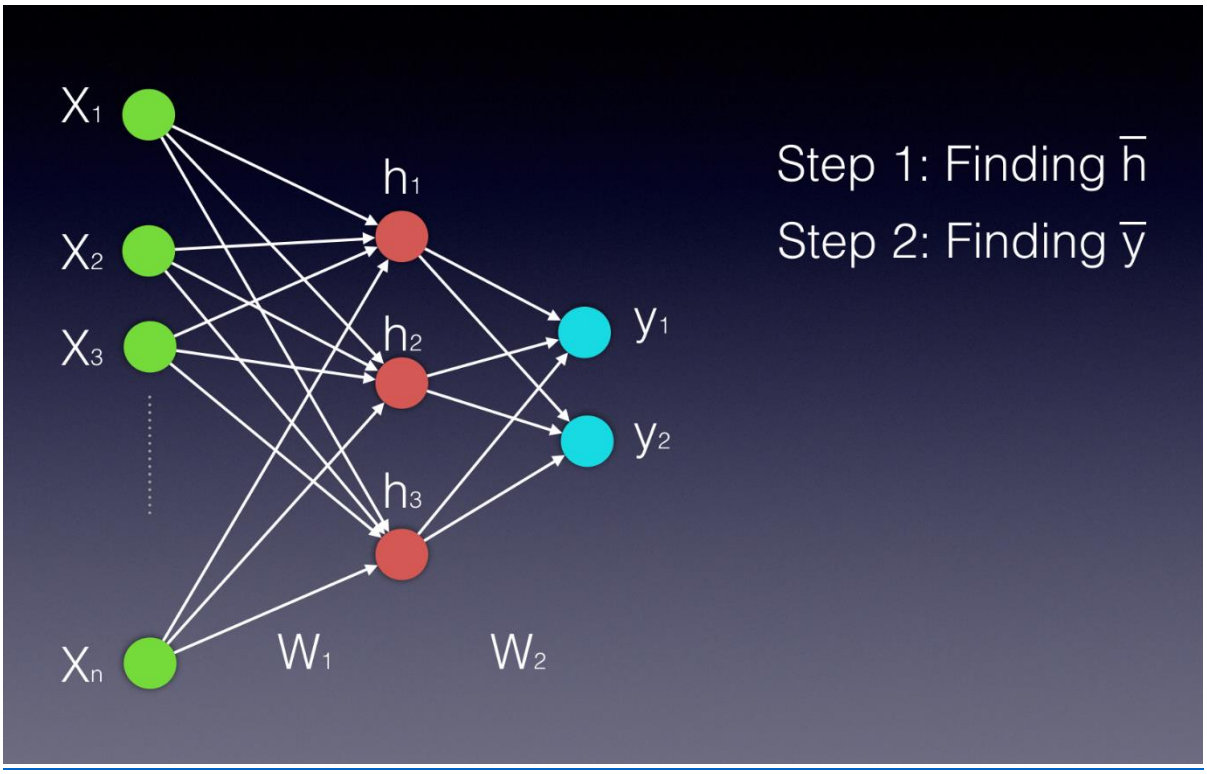
Feedforward

In this section we will look closely at the math behind the feedforward process. With the use of basic Linear Algebra tools, these calculations are pretty simple!

If you are not feeling confident with linear combinations and matrix multiplications, you can use the following links as a refresher:

- Linear Combination
- Matrix Multiplication

Assuming that we have a single hidden layer, we will need two steps in our calculations. The first will be calculating the value of the hidden states and the latter will be calculating the value of the outputs.



Notice that both the hidden layer and the output layer are displayed as vectors, as they are both represented by more than a single neuron.

Our first video will help you understand the first step- **Calculating the value of the hidden states**.

https://www.youtube.com/watch?v=4rCfnWbx8-0&feature=emb_logo

As you saw in the video above, vector h' of the hidden layer will be calculated by multiplying the input vector with the weight matrix W1W^{1}W1 the following way:

h’¯=(x¯W1)\bar{h'} = (\bar{x} W^1)h’¯=(x¯W1)

Using vector by matrix multiplication, we can look at this computation the following way:

$$\begin{bmatrix} h'_1 & h'_2 & h'_3 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 & ..... & x_n \end{bmatrix} \cdot \begin{bmatrix} W_{11} & W_{12} & W_{13} \\ W_{21} & W_{22} & W_{23} \\ \vdots & & \\ W_{n1} & W_{n2} & W_{n3} \end{bmatrix}$$

Equation 1

After finding h'h'h' , we need an activation function (Φ \PhiΦ) to finalize the computation of the hidden layer's values. This activation function can be a Hyperbolic Tangent, a Sigmoid or a ReLU function. We can use the following two equations to express the final hidden vector h¯ \bar{h}h¯:

h¯=Φ(x¯W1)\bar{h} = \Phi(\bar{x} W^1)h¯=Φ(x¯W1)

or

h¯=Φ(h') \bar{h} = \Phi(h')h¯=Φ(h')

Since WijW_{ij}Wijrepresents the weight component in the weight matrix, connecting neuron **i** from the input to neuron **j** in the hidden layer, we can also write these calculations in the following way: (notice that in this example we have *n* inputs and only 3 hidden neurons)

$$\begin{aligned} h_1 &= \Phi(x_1 W_{11} + x_2 W_{21} + ... + x_n W_{n1}) \\ h_2 &= \Phi(x_1 W_{12} + x_2 W_{22} + ... + x_n W_{n2}) \\ h_3 &= \Phi(x_1 W_{13} + x_2 W_{23} + ... + x_n W_{n3}) \end{aligned}$$

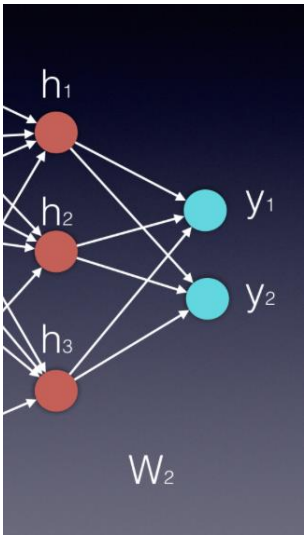
Equation 2

More information on the activation functions and how to use them can be found here

This next video will help you understand the second step- **Calculating the values of the Outputs**.

https://www.youtube.com/watch?v=kTYbTVh1d0k&feature=emb_logo

As you've seen in the video above, the process of calculating the output vector is mathematically similar to that of calculating the vector of the hidden layer. We use, again, a vector by matrix multiplication, which can be followed by an activation function. The vector is the newly calculated hidden layer and the matrix is the one connecting the hidden layer to the output.



Essentially, each new layer in an neural network is calculated by a vector by matrix multiplication, where the vector represents the inputs to the new layer and the matrix is the one connecting these new inputs to the next layer.

In our example, the input vector is h¯\bar{h}h¯ and the matrix is W2W^2W2, therefore y¯=h¯W2\bar{y}=\bar{h}W^2y¯ =h¯W2. In some applications it can be beneficial to use a softmax function (if we want all output values to be between zero and 1, and their sum to be 1).

$$\begin{bmatrix} y_1 & y_2 \end{bmatrix} = \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$$

Equation 3

The two error functions that are most commonly used are the Mean Squared Error (MSE) (usually used in regression problems) and the cross entropy (usually used in classification problems).

In the above calculations we used a variation of the MSE.

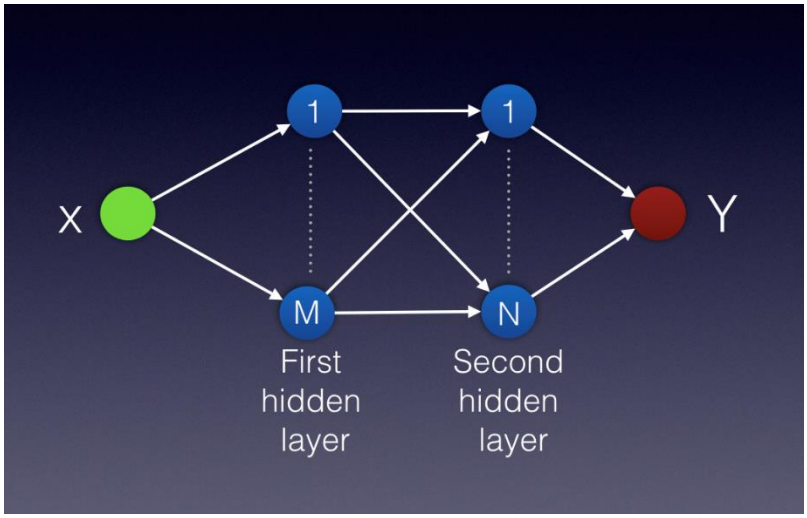
The next few videos will focus on the backpropagation process, or what we also call stochastic gradient decent with the use of the chain rule.

The following picture is of a feedforward network with

- A single input x
- Two hidden layers
- A singe output

The first hidden layer has M neurons.

The second hidden layer has N neurons.



QUIZ QUESTION

What is the total number of multiplication operations needed for a single feedforward pass?

- MN
- M+N
- M+N+2MN
- ✓ M+N+NM

Solution

To calculate the number of multiplications needed for a single feedforward pass, we can break down the network to three steps:

- Step 1: From the single input to the first hidden layer
- Step 2: From the first hidden layer to the second hidden layer
- Step 2: From the second hidden layer to the single output

Step 1

The single input is multiplied by a vector with M values. Each value in the vector will represent a weight connecting the input to the first hidden layer. Therefore, we will have **M** multiplication operations.

Step 2

Each value in the first hidden layer (M in total) will be multiplied by a vector with N values. Each value in the vector will represent a weight connecting the neurons in the first hidden layer to the neurons in the second hidden layer. Therefore, we will have here M times N calculations, or simply **MN** multiplication operations.

Step 3

Each value in the second hidden layer (N in total) will be multiplied once, by the weight element connecting it to the single output. Therefore, we will have **N** multiplication operations.
In total, we will add the number of operations we calculated in each step: **M+MN+N** .

Backpropagation Theory

Since partial derivatives are the key mathematical concept used in backpropagation, it's important that you feel confident in your ability to calculate them. Once you know how to calculate basic derivatives, calculating partial derivatives is easy to understand.
For more information on partial derivatives use the following link

For calculation purposes in future quizzes of the lesson, you can use the following link as a reference for common derivatives.

In the **backpropagation** process we minimize the network error slightly with each iteration, by adjusting the weights. The following video will help you understand the mathematical process we use for computing these adjustments.

https://www.youtube.com/watch?time_continue=1&v=Xlgd8I3TWUg&feature=emb_logo

If we look at an arbitrary layer k, we can define the amount by which we change the weights from neuron i to neuron j stemming from layer k as: ΔWk\Delta W^kΔWkij_{ij}ij.

The superscript (*k*) indicates that the weight connects layer *k* to layer *k+1*.

Therefore, the weight update rule for that neuron can be expressed as:

Wnew=Wprevious+ΔWkW_{new} = W_{previous} +\Delta W^kWnew=Wprevious+ΔWkij_{ij}ij

Equation 4

The updated value ΔWijk\Delta W_{ij}^kΔWijk is calculated through the use of the gradient calculation, in the following way:

ΔWijk=α(−∂E∂W)\Delta W_{ij}^k=\alpha (-\frac{\partial E}{\partial W})ΔWijk=α(−∂W∂E), where α\alphaα is a small positive number called the **Learning Rate**.

Equation 5

From these derivation we can easily see that the weight updates are calculated the by the following equation:

Wnew=Wprevious+α(−∂E∂W)W_{new}= W_{previous} +\alpha (-\frac{\partial E}{\partial W})Wnew=Wprevious +α(−∂W∂E)

Equation 6

Since many weights determine the network's output, we can use a vector of the partial derivatives (defined by the Greek letter Nabla ∇\nabla∇) of the network error - each with respect to a different weight.

Wnew=Wprevious+α∇W(−E)W_{new}= W_{previous}+\alpha \nabla_W(-E)Wnew=Wprevious+α∇W(−E)

Equation 7

Here you can find other good resources for understanding and tuning the Learning Rate:

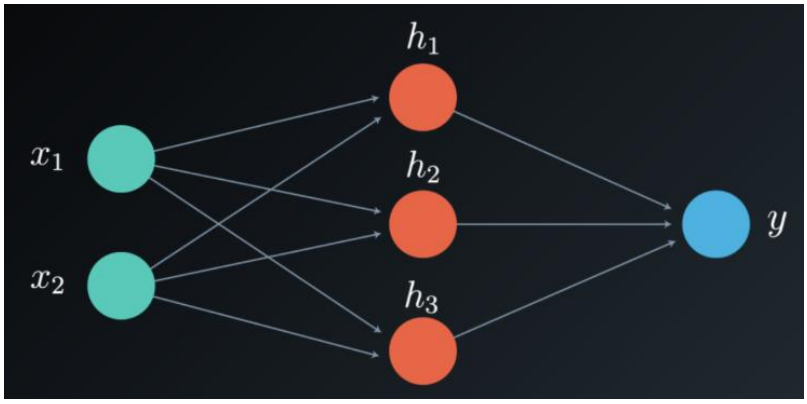
- resource 1
- resource 2

The following video is given as a refresher to **overfitting** . You have already seen this concept in the *Training Neural Networks* lesson. Feel free to skip it and jump right into the next video.

https://www.youtube.com/watch?v=rmBLnVbFfFY&feature=emb_logo

Backpropagation- Example (part a)

We will now continue with an example focusing on the backpropagation process, and consider a network having two inputs [x1,x2][x_1, x_2][x1,x2], three neurons in a single hidden layer [h1,h2,h3][h_1, h_2, h_3][h1,h2,h3] and a single output yyy.



The weight matrices to update are W1W^1W1 from the input to the hidden layer, and W2W^2 W2 from the hidden layer to the output. Notice that in our case W2W^2 W2 is a vector, not a matrix, as we only have one output.

https://www.youtube.com/watch?v=3k72z_WaeXg&feature=emb_logo

The chain of thought in the weight updating process is as follows:
To update the weights, we need the network error. To find the network error, we need the network output, and to find the network output we need the value of the hidden layer, vector h¯\bar {h}h¯.
h¯=[h1,h2,h3]\bar{h}=[h_1, h_2, h_3]h¯=[h1,h2,h3]
Equation 8

Each element of vector h¯\bar {h}h¯ is calculated by a simple linear combination of the input vector with its corresponding weight matrix W1W^1W1, followed by an activation function.

$$h_1 = \Phi(\sum_i^2 (x_i W_{i1}))$$
$$h_2 = \Phi(\sum_i^2 (x_i W_{i2}))$$
$$h_3 = \Phi(\sum_i^2 (x_i W_{i3}))$$

Equation 9

We now need to find the network's output, yyy. yyy is calculated in a similar way by using a linear combination of the vector h¯\bar{h}h¯ with its corresponding elements of the weight vector W2W^2W2.

$$y = \sum_{i}^{3} (h_i W_i)$$

Equation 10

After computing the output, we can finally find the network error.
As a reminder, the two Error functions most commonly used are the Mean Squared Error (MSE) (usually used in regression problems) and the cross entropy (often used in classification problems).
In this example, we use a variation of the MSE:
E=(d−y)22E=\frac{(d-y)^2}{2} E=2(d−y)2,
where dd d is the desired output and yy y is the calculated one. Notice that **y** and **d** are not vectors in this case, as we have a single output.
The error is their squared difference, E=(d−y)2E=(d-y)^2E=(d−y)2, and is also called the network's **Loss Function**. We are dividing the error term by 2 to simplify notation, as will become clear soon.

The aim of the backpropagation process is to minimize the error, which in our case is the Loss Function. To do that we need to calculate its partial derivative with respect to all of the weights.
Since we just found the output y, we can now minimize the error by finding the updated values ΔWijk\Delta W_{ij}^kΔWijk. The superscript k indicates that we need to update each and every layer k.

As we noted before, the weight update value ΔWijk\Delta W_{ij}^kΔWijk is calculated with the use of the gradient the following way:
ΔWijk=α(−∂E∂W)\Delta W_{ij}^k=\alpha (-\frac{\partial E}{\partial W})ΔWijk=α(−∂W∂E)
Therefore:
ΔWijk=α(−∂E∂W)=−α2∂(d−y)2∂Wij=−2α2(d−y)∂(d−y)∂Wij\Delta W_{ij}^k=\alpha (-\frac{\partial E}{\partial W})=-\frac{\alpha}{2} \frac{\partial (d-y)^2}{\partial W_{ij}}=-2\frac{\alpha}{2}(d-y) \large \frac{\partial (d-y)}{\partial W_{ij}}ΔWijk=α(−∂W∂E)=−2α∂Wij∂(d−y)2=−22α(d−y)∂Wij∂(d−y)
which can be simplified as:
ΔWijk=α(d−y)∂y∂Wij\Delta W_{ij}^k=\alpha(d-y) \frac{\partial y}{\partial W_{ij}}ΔWijk=α(d−y)∂Wij∂y
Equation 11
(Notice that ddd is a constant value, so it's partial derivative is simply a zero)

This partial derivative of the output with respect to each weight, defines the gradient and is often denoted by the Greek letter δ\deltaδ.

$$\delta_{ij} = \frac{\partial y}{\partial W_{ij}}$$

Equation 12

We will find all the elements of the gradient using the chain rule.

If you are feeling confident with the **chain rule** and understand how to apply it, skip the next video and continue with our example. Otherwise, give Luis a few minutes of your time as he takes you through the process!

https://www.youtube.com/watch?v=YAhIBOnbt54&feature=emb_logo

Backpropagation- Example (part b)

Now that we understand the chain rule, we can continue with our **backpropagation** example, where we will calculate the gradient

https://www.youtube.com/watch?time_continue=387&v=yiSwuMP2UIA&feature=emb_logo

In our example we only have one hidden layer, so our backpropagation process will consist of two steps:

Step 1: Calculating the gradient with respect to the weight vector W2W^2W2 (from the output to the hidden layer).
Step 2: Calculating the gradient with respect to the weight matrix W1W^1W1 (from the hidden layer to the input).

Step 1 (Note that the weight vector referenced here will be W2W^2W2. All indices referring to W2W^2W2 have been omitted from the calculations to keep the notation simple).

$$y = \sum_{i}^{3}(h_i W_i)$$
$$\Rightarrow \delta_i = \frac{\partial y}{\partial W_i} = \frac{\partial \sum_i^3 (h_i W_i)}{\partial W_i} = h_i$$

Equation 13

As you may recall:

ΔWij=α(d−y)∂y∂Wij\large\Delta W_{ij}=\alpha(d-y) \frac{\partial y}{\partial W_{ij}}ΔWij=α(d−y)∂Wij∂y

In this specific step, since the output is of only a single value, we can rewrite the equation the following way (in which we have a weights *vector*):

ΔWi=α(d−y)∂y∂Wi\large\Delta W_i=\alpha(d-y) \frac{\partial y}{\partial W_i}ΔWi=α(d−y)∂Wi∂y

Since we already calculated the gradient, we now know that the incremental value we need for step one is:

ΔWi=α(d−y)hi\Delta W_i=\alpha(d-y) h_iΔWi=α(d−y)hi

Equation 14

Having calculated the incremental value, we can update vector W2W^2W2 the following way:

$$W^2_{new} = W^2_{previous} + \Delta W^2_i$$
$$W^2_{new} = W^2_{previous} + \alpha(d-y)h_i$$

Equation 15

Step 2 (In this step, we will need to use both weight matrices. Therefore we will not be omitting the weight indices.)

In our second step we will update the weights of matrix W1W^1W1 by calculating the partial derivative of yyy with respect to the weight matrix W1W^1W1.

The chain rule will be used the following way:

obtain the partial derivative of yyy with respect to h¯\bar{h}h¯, and multiply it by the partial derivative of h¯\bar{h}h¯ with respect to the corresponding elements in W1W^1W1. Instead of referring to vector h¯\bar{h}h¯, we can observe each element and present the equation the following way:

$$\delta_{ij} = \frac{\partial y}{\partial W_{ij}^1} = \sum_{p=1}^{N} (\frac{\partial y}{\partial h_p} \frac{\partial h_p}{\partial W_{ij}^1})$$

Equation 16

In this example we have only 3 neurons the the single hidden layer, therefore this will be a linear combination of three elements:

$$\delta_{ij} = \frac{\partial y}{\partial W_{ij}^1} = \sum_{p=1}^{3} (\frac{\partial y}{\partial h_p} \frac{\partial h_p}{\partial W_{ij}^1})$$

Equation 17

We will calculate each derivative separately. ∂y∂hj\frac{\partial y}{\partial h_j}∂hj∂y will be calculated first, followed by ∂hj∂Wij1\frac{\partial h_j}{\partial W^1_{ij}}∂Wij1∂hj.

$$\frac{\partial y}{\partial h_j} = \frac{\partial \sum_{i=1}^{3} (h_i W_i^2)}{\partial h_j} = W_j^2$$

Equation 18

Notice that most of the derivatives were zero, leaving us with the simple solution of ∂y∂hj=Wj2 \frac{\partial y}{\partial h_{j}}=W^2_j∂hj∂y=Wj2

To calculate ∂hj∂Wij1\frac{\partial h_j}{\partial W^1_{{ij}}}∂Wij1∂hj we need to remember first that

$$h_j = \Phi(\sum_{i=1}^{2}(x_i W_{ij}^1))$$

Equation 19

Therefore:

$$\frac{\partial h_j}{\partial W_{ij}^1} = \frac{\partial \Phi(\sum_{i=1}^{2}(x_i W_{ij}^1))}{\partial W_{ij}^1}$$

Equation 20

Since the function hj\ h_j hj is an activation function (Φ\PhiΦ) of a linear combination, its partial derivative will be calculated the following way:

$$\frac{\partial h_j}{\partial W_{ij}^1} = \frac{\partial \Phi(\sum_{i=1}^{2}(x_i W_{ij}^1))}{\partial W_{ij}^1} = \frac{\partial \Phi(\sum_{i=1}^{2}(x_i W_{ij}^1))}{\partial(\sum_{i=1}^{2}(x_i W_{ij}^1))} \frac{\partial(\sum_{i=1}^{2}(x_i W_{ij}^1))}{\partial W_{ij}^1}$$

Equation 21

Given that there are various activation functions, we will leave the partial derivative of Φ\PhiΦ using a general notation. Each neuron j will have its own value for Φ\PhiΦ and Φ′\Phi'Φ′, according to the activation function we choose to use.

$$\frac{\partial \Phi(\sum_{i=1}^{2}(x_i W_{ij}^1))}{\partial(\sum_{i=1}^{2}(x_i W_{ij}^1))} = \Phi_j'$$

Equation 22

The second calculation of equation 21 can be calculated the following way:

(Notice how simple the result is, as most of the components of this partial derivative are zero).

$$\frac{\partial(\sum_{i=1}^{2}(x_i W_{ij}^1))}{\partial W_{ij}^1} = x_i$$

Equation 23

After understanding how to treat each multiplication of equation 21 separately, we can now summarize it the following way:

$$\frac{\partial h_j}{\partial W_{ij}^1} = \Phi'_j x_i$$

Equation 24

We are ready to finalize **step 2**, in which we update the weights of matrix W1W^1W1 by calculating the gradient shown in equation 17. From the above calculations, we can conclude that:

$$\delta_{ij} = \frac{\partial y}{\partial W_{ij}^1} = \sum_{p=1}^{3} \frac{\partial y}{\partial h_p} \frac{\partial h_p}{\partial W_{ij}^1} = W_j^2 \Phi'_j x_i$$

Equation 25

Since ΔWij1=α(d−y)∂y∂Wij1\Delta W^1_{ij}=\alpha(d-y) \large\frac{\partial y}{\partial W^1_{ij}}ΔWij1 =α(d−y)∂Wij1∂y , when finalizing step 2, we have:

$$\Delta W_{ij}^{1} = \alpha(d - y)W_{j}^{2}\Phi'_{j}x_{i}$$

Equation 26

Having calculated the incremental value, we can update vector W1W^1W1 the following way:

Wnew1=Wprevious1+ΔWij1W^1_{new}=W^1_{previous}+\Delta W^1_{ij}Wnew1=Wprevious1+ΔWij1

Wnew1=Wprevious1+α(d−y)Wj2Φj′xiW^1_{new}=W^1_{previous}+\alpha(d-y)W^2_j\Phi'_jx_iWnew1 =Wprevious1+α(d−y)Wj2Φj′xi

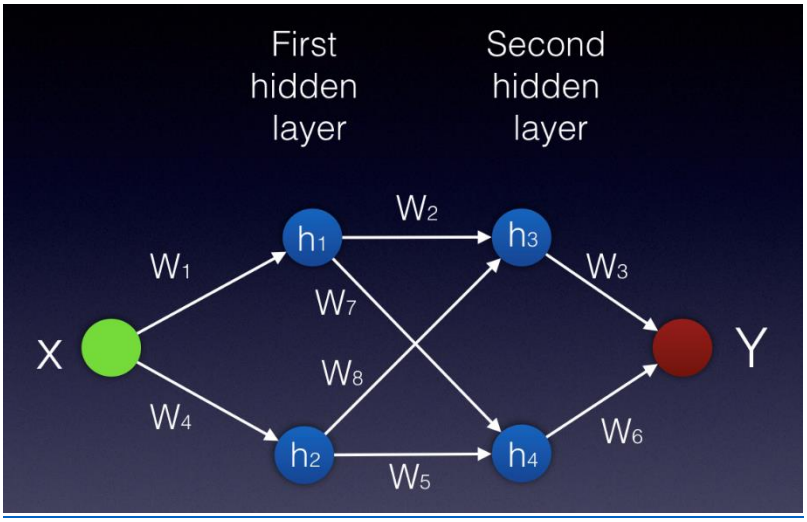
Equation 27

After updating the weight matrices we begin once again with the Feedforward pass, starting the process of updating the weights all over again.

This video touches on the subject of Mini Batch Training. We will further explain things in our **Hyperparameters** lesson coming up.

The following picture is of a feedforward network with

- A single input x
- Two hidden layers with two neurons in each layer
- A single output



QUIZ QUESTION

What is the update rule of weight matrix W1?

(In other words, what is the partial derivative of y with respect to W1?)

Hint: Use the chain rule

✓ Equation A

Equation A $$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial h_3}\frac{\partial h_3}{\partial h_1}\frac{\partial h_1}{\partial W_1} + \frac{\partial y}{\partial h_4}\frac{\partial h_4}{\partial h_1}\frac{\partial h_1}{\partial W_1}$$

Equation B $$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial h_3}\frac{\partial h_3}{\partial h_1}\frac{\partial h_1}{\partial W_1} + \frac{\partial y}{\partial h_4}\frac{\partial h_4}{\partial h_2}\frac{\partial h_2}{\partial W_1}$$

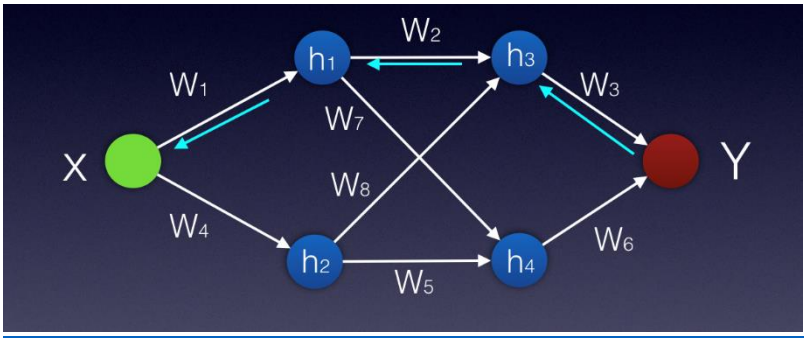
Equation C $$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial h_3}\frac{\partial h_3}{\partial h_1}\frac{\partial h_1}{\partial W_1}$$

Equation D $$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial h_4}\frac{\partial h_4}{\partial h_1}\frac{\partial h_1}{\partial W_1}$$

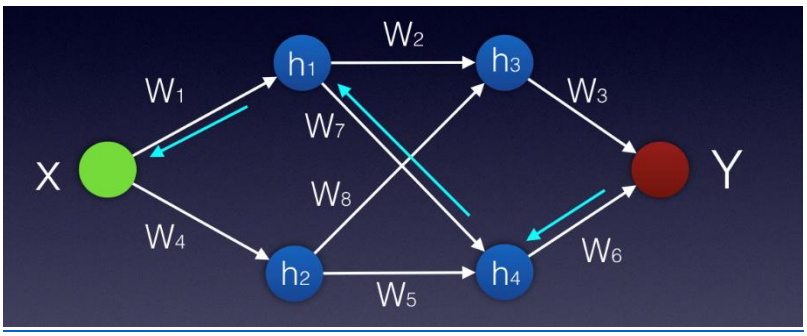
There are two separate paths which W1W_1*W*1 contributes to the output in:

- Path A
- Path B

(both displayed in the pictures below)



Path A



Path B

The mathematical derivations considering path A (while applying the chain rule) are:

$$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial h_3}\frac{\partial h_3}{\partial h_1}\frac{\partial h_1}{\partial W_1}$$

The mathematical derivations considering path B (while applying the chain rule) are:

$$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial h_4}\frac{\partial h_4}{\partial h_1}\frac{\partial h_1}{\partial W_1}$$

To finalize our calculations we need to consider all of the paths contributing to the calculation of y. In this case we have the two paths mentioned. Therefore, the final calculation will be the addition of the derivatives calculated in each path.

$$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial h_3}\frac{\partial h_3}{\partial h_1}\frac{\partial h_1}{\partial W_1} + \frac{\partial y}{\partial h_4}\frac{\partial h_4}{\partial h_1}\frac{\partial h_1}{\partial W_1}$$

RNN part a

We are finally ready to talk about Recurrent Neural Networks (or RNNs), where we will be opening the doors to new content!

https://www.youtube.com/watch?v=ofbnDxGSUcg&feature=emb_logo

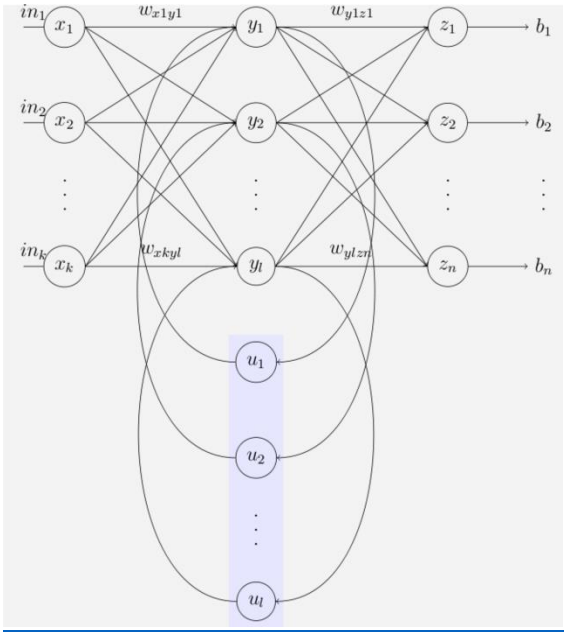
RNNs are based on the same principles as those behind FFNNs, which is why we spent so much time reminding ourselves of the feedforward and backpropagation steps that are used in the training phase.

There are two main differences between FFNNs and RNNs. The Recurrent Neural Network uses:

- **sequences** as inputs in the training phase, and
- **memory** elements

Memory is defined as the output of hidden layer neurons, which will serve as additional input to the network during next training step.

The basic three layer neural network with feedback that serve as memory inputs is called the **Elman Network** and is depicted in the following picture:



Elman Network, source: Wikipedia

As mentioned in the *History* concept, here is the original Elman Network publication from 1990. This link is provided here as it's a significant milestone in the world on RNNs. To simplify things a bit, you can take a look at the following additional info.

Let's continue now to the next video with more information about RNNs.

https://www.youtube.com/watch?v=wsif3p5t7CI&feature=emb_logo

As we've see, in FFNN the output at any time *t*, is a function of the current input and the weights. This can be easily expressed using the following equation:

$$\bar{y}_t = F(\bar{x}_t, W)$$

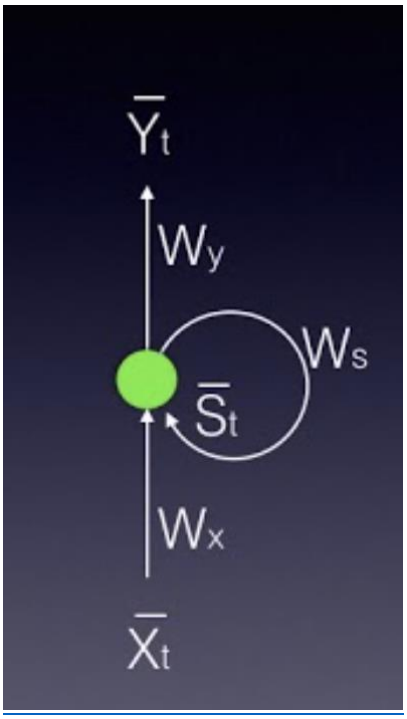
Equation 28

In RNNs, our output at time t, depends not only on the current input and the weight, but also on previous inputs. In this case the output at time t will be defined as:

$$\bar{y}_t = F(\bar{x}_t, \bar{x}_{t-1}, \bar{x}_{t-2}, \cdots, \bar{x}_{t-t_0}, W)$$

Equation 29

This is the RNN **folded model**:



The RNN folded model

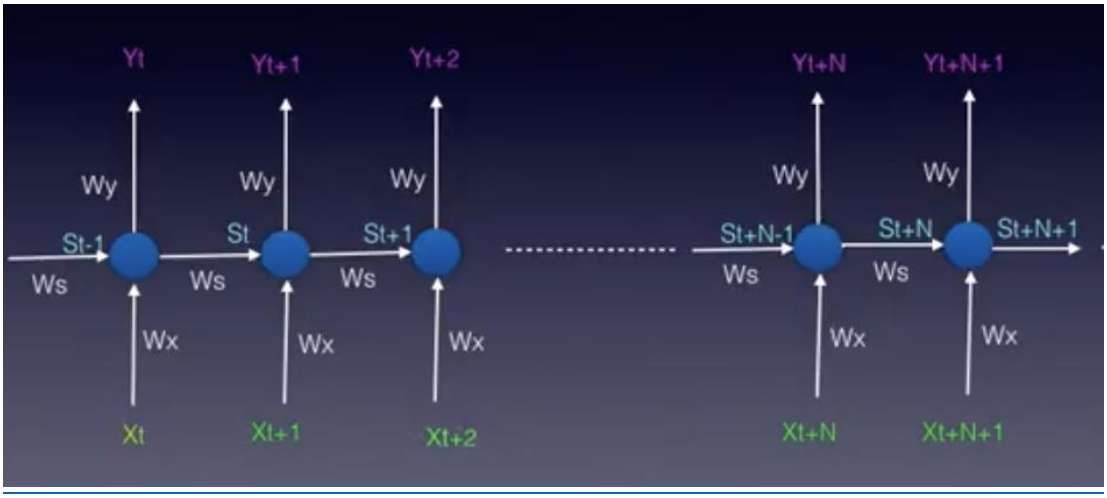
In this picture, x¯\bar{x}x¯ represents the input vector, y¯\bar{y}y¯ represents the output vector and s¯\bar{s}s¯ denotes the state vector.

WxW_xWx is the weight matrix connecting the inputs to the state layer.

WyW_yWy is the weight matrix connecting the state layer to the output layer.

WsW_sWs represents the weight matrix connecting the state from the previous timestep to the state in the current timestep.

The model can also be "unfolded in time". The **unfolded model** is usually what we use when working with RNNs.



The RNN unfolded model

In both the folded and unfolded models shown above the following notation is used:

x¯\bar{x}x¯ represents the input vector, y¯\bar{y}y¯ represents the output vector and s¯\bar{s}s¯ represents the state vector.

WxW_xWx is the weight matrix connecting the inputs to the state layer.

WyW_yWy is the weight matrix connecting the state layer to the output layer.

WsW_sWs represents the weight matrix connecting the state from the previous timestep to the state in the current timestep.

In FFNNs the hidden layer depended only on the current inputs and weights, as well as on an activation function Φ\PhiΦ in the following way:

h¯=Φ(x¯W)\bar{h}=\Phi(\bar{x}W)h¯=Φ(x¯W).

Equation 30

In RNNs the state layer depended on the current inputs, their corresponding weights, the activation function and **also** on the previous state:

$$\bar{s}_t = \Phi(\bar{x}_t W_x + \bar{s}_{t-1} W_s)$$

Equation 31

The output vector is calculated exactly the same as in FFNNs. It can be a linear combination of the inputs to each output node with the corresponding weight matrix WyW_yWy, or a softmax function of the same linear combination.

y¯t=s¯tWy\bar{y}_t=\bar{s}_t W_yy¯t=s¯tWy

or

y¯t=σ(s¯tWy)\bar{y}_t=\sigma(\bar{s}_t W_y)y¯t=σ(s¯tWy)

Equation 32

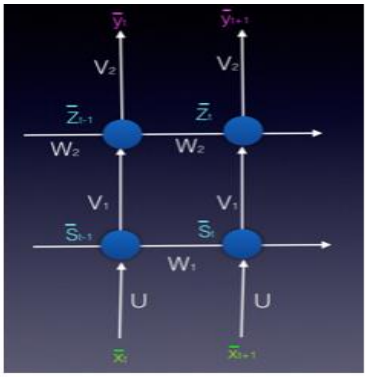
The next video will focus on the **unfolded model** as we try to further understand it.

https://www.youtube.com/watch?v=xLIA_PTWXog&feature=emb_logo

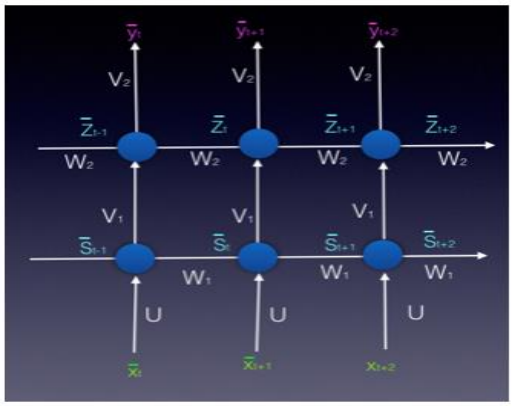
QUIZ QUESTION

Look at the above picture of a folded model of a RNN. Which of the pictures below represents the unfolded model of the same network?

- Picture A
- Picture B
- Both A and B are correct



Picture A



Picture B

RNN Example

In this example we will illustrate how RNNs can be helpful in detecting sequences. When detecting a sequence, the system has to remember what the previous inputs were, so it makes sense to use a recurrent network.

If you are unfamiliar with the term sequence detection, the idea is to see if a specific pattern of inputs has entered the system. In our example the pattern will be the word U,D,A,C,I,T,Y.

https://www.youtube.com/watch?v=MDLk3fhpTx0&feature=emb_logo

Backpropagation Through Time

We are now ready to understand how to train the RNN.

When we train RNNs we also use backpropagation, but with a conceptual change. The process is similar to that in the FFNN, with the exception that we need to consider previous time steps, as the system has memory. This process is called **Backpropagation Through Time (BPTT)** and will be the topic of the next three videos.

- As always, don't forget to take notes.

In the following videos we will use the Loss Function for our error. The Loss Function is the square of the difference between the desired and the calculated outputs. There are variations to the Loss Function, for example, factoring it with a scalar. In the backpropagation example we used a factoring scalar of 1/2 for calculation convenience.

As described previously, the two most commonly used are the Mean Squared Error (MSE) (usually used in regression problems) and the cross entropy (usually used in classification problems).

Here, we are using a variation of the MSE.

https://www.youtube.com/watch?v=eE2L3-2wKac&feature=emb_logo

Before diving into Backpropagation Through Time we need a few reminders.

The state vector s¯t\bar{s}_ts¯t is calculated the following way:

$$\bar{s}_t = \Phi(\bar{x}_t W_x + \bar{s}_{t-1} W_s)$$

Equation 33

The output vector y¯t\bar{y}_ty¯t can be product of the state vector s¯t\bar{s}_ts¯t and the corresponding weight elements of matrix WyW_yWy. As mentioned before, if the desired outputs are between 0 and 1, we can also use a softmax function. The following set of equations depicts these calculations:

$$\bar{y}_t = \bar{s}_t W_y$$
$$Or$$
$$\bar{y}_t = \sigma(\bar{s}_t W_y)$$

Equation 34

As mentioned before, for the error calculations we will use the Loss Function, where

EtE_tEt represents the output error at time t

dtd_tdt represents the desired output at time t

ytyty_tyt represents the calculated output at time t

$$E_t = (\bar{d}_t - \bar{y}_t)^2$$

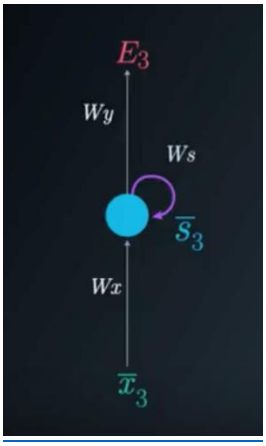
Equation 35

In **BPTT** we train the network at timestep t as well as take into account all of the previous timesteps.

The easiest way to explain the idea is to simply jump into an example.

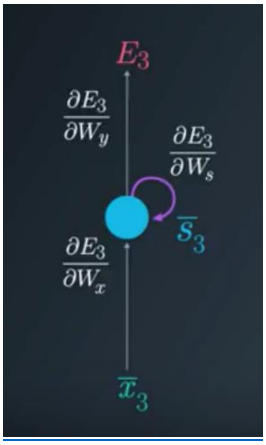
In this example we will focus on the **BPTT** process for time step t=3. You will see that in order to adjust all three weight matrices, Wx,WsW_x, W_s Wx,Ws and WyW_yWy, we need to consider timestep 3 as well as timestep 2 and timestep 1.

As we are focusing on timestep t=3, the Loss function will be: E3=(d¯3−y¯3)2E_3=(\bar{d}_3-\bar{y}_3)^2E3=(d¯3−y¯3)2



The Folded Model at Timestep 3

To update each weight matrix, we need to find the partial derivatives of the Loss Function at time 3, as a function of all of the weight matrices. We will modify each matrix using gradient descent while considering the previous timesteps.



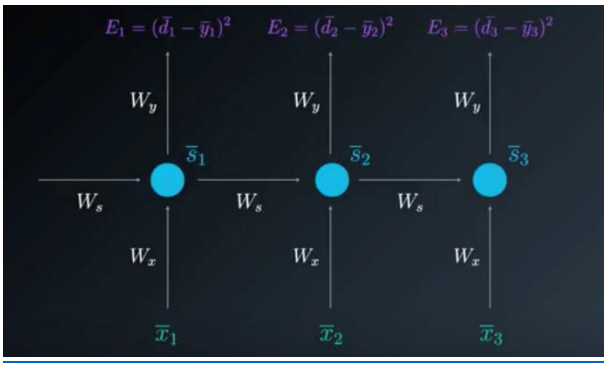
Gradient Considerations in the Folded Model

We will now unfold the model. You will see that unfolding the model in time is very helpful in visualizing the number of steps (translated into multiplication) needed in the Backpropagation Through Time process. These multiplications stem from the chain rule and are easily visualized using this model.

In this video we will understand how to use Backpropagation Through Time (BPTT) when adjusting two weight matrices:

- WyW_yWy - the weight matrix connecting the state the output
- WsW_sWs - the weight matrix connecting one state to the next state

The unfolded model can be very helpful in visualizing the BPTT process.



The Unfolded Model at timestep 3

Gradient calculations needed to adjust WyW_yWy

https://www.youtube.com/watch?v=bUU9BEQw0IA&feature=emb_logo

The partial derivative of the Loss Function with respect to WyW_yWy is found by a simple one step chain rule: (Note that in this case we do not need to use BPTT. Visualization of the calculations path can be found in the video).

$$\frac{\partial E_3}{\partial W_y} = \frac{\partial E_3}{\partial \bar{y}_3} \frac{\partial \bar{y}_3}{\partial W_y}$$

Equation 36

Generally speaking, we can consider multiple timesteps back, and not only 3 as in this example. For an arbitrary timestep N, the gradient calculation needed for adjusting WyW_yWy, is:

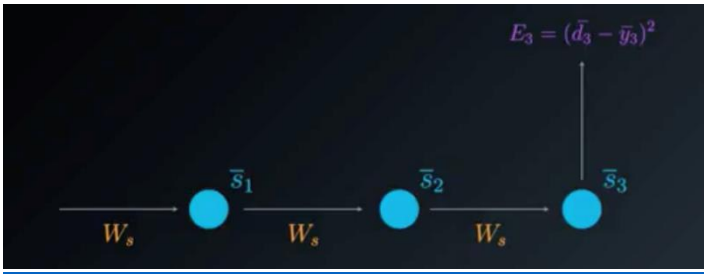
$$\frac{\partial E_N}{\partial W_y} = \frac{\partial E_N}{\partial \bar{y}_N} \frac{\partial \bar{y}_N}{\partial W_y}$$

Equation 37

Gradient calculations needed to adjust WsW_sWs

We still need to adjust WsW_sWs the weight matrix connecting one state to the next and WxW_xWx the weight matrix connecting the input to the state. We will arbitrarily start with WsW_sWs.

To understand the **BPTT** process, we can simplify the unfolded model. We will focus on the contributions of WsW_sWs to the output, the following way:



Simplified Unfolded model for Adjusting Ws

When calculating the partial derivative of the Loss Function with respect to WsW_sWs, we need to consider all of the states contributing to the output. In the case of this example it will be states s3¯\bar{s_3}s3¯ which depends on its predecessor s2¯\bar{s_2}s2¯ which depends on its predecessor s1¯\bar{s_1}s1¯, the first state.

In **BPTT** we will take into account every gradient stemming from each state, **accumulating** all of these contributions.

- At timestep t=3, the contribution to the gradient stemming from s3¯\bar{s_3}s3¯ is the following : (Notice the use of the chain rule here. If you need, go back to the video to visualize the calculation path).

$$\frac{\partial E_3}{\partial W_s} = \frac{\partial E_3}{\partial \bar{y}_3} \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \frac{\partial \bar{s}_3}{\partial W_s}$$

Equation 38

- At timestep t=3, the contribution to the gradient stemming from s2¯\bar{s_2}s2¯ is the following : (Notice how the equation, derived by the chain rule, considers the contribution of s2¯\bar{s_2}s2¯ to s3¯\bar{s_3}s3¯ . If you need, go back to the video to visualize the calculation path).

$$\frac{\partial E_3}{\partial W_s} = \frac{\partial E_3}{\partial \bar{y}_3} \frac{\partial \bar{y}_3}{\partial \bar{s}_3} \frac{\partial \bar{s}_3}{\partial \bar{s}_2} \frac{\partial \bar{s}_2}{\partial W_s}$$

Equation 39

- At timestep t=3, the contribution to the gradient stemming from s1¯\bar{s_1}s1¯ is the following : (Notice how the equation, derived by the chain rule, considers the contribution of s1¯\bar{s_1}s1¯ to s2¯\bar{s_2}s2¯ and s3¯\bar{s_3}s3¯ . If you need, go back to the video to visualize the calculation path).

$$\frac{\partial E_3}{\partial W_s} = \frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s_3}}\frac{\partial \bar{s_3}}{\partial \bar{s_2}}\frac{\partial \bar{s_2}}{\partial \bar{s_1}}\frac{\partial \bar{s_1}}{\partial W_s}$$

Equation 40

After considering the contributions from all three states: s3¯\bar{s_3}s3¯ ,s2¯\bar{s_2}s2¯ and s1¯\bar{s_1}s1¯, we will **accumulate** them to find the final gradient calculation.

The following equation is the gradient contributing to the adjustment of WsW_sWs using **Backpropagation Through Time**:

$$\begin{aligned}\frac{\partial E_3}{\partial W_s} = &\frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s_3}}\frac{\partial \bar{s_3}}{\partial W_s} + \\ &\frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s_3}}\frac{\partial \bar{s_3}}{\partial \bar{s_2}}\frac{\partial \bar{s_2}}{\partial W_s} + \\ &\frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s_3}}\frac{\partial \bar{s_3}}{\partial \bar{s_2}}\frac{\partial \bar{s_2}}{\partial \bar{s_1}}\frac{\partial \bar{s_1}}{\partial W_s}\end{aligned}$$

Equation 41

In this example we had 3 time steps to consider, therefore we accumulated three partial derivative calculations. Generally speaking, we can consider multiple timesteps back. If you look closely at the three components of equation 41, you will notice a pattern. You will find that as we propagate a step back, we have an additional partial derivatives to consider in the chain rule. Mathematically this can be easily written in the following general equation for adjusting WsW_sWs using **BPTT**:

$$\frac{\partial E_N}{\partial W_s} = \sum_{i=1}^{N}\frac{\partial E_N}{\partial \bar{y}_N}\frac{\partial \bar{y}_N}{\partial \bar{s_i}}\frac{\partial \bar{s_i}}{\partial W_s}$$

Equation 42

Notice that Equation 6 considers a general setting of N steps back. As mentioned in this lesson, capturing relationships that span more than 8 to 10 steps back is practically impossible due to the vanishing gradient problem. We will talk about a solution to this problem in our LSTM section coming up soon.

We still need to adjust WxW_xWx, the weight matrix connecting the input to the state.

Let's take a small break. You can use this time to go over the **BPTT** process we've seen so far. Try to get yourself comfortable with the math.

Once you are feeling confident with the content of the video you just viewed, try to derive the calculations for adjusting the last matrix, WxW_xWx by yourself. This is by no means a must, but if you feel that you are up for the challenge, go for it! It will be interesting to compare your notes with ours.

If you chose to take on the challenge, focus on simplifying the unfolded model, leaving only what you need for the calculations. Sketch the backpropagation "path", and step by step think of how the chain rule helps with the derivations here. Don't forget to **accumulate!**.

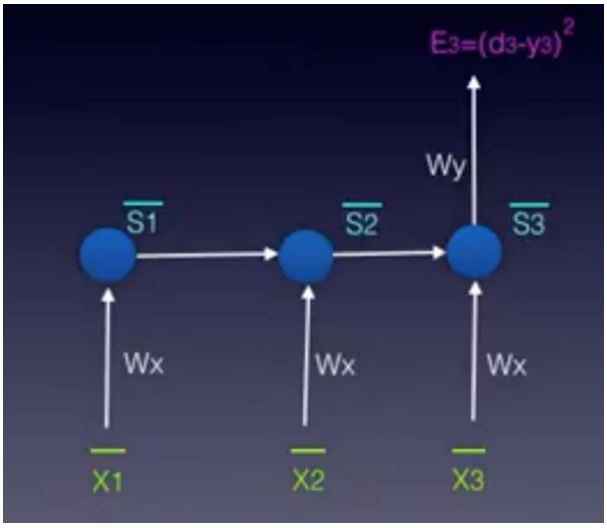
Last step! Adjusting WxW_xWx, the weight matrix connecting the input to the state.

If you took on the previous challenge of deriving the math by yourself first, sit back, fasten your seat belts and compare our notes to yours! Don't worry if you made mistakes, we all do. Your mistakes will help you learn what to avoid next time.

https://www.youtube.com/watch?v=uBy_eIJDD1M&feature=emb_logo

Gradient calculations needed to adjust WxW_xWx

To further understand the **BPTT** process, we will simplify the unfolded model again. This time the focus will be on the contributions of WxW_xWx to the output, the following way:



Simplified Unfolded model for Adjusting Wx

When calculating the partial derivative of the Loss Function with respect to to WxW_xWx we need to consider, again, all of the states contributing to the output. As we saw before, in the case of this example it will be states s3¯\bar{s_3}s3¯ which depend on its predecessor s2¯\bar{s_2}s2¯ which depends on its predecessor s1¯\bar{s_1}s1¯, the first state.

As we mentioned previously, in **BPTT** we will take into account each gradient stemming from each state, **accumulating** all of the contributions.

- At timestep t=3, the contribution to the gradient stemming from s3¯\bar{s_3}s3¯ is the following : (Notice the use of the chain rule here. If you need, go back to the video to visualize the calculation path).

$$\frac{\partial E_3}{\partial W_x} = \frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s}_3}\frac{\partial \bar{s}_3}{\partial W_x}$$

Equation 43

- At timestep t=3, the contribution to the gradient stemming from s2¯\bar{s_2}s2¯ is the following : (Notice how the equation, derived by the chain rule, considers the contribution of s2¯\bar{s_2}s2¯ to s3¯\bar{s_3}s3¯ . If you need, go back to the video to visualize the calculation path).

$$\frac{\partial E_3}{\partial W_x} = \frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s}_3}\frac{\partial \bar{s}_3}{\partial \bar{s}_2}\frac{\partial \bar{s}_2}{\partial W_x}$$

Equation 44

- At timestep t=3, the contribution to the gradient stemming from s1¯\bar{s_1}s1¯ is the following : (Notice how the equation, derived by the chain rule, considers the contribution of s1¯\bar{s_1}s1¯ to s2¯\bar{s_2}s2¯ and s3¯\bar{s_3}s3¯ . If you need, go back to the video to visualize the calculation path).

$$\frac{\partial E_3}{\partial W_x} = \frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s_3}}\frac{\partial \bar{s_3}}{\partial \bar{s_2}}\frac{\partial \bar{s_2}}{\partial \bar{s_1}}\frac{\partial \bar{s_1}}{\partial W_x}$$

Equation 45

After considering the contributions from all three states: s3¯\bar{s_3}s3¯ ,s2¯\bar{s_2}s2¯ and s1¯\bar{s_1}s1¯, we will **accumulate** them to find the final gradient calculation.

The following equation is the gradient contributing to the adjustment of WxW_xWx using **Backpropagation Through Time**:

$$\begin{aligned}\frac{\partial E_3}{\partial W_x} = &\frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s_3}}\frac{\partial \bar{s_3}}{\partial W_x} + \\ &\frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s_3}}\frac{\partial \bar{s_3}}{\partial \bar{s_2}}\frac{\partial \bar{s_2}}{\partial W_x} + \\ &\frac{\partial E_3}{\partial \bar{y}_3}\frac{\partial \bar{y}_3}{\partial \bar{s_3}}\frac{\partial \bar{s_3}}{\partial \bar{s_2}}\frac{\partial \bar{s_2}}{\partial \bar{s_1}}\frac{\partial \bar{s_1}}{\partial W_x}\end{aligned}$$

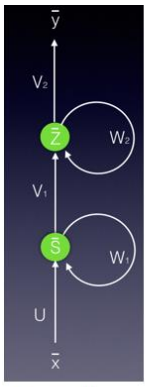
Equation 46

As mentioned before, in this example we had 3 time steps to consider, therefore we accumulated three partial derivative calculations. Generally speaking, we can consider multiple timesteps back. If you look closely at equations 1, 2 and 3, you will notice a pattern again. You will find that as we propagate a step back, we have an additional partial derivatives to consider in the chain rule. Mathematically this can be easily written in the following general equation for adjusting WxW_xWx using **BPTT**:

$$\frac{\partial E_N}{\partial W_x} = \sum_{i=1}^{N}\frac{\partial E_N}{\partial \bar{y}_N}\frac{\partial \bar{y}_N}{\partial \bar{s_i}}\frac{\partial \bar{s_i}}{\partial W_x}$$

Equation 47

Notice the similarities between the calculations of ∂E3∂Ws\frac{\partial{E_3} }{\partial W_s}∂Ws∂E3 and ∂E3∂Wx\frac{\partial{E_3} }{\partial W_x}∂Wx∂E3. Hopefully after understanding the calculation process of ∂E3∂Ws\frac{\partial{E_3} }{\partial W_s}∂Ws∂E3, understanding that of ∂E3∂Wx\frac{\partial{E_3} }{\partial W_x}∂Wx ∂E3 was straight forward.



A folded RNN model

QUIZ QUESTION

Consider the above folded RNN Model. Both states **S** and **Z** have multiple neurons in each layer. The mathematical derivation of state **Z** at time t is:

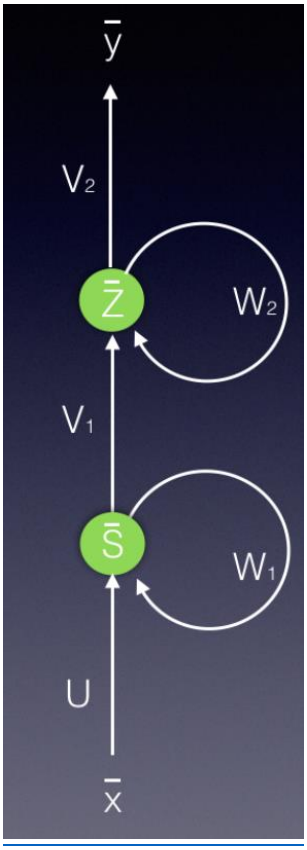
Equation B $$\bar{z}_t = \phi(\bar{s}_t v_1 + \bar{z}_t w_2)$$

Equation C $$z_t = \phi(s_t v_1 + z_t w_2)$$

Equation D $$\bar{z}_t = \phi(\bar{s}_t v_1 + \bar{z}_{t-1} w_2)$$

Solution

¯ and ¯ are vectors, as we indicate that they have multiple neurons in each layer. Using this logic we can understand that equations A and C are incorrect. Since w_2 connects the hidden state ¯ to itself, we know that we need to consider the previous timestep here. Therefore only equation D is the correct one.



A folded RNN model

Quiz Question

Lets look at the same folded model again (displayed above). Assume that the error is noted by the symbol **E**. What is the update rule of weight matrix V1 at time t, over a single timestep ?

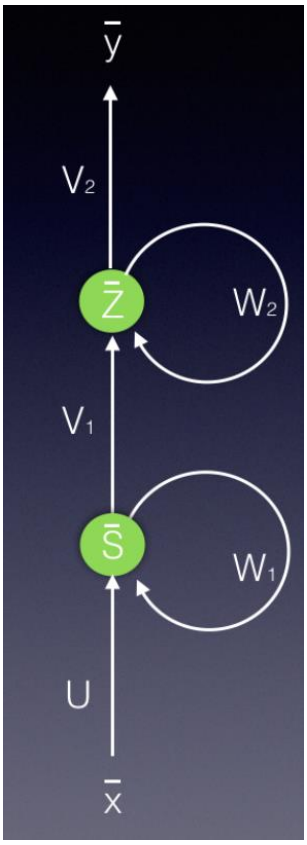
Equation B $\Delta v_1 = -\alpha \frac{\partial E_t}{\partial v_1} = -\alpha \frac{\partial E_t}{\partial \bar{y}_t} \frac{\partial \bar{y}_t}{\partial \bar{Z}_t} \frac{\partial \bar{Z}_t}{\partial \bar{v_1}}$

Equation C $\Delta v_1 = \frac{\partial E_t}{\partial v_1} = \frac{\partial E_t}{\partial \bar{y}_t} \frac{\partial \bar{y}_t}{\partial \bar{Z}_t} \frac{\partial \bar{Z}_t}{\partial \bar{v_1}}$

Equation D $\Delta v_1 = -\alpha \frac{\partial E_t}{\partial v_1} = -\alpha \frac{\partial E_t}{\partial \bar{y}_t} \frac{\partial \bar{y}_t}{\partial \bar{v_1}}$

Solution

Equation B Is the only equation with the correct derivation of the chain rule with the proper use of the learning rate.



A folded RNN model

Quiz Question

Lets look at the same folded model again (displayed above). Assume that the error is noted by the symbol **E**. What is the update rule of weight matrix U at time t+1 (over 2 timesteps) ? Hint: Use the unfolded model for a better visualization.

Equation C

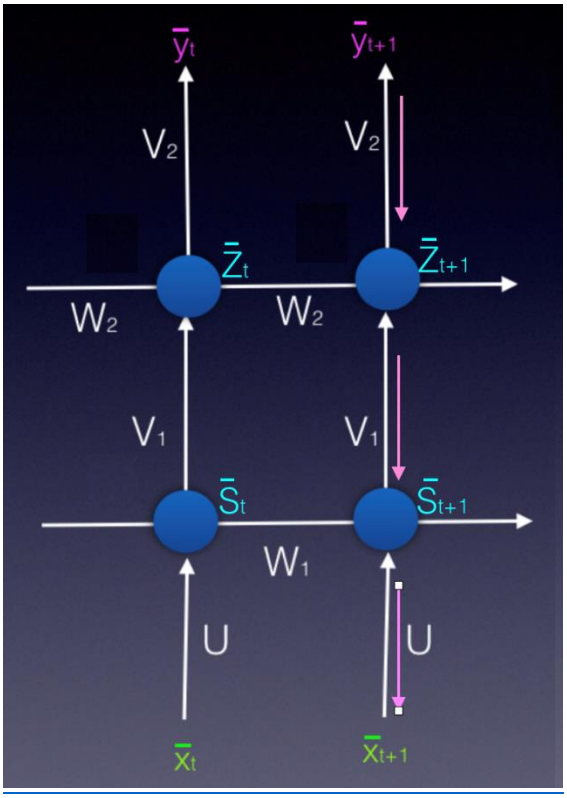
$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial U} + \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{z}_t} \frac{\partial \bar{z}_t}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U}$$
$$+ \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U}$$

Solution

To understand how to update weight matrix U, we will need to unfold the model in time. We will unfold the model over two time steps, as we need to look only time t and time t+1. The following three pictures will help you understand the **three** paths we need to consider. Notice that we have two hidden layers that serve as memory elements, so this case will be different than the one we saw in the video, but the idea is the same. We will use **BPTT** while applying the chain rule.

Solution

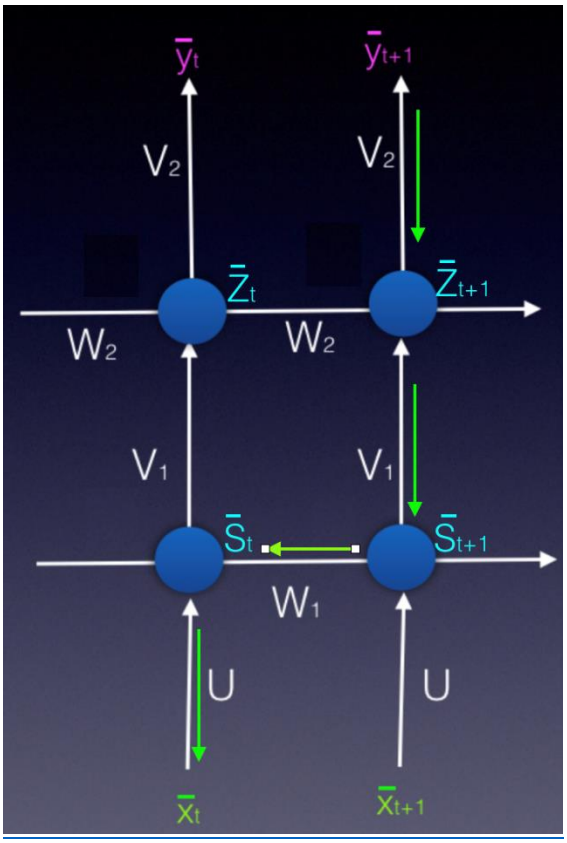
To understand how to update weight matrix U, we will need to unfold the model in time. We will unfold the model over two time steps, as we need to look only time t and time t+1. The following three pictures will help you understand the **three** paths we need to consider. Notice that we have two hidden layers that serve as memory elements, so this case will be different than the one we saw in the video, but the idea is the same. We will use **BPTT** while applying the chain rule.



The first path to consider

The following is the equation we derive using the first path:

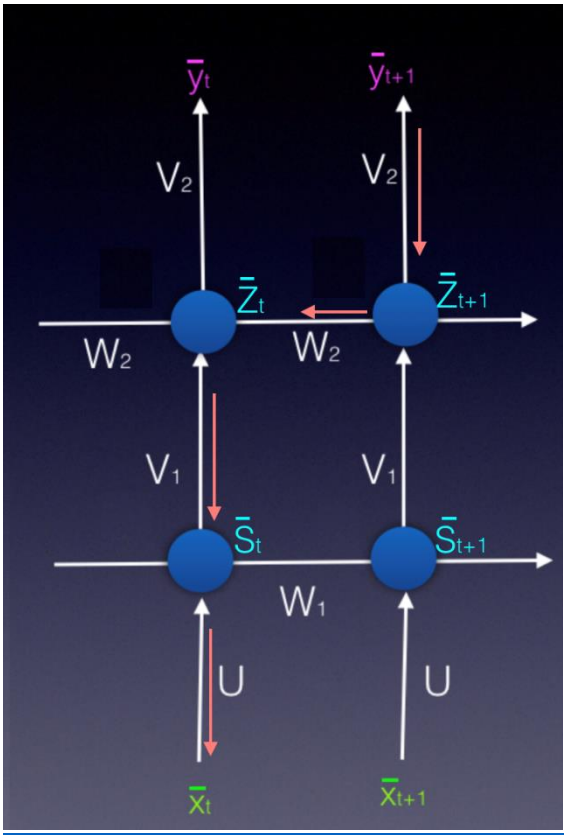
$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial U}$$



The second path to consider

The following is the equation we derive using the second path:

$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U}$$



The third path to consider

The following is the equation we derive using the third path:

$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{z}_t} \frac{\partial \bar{z}_t}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U}$$

Finally, after considering all three paths, we can derive the correct equation for the purposes of updating weight matrix U, using BPTT:

$$\frac{\partial E_{t+1}}{\partial U} = \frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial U}$$
$$+$$
$$\frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{z}_t} \frac{\partial \bar{z}_t}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U}$$
$$+$$
$$\frac{\partial E_{t+1}}{\partial \bar{y}_{t+1}} \frac{\partial \bar{y}_{t+1}}{\partial \bar{z}_{t+1}} \frac{\partial \bar{z}_{t+1}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial \bar{s}_t} \frac{\partial \bar{s}_t}{\partial U}$$

The final answer for BPTT Quiz 3

Some more math

Send Feedback

Toggle Sidebar

Some more math

Send Feedback

This section is given as bonus material and is not mandatory. If you are curious how we derived the final accumulative equation for BPTT, this section will help you out.

In the previous videos, we talked about **Backpropagation Through Time**. We used a lot of partial derivatives, accumulating the contributions to the change in the error from each state. Remember? When we needed a general scheme for the BPTT, I simply displayed the equation without giving you further explanations.

As a reminder, the following two equations were derived when adjusting the weights of matrix WsW_sWs and matrix WxW_xWx:

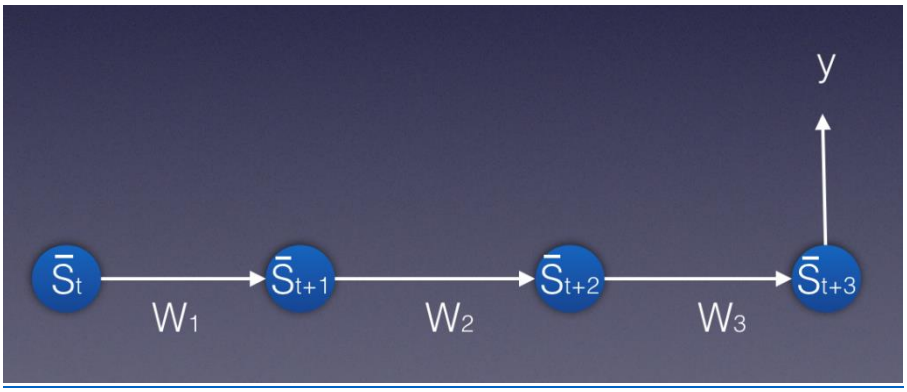
$$\frac{\partial E_N}{\partial W_s} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N} \frac{\partial \bar{y}_N}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial W_s}$$

Equation 48: BPTT calculations for the purpose of adjusting Ws

$$\frac{\partial E_N}{\partial W_x} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N} \frac{\partial \bar{y}_N}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial W_x}$$

Equation 49: BPTT calculations for the purpose of adjusting Wx

To generalize the case, we will avoid proving equation 48 or 49, and will focus on a general framework. Let's look at the following sketch, presenting a portion of a network:



In the picture above, we have four states, starting with sts_tst. We will initially consider the three weight matrices W1W_1W1,W2W_2W2 and W3W_3W3 as three different matrices.

Using the chain rule we can derive the following three equations:

$$\frac{\partial y}{\partial W_3} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial W_3}$$

$$\frac{\partial y}{\partial W_2} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial W_2}$$

$$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial W_1}$$

Equation 50 (Equation set)

In **Backpropagation Through Time** we accumulate the contributions, therefore:

$$\frac{\partial y}{\partial W} = \frac{\partial y}{\partial W_1} + \frac{\partial y}{\partial W_2} + \frac{\partial y}{\partial W_3}$$

Equation 51

Since this network is displayed as *unfolded in time*, we understand that the weight matrices connecting each of the states are identical. Therefore:

W1W_1W1=W2W_2W2=W3W_3W3

Lets simply call it weight matrix WWW. Therefore:

W1W_1W1=W2W_2W2=W3W_3W3=WWW

Equation 52

From *equation 52*, *equation 51* and the *set of equations 50* we derive that:

$$\frac{\partial y}{\partial W} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial W} + \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial W} + \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial W}$$

Equation 53

Equation 53 summarizes the mathematical procedure of BPTT and can be simply written as:

$$\frac{\partial y}{\partial W} = \sum_{i=t+1}^{t+3} \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial W}$$

Equation 54

Notice that for i=t+1i=t+1i=t+1, we derive the following:

$$\frac{\partial y}{\partial W} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial W}$$

Equation 55

With the use of the chain rule we can derive the following equation (displayed in *set of equations 50*).

$$\frac{\partial y}{\partial W} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial W}$$

Equation 56

A general derivation of the BPTT calculation can be displayed the following way:

$$\frac{\partial y}{\partial W} = \sum_{i=t+1}^{t+N} \frac{\partial y}{\partial \bar{s}_{t+N}} \frac{\partial \bar{s}_{t+N}}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial W}$$

Equation 57

Some more math

Send Feedback

Toggle Sidebar

Some more math

Send Feedback

This section is given as bonus material and is not mandatory. If you are curious how we derived the final accumulative equation for BPTT, this section will help you out.

In the previous videos, we talked about **Backpropagation Through Time**. We used a lot of partial derivatives, accumulating the contributions to the change in the error from each state. Remember? When we needed a general scheme for the BPTT, I simply displayed the equation without giving you further explanations.

As a reminder, the following two equations were derived when adjusting the weights of matrix WsW_sWs and matrix WxW_xWx:

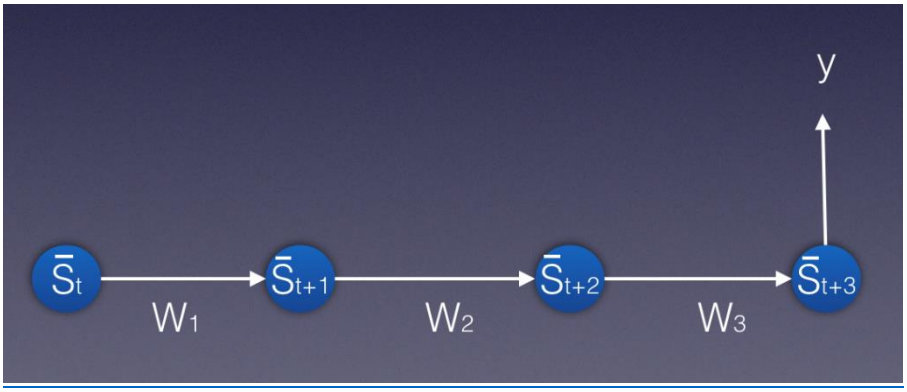
$$\frac{\partial E_N}{\partial W_s} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N} \frac{\partial \bar{y}_N}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial W_s}$$

Equation 48: BPTT calculations for the purpose of adjusting Ws

$$\frac{\partial E_N}{\partial W_x} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N} \frac{\partial \bar{y}_N}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial W_x}$$

Equation 49: BPTT calculations for the purpose of adjusting Wx

To generalize the case, we will avoid proving equation 48 or 49, and will focus on a general framework. Let's look at the following sketch, presenting a portion of a network:



In the picture above, we have four states, starting with sts_tst. We will initially consider the three weight matrices W1W_1W1,W2W_2W2 and W3W_3W3 as three different matrices.

Using the chain rule we can derive the following three equations:

$$\frac{\partial y}{\partial W_3} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial W_3}$$

$$\frac{\partial y}{\partial W_2} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial W_2}$$

$$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s}_{t+1}}{\partial W_1}$$

Equation 50 (Equation set)

In **Backpropagation Through Time** we accumulate the contributions, therefore:

$$\frac{\partial y}{\partial W} = \frac{\partial y}{\partial W_1} + \frac{\partial y}{\partial W_2} + \frac{\partial y}{\partial W_3}$$

Equation 51

Since this network is displayed as *unfolded in time*, we understand that the weight matrices connecting each of the states are identical. Therefore:

W1W_1W1=W2W_2W2=W3W_3W3

Lets simply call it weight matrix WWW. Therefore:

W1W_1W1=W2W_2W2=W3W_3W3=WWW

Equation 52

From *equation 52*, *equation 51* and the *set of equations 50* we derive that:

$$\frac{\partial y}{\partial W} = \frac{\partial y}{\partial \bar{s}_{t+3}}\frac{\partial \bar{s}_{t+3}}{\partial W} + \frac{\partial y}{\partial \bar{s}_{t+3}}\frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}}\frac{\partial \bar{s}_{t+2}}{\partial W} + \frac{\partial y}{\partial \bar{s}_{t+3}}\frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}}\frac{\partial \bar{s}_{t+2}}{\partial \bar{s}_{t+1}}\frac{\partial \bar{s}_{t+1}}{\partial W}$$

Equation 53

Equation 53 summarizes the mathematical procedure of BPTT and can be simply written as:

$$\frac{\partial y}{\partial W} = \sum_{i=t+1}^{t+3} \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial W}$$

Equation 54

Notice that for i=t+1i=t+1i=t+1, we derive the following:

$$\frac{\partial y}{\partial W} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s_{t+1}}}{\partial W}$$

Equation 55

With the use of the chain rule we can derive the following equation (displayed in *set of equations 50*).

$$\frac{\partial y}{\partial W} = \frac{\partial y}{\partial \bar{s}_{t+3}} \frac{\partial \bar{s}_{t+3}}{\partial \bar{s}_{t+2}} \frac{\partial \bar{s}_{t+2}}{\partial \bar{s}_{t+1}} \frac{\partial \bar{s_{t+1}}}{\partial W}$$

Equation 56

A general derivation of the BPTT calculation can be displayed the following way:

$$\frac{\partial y}{\partial W} = \sum_{i=t+1}^{t+N} \frac{\partial y}{\partial \bar{s}_{t+N}} \frac{\partial \bar{s}_{t+N}}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial W}$$

Equation 57

RNN Summary

Let's summarize what we have seen so far:

https://www.youtube.com/watch?time_continue=2&v=nXP0oGGRrO8&feature=emb_logo

As you have seen, in RNNs the current state depends on the input as well as the previous states, with the use of an activation function.

$$\bar{s}_t = \Phi(\bar{x}_t W_x + \bar{s}_{t-1} W_s)$$

Equation 56

The current output is a simple linear combination of the current state elements with the corresponding weight matrix.

y¯t=s¯tWy\bar{y}_t=\bar{s}_t W_yy¯t=s¯tWy (without the use of an activation function)

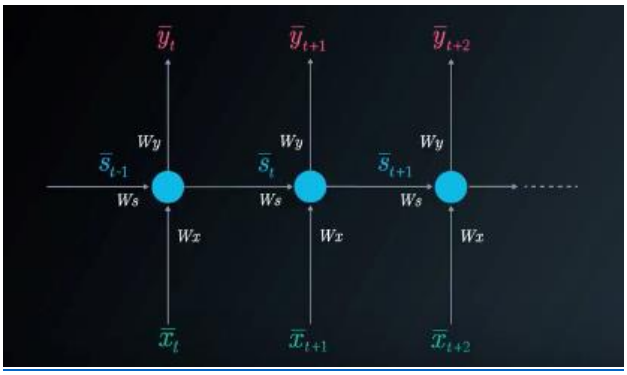
or

y¯t=σ(s¯tWy)\bar{y}_t=\sigma(\bar{s}_t W_y)y¯t=σ(s¯tWy) (with the use of an activation function)

Equation 57

We can represent the recurrent network with the use of a folded model or an unfolded model:

The RNN Folded Model



The RNN Unfolded Model

In the case of a single hidden (state) layer, we will have three weight matrices to consider. Here we use the following notations:

WxW_xWx - represents the weight matrix connecting the inputs to the state layer.

WyW_yWy - represents the weight matrix connecting the state to the output.

WsW_sWs - represents the weight matrix connecting the state from the previous timestep to the state in the following timestep.

The gradient calculations for the purpose of adjusting the weight matrices are the following:

$$\frac{\partial E_N}{\partial W_y} = \frac{\partial E_N}{\partial \bar{y}_N} \frac{\partial \bar{y}_N}{\partial W_y}$$

Equation 58

$$\frac{\partial E_N}{\partial W_s} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N} \frac{\partial \bar{y}_N}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial W_s}$$

Equation 59

$$\frac{\partial E_N}{\partial W_x} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial \bar{y}_N} \frac{\partial \bar{y}_N}{\partial \bar{s}_i} \frac{\partial \bar{s}_i}{\partial W_x}$$

Equation 60

In equations _51_ and _52_ we used **Backpropagation Through Time (BPTT)** where we accumulate all of the contributions from previous timesteps.

When training RNNs using BPTT, we can choose to use mini-batches, where we update the weights in batches periodically (as opposed to once every inputs sample). We calculate the gradient for each step but do not update the weights right away. Instead, we update the weights once every fixed number of steps. This helps reduce the complexity of the training process and helps remove noise from the weight updates.

The following is the equation used for **Mini-Batch Training Using Gradient Descent**: (where δij\delta_{ij}δij represents the gradient calculated once every inputs sample and M represents the number of gradients we accumulate in the process).

$$\delta_{ij} = \frac{1}{M} \sum_{k=1}^{M} \delta_{ij_k}$$

Equation 61

If we backpropagate more than ~10 timesteps, the gradient will become too small. This phenomena is known as the **vanishing gradient problem** where the contribution of information decays geometrically over time. Therefore temporal dependencies that span many time steps will effectively be discarded by the network. **Long Short-Term Memory (LSTM)** cells were designed to specifically solve this problem.

In RNNs we can also have the opposite problem, called the **exploding gradient** problem, in which the value of the gradient grows uncontrollably. A simple solution for the exploding gradient problem is **Gradient Clipping**.

More information about Gradient Clipping can be found here.

You can concentrate on Algorithm 1 which describes the gradient clipping idea in simplicity.

From RNN to LSTM

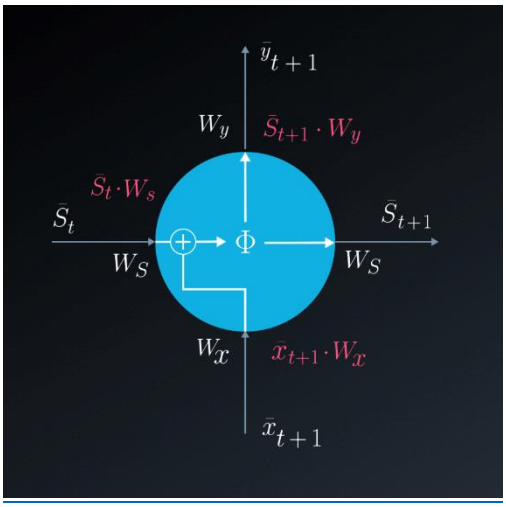
Before we take a close look at the **Long Short-Term Memory (LSTM)** cell, let's take a look at the following video:

https://www.youtube.com/watch?v=MsqybcWmzGY&feature=emb_logo

Long Short-Term Memory Cells, (LSTM) give a solution to the vanishing gradient problem, by helping us apply networks that have temporal dependencies. They were proposed in 1997 by Sepp Hochreiter and Jürgen Schmidhuber

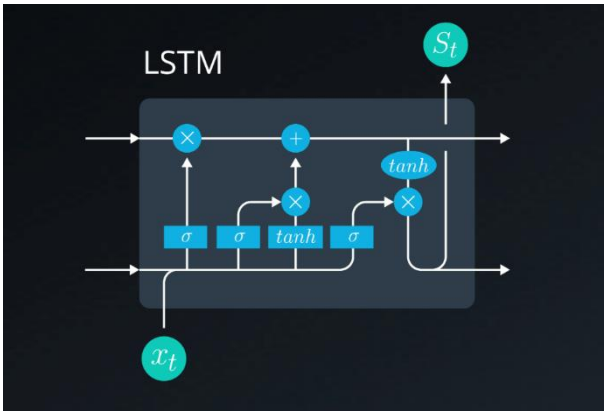
If we take a close look at the RNN neuron, we can see that we have simple linear combinations (with or without the use of an activation function). We can also see that we have a single addition.

Zooming in on the neuron, we can graphically see this in the following configuration:



Closeup Of The RNN Neuron

The **LSTM** cell is a bit more complicated. If we zoom in on the cell, we can see that the mathematical configuration is the following:



Closeup Of the LSTM Cell

The LSTM cell allows a recurrent system to learn over many time steps without the fear of losing information due to the vanishing gradient problem. It is fully differentiable, therefore gives us the option of easily using backpropagation when updating the weights.

In our next set of videos Luis will help you understand LSTMs further.

Wrap Up

In this lesson we reviewed the FFNN and its training process. We dove into RNNs, understanding the motivation behind them, when they are used, and how are they designed. We also learned how to train them using BPTT. We reviewed current applications, such as machine translation, and gave intuition to why so many other applications use RNNs, and in particular LSTMs.

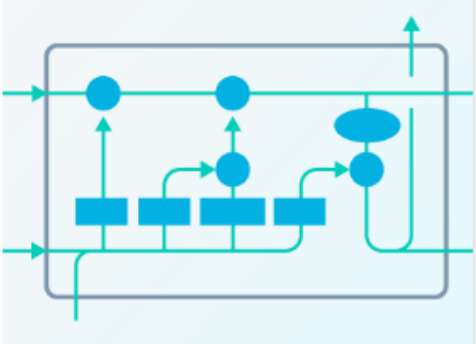
Next, Luis will guide you through the *LSTM* architecture

Lesson 4 LSTMs:

Long Short-Term Memory Networks (LSTMs)

Luis explains Long Short-Term Memory Networks (LSTM), and similar architectures which have the benefits of preserving long term memory.

VIEW LESSON →



Hi! It's Luis again!

Now that you've gone through the **Recurrent Neural Network lesson**, I'll be teaching you what an **LSTM** is. This stands for **Long Short Term Memory Networks**, and are quite useful when our neural network needs to switch between remembering recent things, and things from long time ago. But first, I want to give you some great references to study this further. There are many posts out there about LSTMs, here are a few of my favorites:

- Chris Olah's LSTM post
- Edwin Chen's LSTM post
- Andrej Karpathy's lecture on RNNs and LSTMs from CS231n

So, let's dig in!

RNN vs LSTM

https://www.youtube.com/watch?time_continue=11&v=70MgF-IwAr8&feature=emb_logo

Basics of LSTM

https://www.youtube.com/watch?time_continue=35&v=gjb68a4XsqE&feature=emb_logo

Architecture of LSTM

https://www.youtube.com/watch?time_continue=9&v=ycwthhdx8ws&feature=emb_logo

- Notebook: LSTM structure and part of speech tagging, pytorch

The Learn Gate

he output of the *Learn Gate* is NtitN_ti_t*Nti*t where:

$$N_t = \tanh(W_n[STM_{t-1}, E_t] + b_n)$$

$$i_t = \sigma(W_i[STM_{t-1}, E_t] + b_i)$$

Equation 1

https://www.youtube.com/watch?v=aVHVI7ovbHY&feature=emb_logo

Forget Gate

The output of the *Forget Gate* is LTMt−1ftLTM_{t-1}f_t*LTMt*−1*ft* where:

$$f_t = \sigma(W_f[STM_{t-1}, E_t] + b_f)$$

Equation 2

https://www.youtube.com/watch?v=iWxpfxLUPSU&feature=emb_logo

The Remember Gate

https://www.youtube.com/watch?time_continue=3&v=0qlm86HaXuU&feature=emb_logo

The Use Gate

https://www.youtube.com/watch?time_continue=3&v=0qlm86HaXuU&feature=emb_logo

At 00:27 : Luis refers to obtaining New Short Term Memory instead it's **New Long Term Memory**.

The output of the *Use Gate* is UtVtU_t V_t*UtVt* where:

$$U_t = \tanh(W_u LTM_{t-1} f_t + b_u)$$

$$V_t = \sigma(W_v[STM_{t-1}, E_t] + b_v)$$

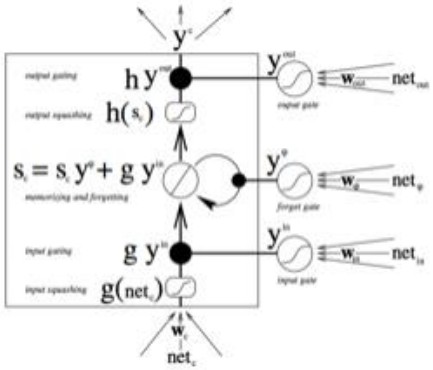
Equation 4

Putting it All Together

https://www.youtube.com/watch?time_continue=9&v=IF8FlKW-Zo0&feature=emb_logo

If you would like to deepen your knowledge even more, go over the following tutorial. Focus on the overview titled: **Long Short-Term Memory Units (LSTMs).**

If you are feeling confident enough, skip the overview and jump right into our next question:



The LSTM cell- taken form the Deep Learning tutorial

QUIZ QUESTION

The illustration above is of a LSTM cell. What would be the values of the three gates in situations where the cell retains information for a long period, without accepting a new input or producing an output?

The inputs gate: close to 0; the forget gate: close to 1; the output gate: close to 0

- Notebook: LSTM for Part of Speech Tagging

Character-Level RNN

We will be building and training a basic character-level RNN, specifically an LSTM which reads words as a series of characters and outputs a prediction and "hidden state" at each time step, feeding its previous hidden state into each next step. Next, we'll have our team product lead, Mat, talk about what a character-level RNN can be expected to do.

https://www.youtube.com/watch?v=m_S2hs6-j5w&feature=emb_logo

Sequences of Data

The core reason that recurrent neural networks are exciting is that they allow us to operate over sequences of vectors: sequences in the input, the output, or in some cases, both!

Most RNN's expect to see a sequence of data in a fixed batch size, much like we've seen images processed in fixed batch sizes. The batches of data affect how the hidden state of an RNN train, so next you'll learn more about batching data.

https://www.youtube.com/watch?v=pdSr5F-9qE0&feature=emb_logo

- Notebook: Character-Level LSTM

Other architectures

https://www.youtube.com/watch?time_continue=10&v=MsxFDuYlTuQ&feature=emb_logo

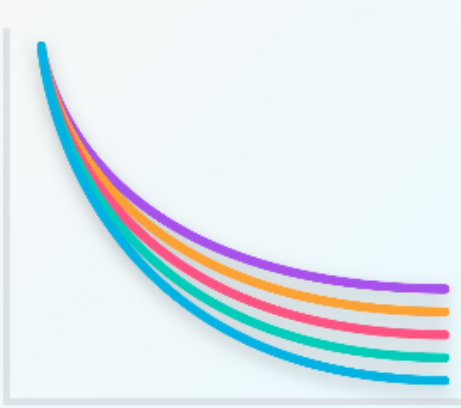
Lesson 5: Hyperparameters

LESSON 5

Hyperparameters

Learn about a number of different hyperparameters that are used in defining and training deep learning models. We'll discuss starting values and intuitions for tuning each hyperparameter.

VIEW LESSON →



Introduction

https://www.youtube.com/watch?time_continue=1&v=erwnzFD7AeE&feature=emb_logo

Learning Rate

https://www.youtube.com/watch?time_continue=5&v=HLMjeDez7ps&feature=emb_logo

Say you're training a model. If the output from the training process looks as shown below, what action would you take on the learning rate to improve the training?

```
Epoch 1, Batch 1, Training Error: 8.4181
Epoch 1, Batch 2, Training Error: 8.4177
Epoch 1, Batch 3, Training Error: 8.4177
Epoch 1, Batch 4, Training Error: 8.4173
Epoch 1, Batch 5, Training Error: 8.4169
```

- ○ Decrease the learning rate
- ○ Try again using the same learning rate
- ✓ Increase the learning rate

Say you're training a model. If the output from the training process looks as shown below, what action would you take on the learning rate to improve the training?

```
Epoch 1, Batch 1, Training Error: 8.71
Epoch 1, Batch 2, Training Error: 3.25
Epoch 1, Batch 3, Training Error: 4.93
Epoch 1, Batch 4, Training Error: 3.30
Epoch 1, Batch 5, Training Error: 4.82
```

- ✓ Decrease the learning rate
- ✓ Use an adaptive learning rate

Minibatch Size

https://www.youtube.com/watch?time_continue=4&v=GrrO1NFxaW8&feature=emb_logo

Number of Training Iterations / Epochs

https://www.youtube.com/watch?time_continue=45&v=TTdHpSb4DV8&feature=emb_logo

Number of Hidden Units / Layers

https://www.youtube.com/watch?v=IkGAIQH5wH8&feature=emb_logo

RNN Hyperparameters

https://www.youtube.com/watch?v=yQvnv7l_aUo&feature=emb_logo

LSTM Vs GRU

"These results clearly indicate the advantages of the gating units over the more traditional recurrent units. Convergence is often faster, and the final solutions tend to be better. However, our results are not conclusive in comparing the LSTM and the GRU, which suggests that the choice of the type of gated recurrent unit may depend heavily on the dataset and corresponding task."

Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling by Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, Yoshua Bengio

"The GRU outperformed the LSTM on all tasks with the exception of language modelling"

An Empirical Exploration of Recurrent Network Architectures by Rafal Jozefowicz, Wojciech Zaremba, Ilya Sutskever

"Our consistent finding is that depth of at least two is beneficial. However, between two and three layers our results are mixed. Additionally, the results are mixed between the LSTM and the GRU, but both significantly outperform the RNN."

Visualizing and Understanding Recurrent Networks by Andrej Karpathy, Justin Johnson, Li Fei-Fei

"Which of these variants is best? Do the differences matter? Greff, et al. (2015) do a nice comparison of popular variants, finding that they're all about the same. Jozefowicz, et al. (2015) tested more than ten thousand RNN architectures, finding some that worked better than LSTMs on certain tasks."

Understanding LSTM Networks by Chris Olah

"In our [Neural Machine Translation] experiments, LSTM cells consistently outperformed GRU cells. Since the computational bottleneck in our architecture is the softmax operation we did not observe large difference in training speed between LSTM and GRU cells. Somewhat to our surprise, we found that the vanilla decoder is unable to learn nearly as well as the gated variant."

Massive Exploration of Neural Machine Translation Architectures by Denny Britz, Anna Goldie, Minh-Thang Luong, Quoc Le

Example RNN Architectures

Application	Cell	Layers	Size	Vocabulary	Embedding Size	Learning Rate	
Speech Recognition (large vocabulary)	LSTM	5, 7	600, 1000	82K, 500K	--	--	paper
Speech Recognition	LSTM	1, 3, 5	250	--	--	0.001	paper
Machine Translation (seq2seq)	LSTM	4	1000	Source: 160K, Target: 80K	1,000	--	paper

Application	Cell	Layers	Size	Vocabulary	Embedding Size	Learning Rate	
Image Captioning	LSTM	--	512	--	512	(fixed)	paper
Image Generation	LSTM	--	256, 400, 800	--	--	--	paper
Question Answering	LSTM	2	500	--	300	--	pdf
Text Summarization	GRU		200	Source: 119K, Target: 68K	100	0.001	pdf

QUESTION 1 OF 2

How do Long Short Term Memory (LSTM) cells and Gated Recurrent Unit (GRU) cells compare?

- LSTMs are superior to GRUs in every way
- GRUs are superior to LSTMs in every way
- ✓ It depends.. It's probably worth it to compare the two on my task and dataset.

SUBMIT

QUESTION 2 OF 2

Which embedding size looks more reasonable for the majority of cases?

- ✓ 500

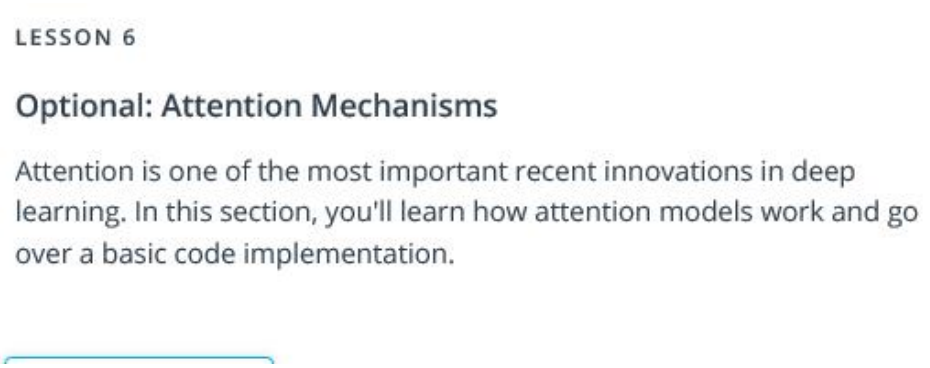
If you want to learn more about hyperparameters, these are some great resources on the topic:

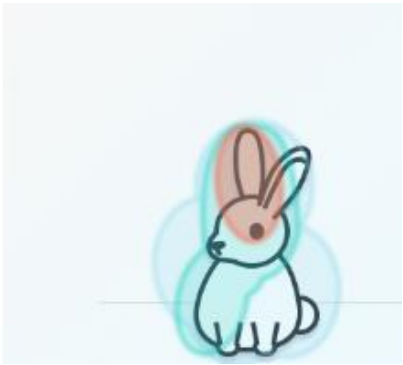
- **Practical recommendations for gradient-based training of deep architectures** by Yoshua Bengio
- **Deep Learning book - chapter 11.4: Selecting Hyperparameters** by Ian Goodfellow, Yoshua Bengio, Aaron Courville
- **Neural Networks and Deep Learning book - Chapter 3: How to choose a neural network's hyper-parameters?** by Michael Nielsen
- **Efficient BackProp (pdf)** by Yann LeCun

More specialized sources:

- **How to Generate a Good Word Embedding?** by Siwei Lai, Kang Liu, Liheng Xu, Jun Zhao
- **Systematic evaluation of CNN advances on the ImageNet** by Dmytro Mishkin, Nikolay Sergievskiy, Jiri Matas
- **Visualizing and Understanding Recurrent Networks** by Andrej Karpathy, Justin Johnson, Li Fei-Fei

Lesson 6: Attention Mechanism





Sequence to Sequence Recap

https://www.youtube.com/watch?time_continue=6&v=MRPHIPR0pGE&feature=emb_logo

Encoders and Decoders

Sequence to Sequence Models

Before we jump into learning about attention models, let's recap what you've learned about sequence to sequence models. We know that RNNs excel at using and generating sequential data, and sequence to sequence models can be used in a variety of applications!

https://www.youtube.com/watch?v=tDJBDwriJYQ&feature=emb_logo

https://www.youtube.com/watch?v=dkHdEAJnV_w&feature=emb_logo

Encoders and Decoders

The encoder and decoder do not have to be RNNs; they can be CNNs too!

In the example above, an LSTM is used to generate a sequence of words; LSTMs "remember" by keeping track of the input words that they see and their own hidden state.

In computer vision, we can use this kind of encoder-decoder model to generate words or captions for an input image or even to generate an image from a sequence of input words. We'll focus on the first case: generating captions for images, and you'll learn more about caption generation in the next lesson. For now know that we can input an image into a CNN (encoder) and generate a descriptive caption for that image using an LSTM (decoder).

Elective: Text Sentiment Analysis

If you would like more practice with analyzing sequences of words with a simple network, now would be a great time to check out the elective section: Text Sentiment Analysis. In this section, Andrew Trask teaches you how to convert words into vectors and then analyze the sentiment of these vectors. He goes through constructing and tuning a model *and* addresses some common errors in text analysis. This section does not contain material that is required to complete this program or the project in this section, but it is interesting and you may find it useful!

Check is a text review is a positive or negative review!

Sequence to Sequence Recap

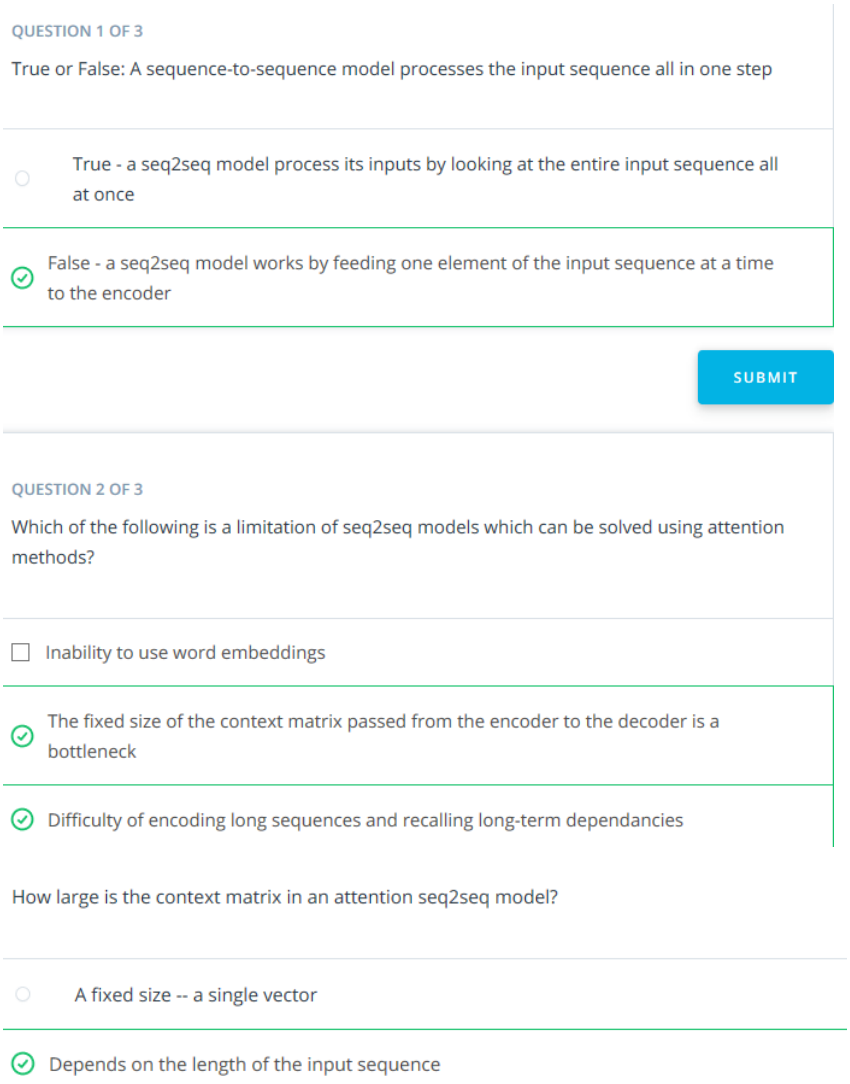
https://www.youtube.com/watch?time_continue=5&v=MRPHIPR0pGE&feature=emb_logo

Encoding-- Attention Overview

https://www.youtube.com/watch?time_continue=6&v=IctAnMaVUKc&feature=emb_logo

Decoding-- Attention Overview

https://www.youtube.com/watch?v=DJxiPd585GY&feature=emb_logo

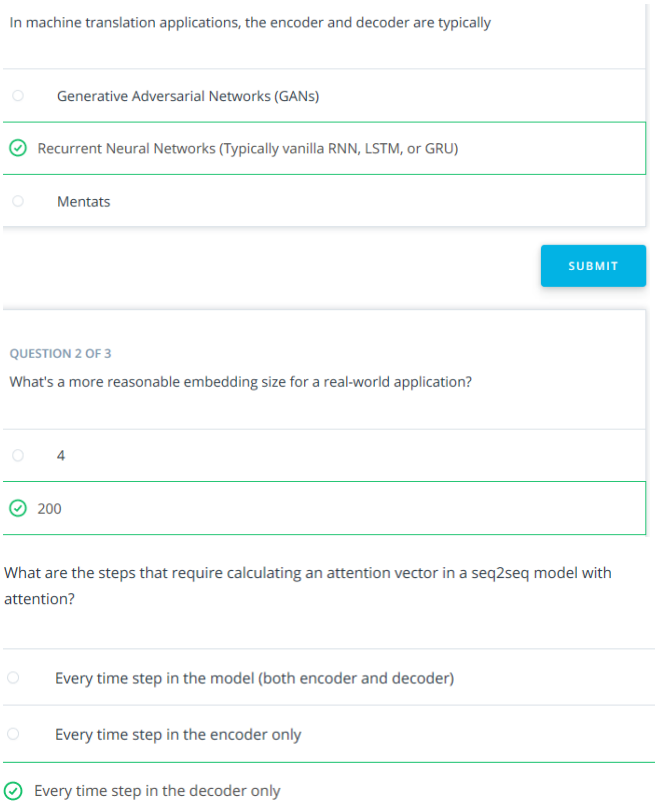


Attention Encoder

https://www.youtube.com/watch?time_continue=5&v=sphe9LDT4rA&feature=emb_logo

Attention Decoder

https://www.youtube.com/watch?time_continue=3&v=5mMz6nN9_Ss&feature=emb_logo



Bahdanau and Luong Attention

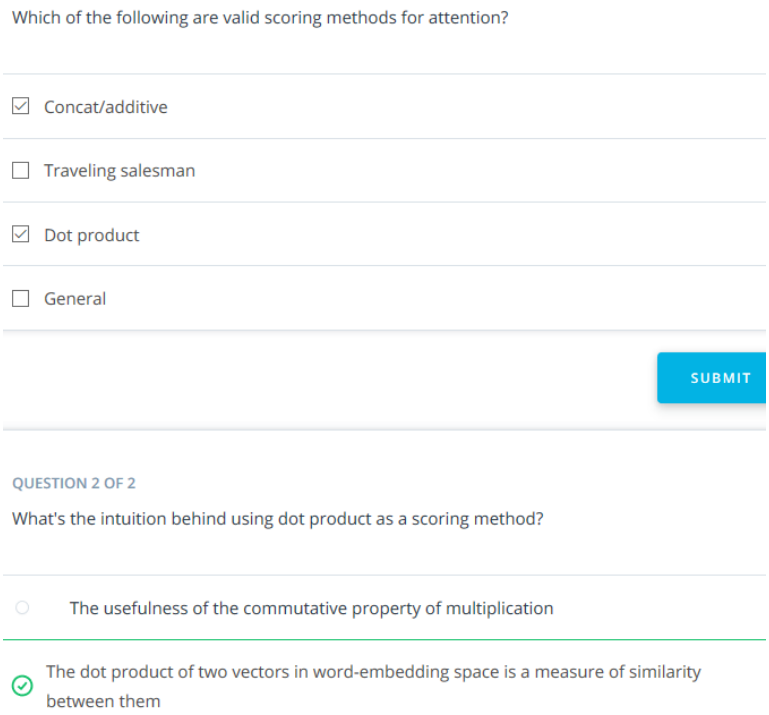
https://www.youtube.com/watch?time_continue=5&v=2eqIUDjefNg&feature=emb_logo

Multiplicative Attention

https://www.youtube.com/watch?time_continue=355&v=1-OwCgrx1eQ&feature=emb_logo

Additive Attention

https://www.youtube.com/watch?time_continue=3&v=93VfVWZ-IvY&feature=emb_logo



Computer Vision Applications

https://www.youtube.com/watch?time_continue=5&v=bhWwc4BYTYc&feature=emb_logo

Other Attention Methods

https://www.youtube.com/watch?time_continue=4&v=VmsR9FVpQiM&feature=emb_logo

The Transformer and Self-Attention

https://www.youtube.com/watch?time_continue=26&v=F-XN72bQiMQ&feature=emb_logo

- Notebook: Attention Basics

Outro

Great job completing the deep learning attention section!

You should have a greater idea about how information is represented in data and how you can programmatically represent *the most important information* in that data.

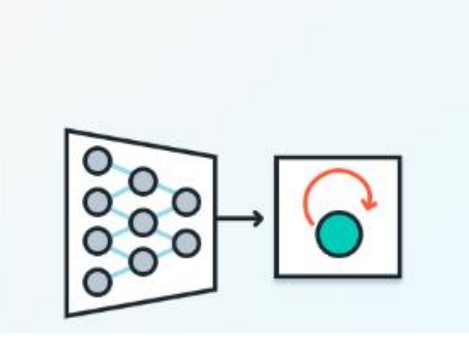
As you move on to the lesson about Image Captioning, please keep in mind the flexibility of an encoder and decoder model. The decoder portion is perhaps the most difficult as you have to decide how to embed image and word vectors into a shape that an LSTM can take as input and learn from. Good luck!

Lesson 7: Image Captioning

LESSON 7

Image Captioning

Learn how to combine CNNs and RNNs to build a complex, automatic image captioning model.



Introduction:

https://www.youtube.com/watch?v=dobNslC2y-o&feature=emb_logo

Leveraging Neural Networks

https://www.youtube.com/watch?time_continue=7&v=thClOUTiNxY&feature=emb_logo

Captions and the COCO Dataset

https://www.youtube.com/watch?time_continue=5&v=DMmJs1w1n7A&feature=emb_logo

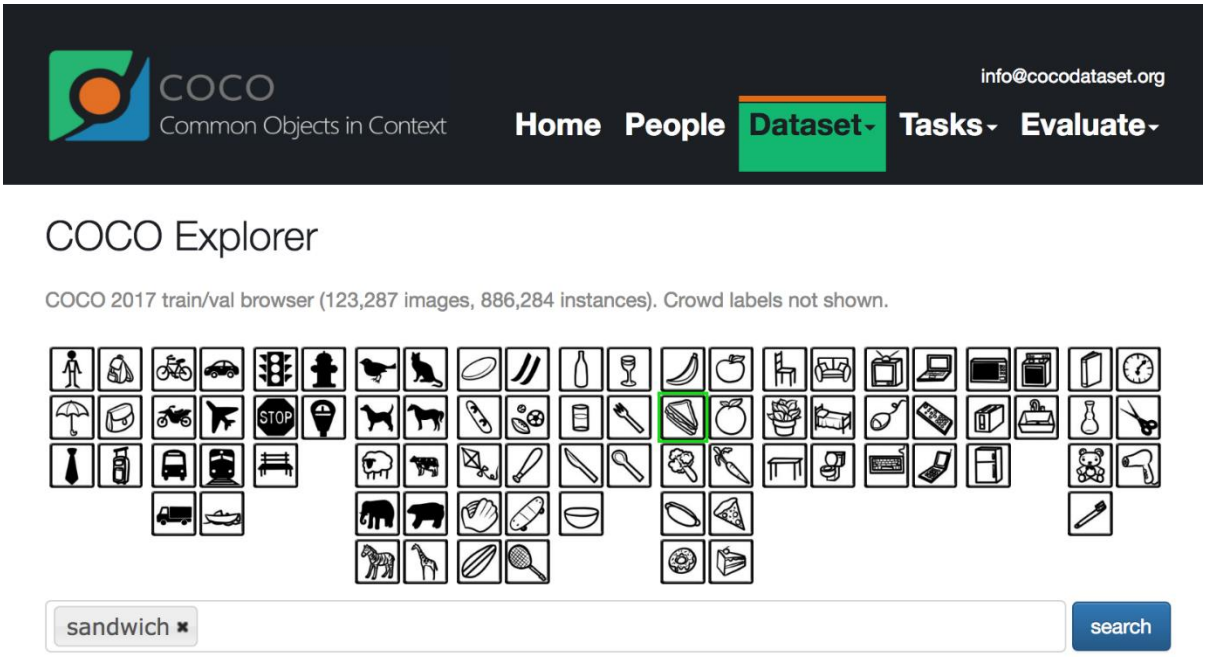
COCO Dataset

The COCO dataset is one of the largest, publicly available image datasets and it is meant to represent realistic scenes. What I mean by this is that COCO does not overly pre-process images, instead these images come in a variety of shapes with a variety of objects and environment/lighting conditions that closely represent what you might get if you compiled images from many different cameras around the world.

To explore the dataset, you can check out the **dataset website**.

Explore

Click on the explore tab and you should see a search bar that looks like the image below. Try selecting an object by it's icon and clicking search!



A sandwich is selected by icon.

You can select or deselect multiple objects by clicking on their corresponding icon. Below are some examples for what a sandwich search turned up! You can see that the initial results show colored overlays over objects like sandwiches and people and the objects come in different sizes and orientations.

COCO sandwich detections

Captions

COCO is a richly labeled dataset; it comes with class labels, labels for segments of an image, *and* a set of captions for a given image. To see the captions for an image, select the text icon that is above the image in a toolbar. Click on the other options and see what the result is.

the counter of a restaurant with food displayed.
a store has their display of food with workers behind it
a few people that are out front of a cafe
a man prepares food behind a counter with two others.
getting a lesson to how to prepare the food.

Example captions for an image of people at a sandwich counter.

When we actually train our model to generate captions, we'll be using these images as input and sampling *one* caption from a set of captions for each image to train on.

Supporting Materials

Creating COCO, paper

CNN-RNN Model

https://www.youtube.com/watch?time_continue=3&v=n7kdMiX1Xz8&feature=emb_logo

The Glue, Feature Vector

https://www.youtube.com/watch?time_continue=1&v=u2ZdcUDnHm0&feature=emb_logo

Tokenizing Captions

https://www.youtube.com/watch?v=aeEFb0eSzJ8&feature=emb_logo

Words to Vectors

At this point, we know that you cannot directly feed words into an LSTM and expect it to be able to train or produce the correct output. These words first must be turned into a numerical representation so that a network can use normal loss functions and optimizers to calculate how "close" a predicted word and ground truth word (from a known, training caption) are. So, we typically turn a sequence of words into a sequence of numerical values; a vector of numbers where each number maps to a specific word in our vocabulary.

https://www.youtube.com/watch?v=4Ieotbeh4u8&feature=emb_logo

To process words and create a vocabulary, we'll be using the Python text processing toolkit: NLTK. in the below video, we have one of our content developers, Arpan, explain the concept of word tokenization with NLTK.

Reference:

- nltk.tokenize package: **http://www.nltk.org/api/nltk.tokenize.html**

Later, you'll see how we take a tokenized representation of a caption to create a Python **dictionary** that maps unique words in our captions dataset to unique integers.

RNN Training

https://www.youtube.com/watch?time_continue=43&v=P-tHxD7kRmA&feature=emb_logo

Video Captioning

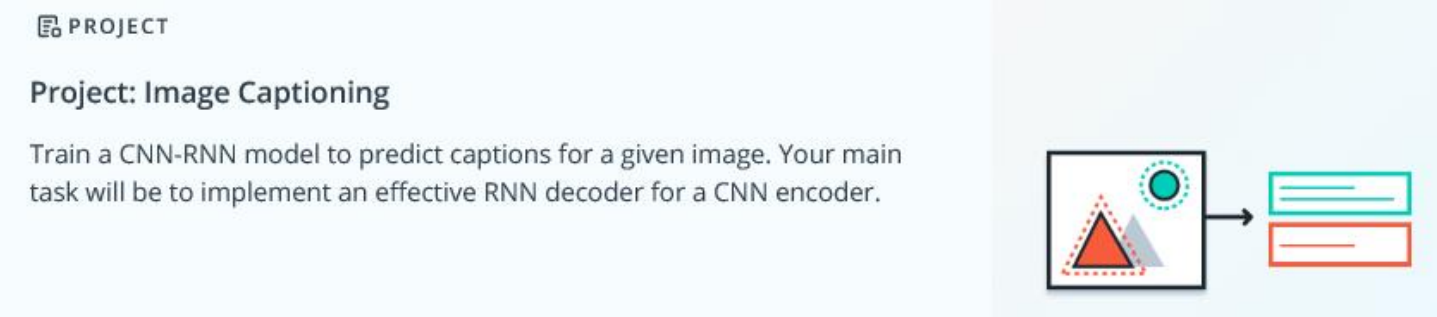
https://www.youtube.com/watch?time_continue=4&v=I_m9JyKTfbQ&feature=emb_logo

Want to Learn more from Kelvin?

Kelvin has worked with us on one more high-level lesson about fully-convolutional networks (FCN's). To learn more from him and about a complex deep learning model, I suggest you check out the elective section: **Elective: More Deep Learning Models** and go to Fully-Convolutional Networks!

https://www.youtube.com/watch?time_continue=16&v=MsxNRaYTNSk&feature=emb_logo

Project: Image Captioning



Project Overview

In this project, you will create a neural network architecture to automatically generate captions from images.

After using the Microsoft Common Objects in COntext **(MS COCO) dataset** to train your network, you will test your network on novel images!

Project Instructions

The project is structured as a series of Jupyter notebooks that are designed to be completed in sequential order:

- 0_Dataset.ipynb
- 1_Preliminaries.ipynb
- 2_Training.ipynb
- 3_Inference.ipynb

You can find these notebooks in the Udacity workspace that appears in the concept titled **Project: Image Captioning**. This workspace provides a Jupyter notebook server directly in your browser.

You can read more about workspaces (and how to toggle GPU support) in the following concept (**Introduction to GPU Workspaces**). This concept will show you how to toggle GPU support in the workspace.

You MUST enable GPU mode for this project and submit your project after you complete the code in the workspace.

A completely trained model is expected to take between 5-12 hours to train well on a GPU; it is suggested that you look at early patterns in loss (what happens in the first hour or so of training) as you make changes to your model, so that you only have to spend this large amount of time training your *final* model.

Should you have any questions as you go, please post in the Student Hub!

Evaluation

Your project will be reviewed by a Udacity reviewer against the CNN project **rubric**. Review this rubric thoroughly, and self-evaluate your project before submission. As in the first project, **you'll find that only some of the notebooks and files are graded**. All criteria found in the rubric must meet specifications for you to pass.

Ready to submit your project?

It is a known issue that the COCO dataset is not well supported for download on Windows, and so you are required to complete the project in the GPU workspace. This will also allow you to bypass setting up an AWS account and downloading the dataset, locally. If you would like to refer to the project code, you may look at **this version (in PyTorch 0.4.0)** at the linked Github repo.

Once you've completed your project, you may **only submit from the workspace** for this project, **linked here**. Click Submit, a button that appears on the bottom right of the *workspace*, to submit your project.

For submitting from the workspace, directly, please make sure that you have deleted any large files and model checkpoints in your notebook directory before submission** or your project file may be too large to download and grade.

GPU Workspaces

Note: To load the COCO data in the workspace, you *must have GPU mode enabled*.

In the next section, you'll learn more about these types of workspaces.

LSTM Decoder

In the project, we pass all our inputs as a sequence to an LSTM. A sequence looks like this: first a feature vector that is extracted from an input image, then a start word, then the next word, the next word, and so on!

Embedding Dimension

The LSTM is defined such that, as it sequentially looks at inputs, it expects that each individual input in a sequence is of a **consistent size** and so we *embed* the feature vector and each word so that they are `embed_size`.

Sequential Inputs

So, an LSTM looks at inputs sequentially. In PyTorch, there are two ways to do this.

The first is pretty intuitive: for all the inputs in a sequence, which in this case would be a feature from an image, a start word, the next word, the next word, and so on (until the end of a sequence/batch), you loop through each input like so:

```
for i in inputs:
    # Step through the sequence one element at a time.
    # after each step, hidden contains the hidden state.
    out, hidden = lstm(i.view(1, 1, -1), hidden)
```

The second approach, which this project uses, is to **give the LSTM our *entire sequence*** and have it produce a set of outputs and the last hidden state:

```
# the first value returned by LSTM is all of the hidden states throughout
# the sequence. the second is just the most recent hidden state

# Add the extra 2nd dimension
inputs = torch.cat(inputs).view(len(inputs), 1, -1)
hidden = (torch.randn(1, 1, 3), torch.randn(1, 1, 3))  # clean out hidden state
out, hidden = lstm(inputs, hidden)
```

Object Tracking and Localization

Lesson 1: introduction to motion

Object Tracking

https://www.youtube.com/watch?time_continue=5&v=ciEo6PyMTeM&feature=emb_logo

Localization Intro

https://www.youtube.com/watch?time_continue=1&v=QkVyEMTUkAw&feature=emb_logo

Motion

https://www.youtube.com/watch?time_continue=5&v=A-QJf04LBb0&feature=emb_logo

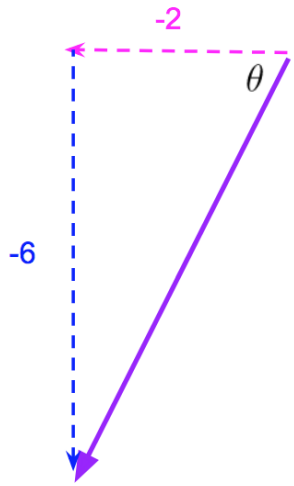
Optical Flow

https://www.youtube.com/watch?v=TOS8UJwCtTg&feature=emb_logo

Motion Vectors

A motion vector for a 2D image, has an x and y component (u, v). A motion vector for any point starts with the location of the point as the origin of the vector and it's destination as the end of the vector (where the arrow point is).

All vectors have a **direction** and a **magnitude** and you can see more examples of vectors describeing motion, **here**.



A vector with (u,v) = (-2, -6)

Looking at the motion vector above with (u,v) = (-2,-6), what is it's magnitude? **(Please round to the nearest 3 digits, ex. 1.234)**

6.324

Reset

Question 2 of 2

What is the *direction* of this vector? Note that a vector pointing straight to the right has a direction of 0 degrees.

-

-71.6 degrees

-

71.6 degrees

-

180 degrees

- 251.6 degrees

Brightness Constancy Assumption

https://www.youtube.com/watch?time_continue=10&v=GHz9Yzt5tro&feature=emb_logo

- Notebook: Optical Flow and Motion Vectors

Tracking Features

Where is Optical Flow Used?

Optical Flow is a well-researched and fast algorithm and it is used in a lot of tracking technology, today! One such example is the **NVIDIA Redtail drone**, which uses optical flow to track surrounding objects in a video stream.

https://www.youtube.com/watch?v=uFf6IZ5MxgU&feature=emb_logo

Next: Localization

This introductory lesson was meant to give you a starting idea of how motion is represented and how one can go about estimating the motion of an object. Most of this section will be about tracking and localizing moving objects!

Next, you'll see how to tackle a complex problem: localization. Localization is all about finding out exactly where an object is in an environment and then tracking it over time as it moves (and in the case of a robot) as it gathers sensor measurements via camera, radar, LiDAR, or other sensors.

Localization is a key concept in the field of autonomous vehicles, so let's dive in!

Lesson 2: Robot Localization

LESSON 2

Robot Localization

Learn to implement a Bayesian filter to locate a robot in space and represent uncertainty in robot motion.

Review Probability

Before Sebastian teaches you about localizing a robot, in the next couple concepts, we'll review how to represent uncertainty in robot motion and sensors. The next couple sections will include multiple videos and quizzes to check your understanding of probability and probability distributions. These are meant to prepare you for this lesson, so make sure to read this material carefully to become familiar with terminology and mathematical representations of uncertainty!

Probability Review

Before learning more about uncertainty in robot localization, let's review how to mathematically represent uncertainty using probability!

A classic example of an event with some certainty associated with it is a coin flip. A typical coin has two sides: heads and tails. Without me telling you anything else, what chance do you think a coin like this has of flipping and landing heads up?

[Heads and tails of a coin.]

Question 1 of 3

If you flip a coin with two sides (heads and tails), what is the probability, P(H), that the coin will land heads up? (1= 100%, 0 = 0% chance)

-

P(H) = 0.3

- P(H) = 0.5
-

P(H) = 0.75

-

P(H) = 1.0

Submit

Question 2 of 3

When you flip a coin and it comes up *heads*, do you think it's more likely that the NEXT flip will come up *tails*? That is, does one flip affect the probability that another flip will be heads or tails?

-

Yes

- No : ANS

Submit

Formal Definition of Probability

The probability of an event, X, occurring is P(X). The value of P(X) must fall in a range of 0 to 1.

- 0 <= P(X) <= 1

An event, X, can have multiple outcomes which we might call X1, X2, .. and so on; the probabilities for all outcomes of X must add up to one. For example, say there are two possible outcomes, X1 and X2:

- If P(X1) = 0.2 then P(X2) = 0.8 because all possible outcomes must sum to 1.

Terminology

Independent Events

Events like coin flips are known as **independent events**; this means that the probability of a single flip does not affect the probability of another flip; P(H) will be 0.5 for each fair coin flip. When flipping a coin multiple times, each flip is an independent event because one flip does not affect the probability that another flip will land heads up or tails up.

Dependent Events

When two events are said to be *dependent*, the probability of one event occurring influences the likelihood of the other event. For example, say you are more likely to go outside if it's sunny weather. If the probability of sunny weather is low on a given day, the probability of you going outside will decrease as well, so the probability of going outside is dependent on the probability of sunny weather.

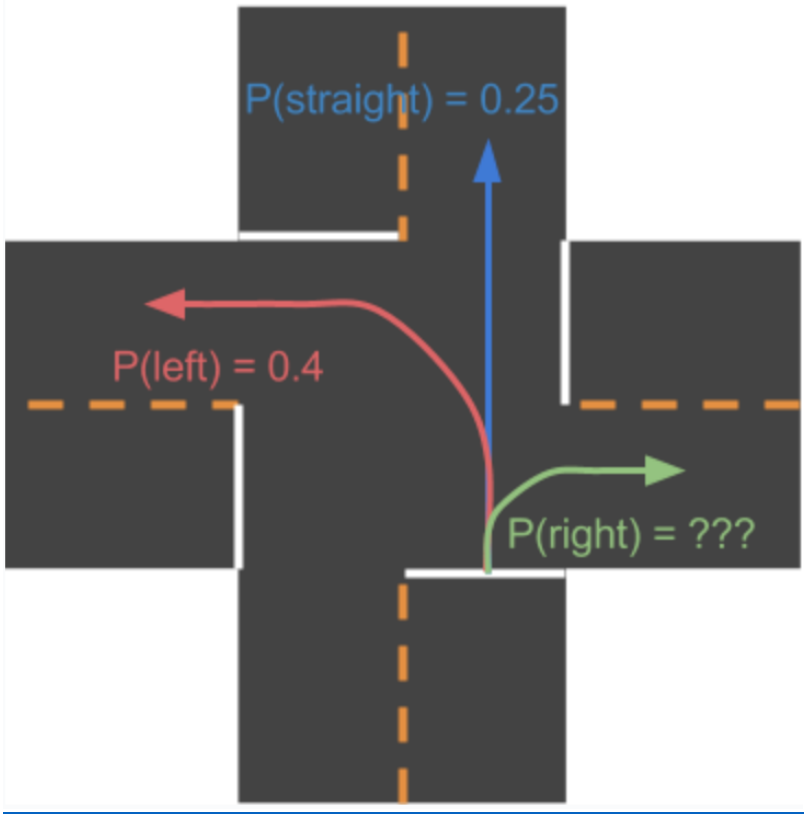
Joint Probability

The probability that two or more independent events will occur together (in the same time frame) is called a **joint probability**, and it is calculated by multiplying the probabilities of each independent event together. For example, the probability that you will flip a coin and it will lands heads up two times in a row can be calculated as follows:

- The probability of a coin flipping heads up, P(H) = 0.5
- The joint probability of two events (a coin landing heads up) happening in a row, is the probability of the first event times the probability of the second event: P(H)*P(H) = (0.5)*(0.5) = 0.25

Quantifying Certainty (and Uncertainty)

When we talk about being certain that a robot is at a certain location (x, y), moving a certain direction, or sensing a certain environment, we can quantify that certainty using probabilistic quantities. Sensor measurements and movement all have some uncertainty associated with them (ex. a speedometer that reads 50mph may be off by a few mph, depending on whether a car is moving up or down hill).



At a particular intersection, cars can either:

1. Turn left
2. Go straight
3. Turn right

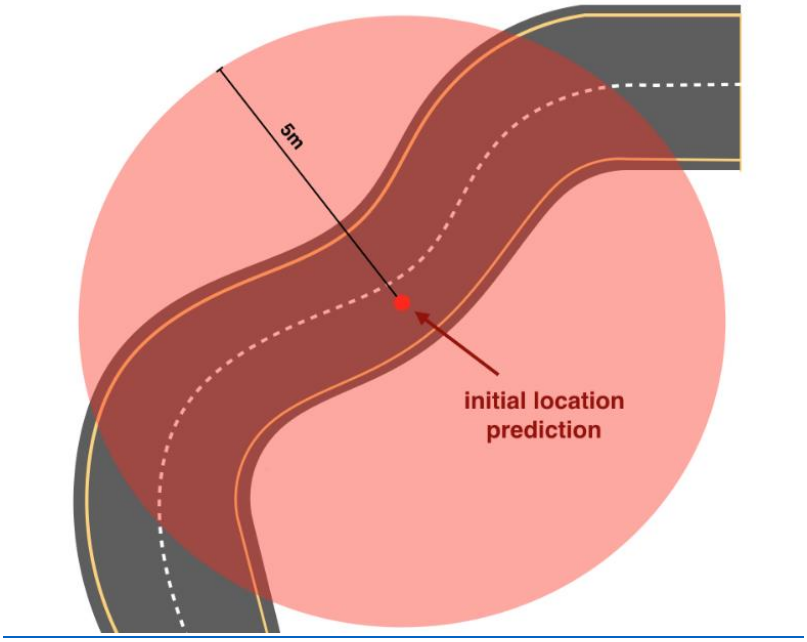
Cars tend to turn left with a probability of 0.4 and go straight with a probability of 0.25. What is the probability that a car turns right?

0.35

Bayes' Rule

Bayes' Rule is extremely important in robotics and it can be summarized in one sentence: given an initial prediction, if we gather additional data (data that our initial prediction depends on), we can improve that prediction!

Initial Scenario



Map of the road and the initial location prediction.

We know a little bit about the map of the road that a car is on (pictured above). We also have an initial GPS measurement; the GPS signal says the car is at the red dot. However, this GPS measurement is inaccurate up to about 5 meters. So, the vehicle could be located anywhere within a 5m radius circle around the dot.

Sensors

Then we gather data from the car's sensors. Self-driving cars mainly use three types of sensors to observe the world:

- **Camera**, which records video,
- **Lidar**, which is a light-based sensor, and
- **Radar**, which uses radio waves.

All of these sensors detect surrounding objects and scenery.

Autonomous cars also have lots of **internal sensors** that measure things like the speed and direction of the car's movement, the orientation of its wheels, and even the internal temperature of the car!

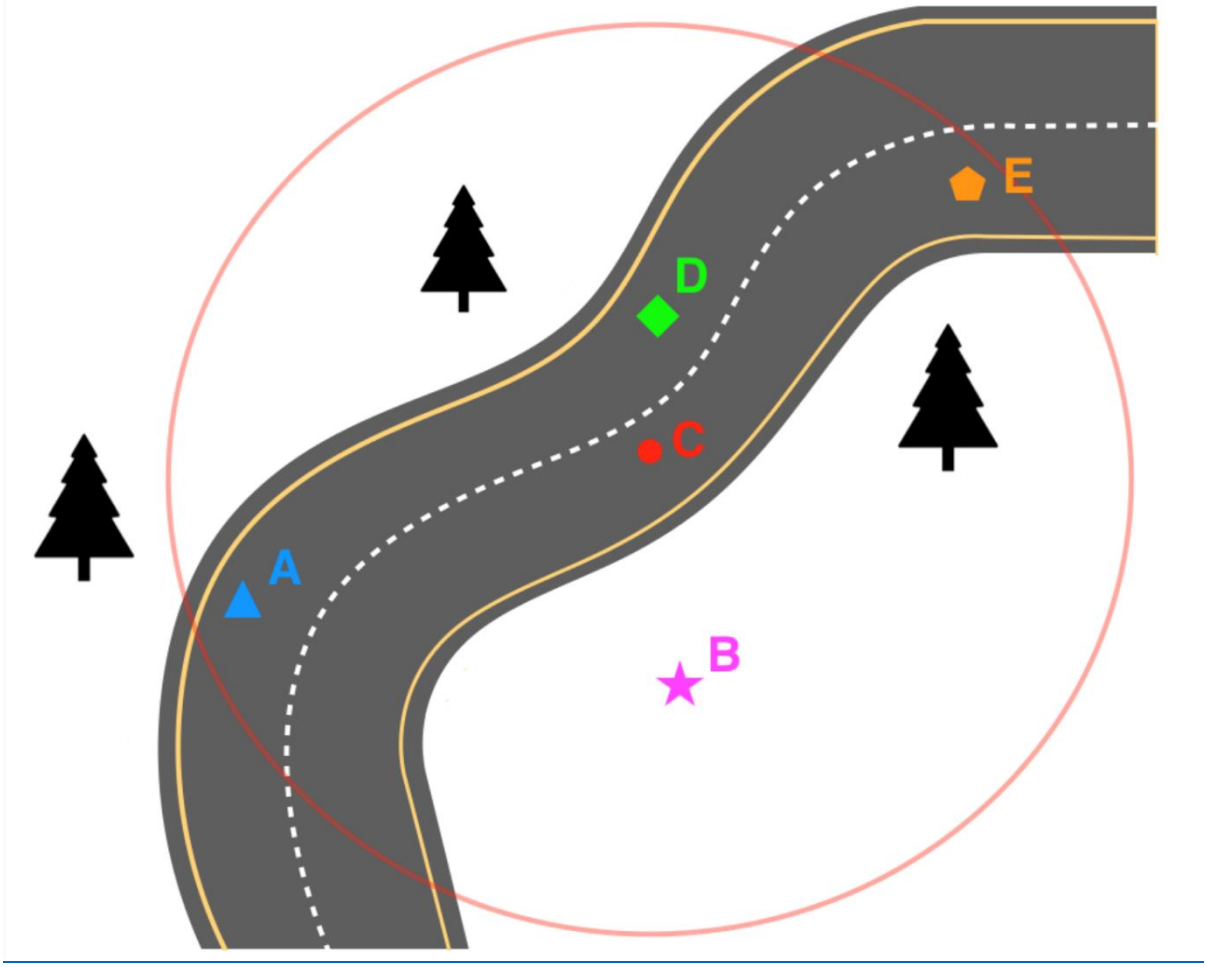
Sensor Measurements

Suppose that our sensors detect some details about the terrain and the way our car is moving, specifically:

- The car could be anywhere within the GPS 5m radius circle,
- The car is moving upwards on this road,
- There is a tree to the left of our car, and

- The car's wheels are pointing to the right.

Knowing only these sensor measurements, examine the map below and answer the following quiz question.



[Road map with additional sensor data]

Quiz Question

After considering the sensor measurements and the initial location prediction, which point on the above map is the best estimate for our car's location?

ANS: A

Reducing Uncertainty

https://www.youtube.com/watch?time_continue=5&v=vhl-SADfti8&feature=emb_logo

What is a Probability Distribution?

https://www.youtube.com/watch?v=gqPjMDVFvNg&feature=emb_logo

Probability distributions allow you to represent the probability of an event using a mathematical equation. Like any mathematical equation:

- probability distributions **can be visualized** using a graph especially in 2-dimensional cases.
- probability distributions **can be worked with using algebra, linear algebra and calculus**.

These distributions make it *much* easier to understand and summarize the probability of a system whether that system be a coin flip experiment or the location of an autonomous vehicle.

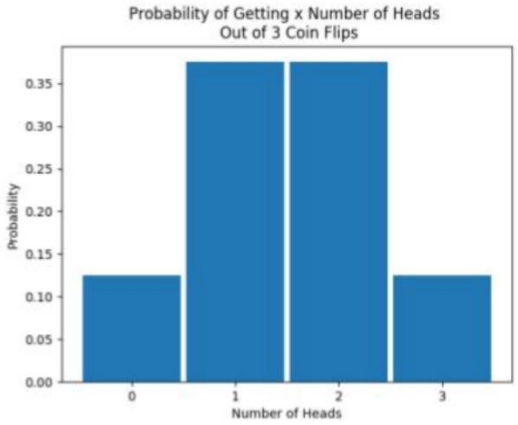
Types of Probability Distributions

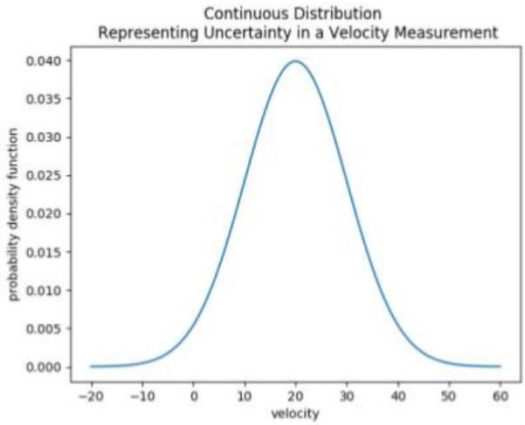
Probability distributions are really helpful for understanding the probability of a system. Looking at the big pictures, there are two types of probability distributions:

- discrete probability distributions
- continuous probability distributions

Before we get into the details about what discrete and continuous mean, take a look at these two visualizations below. The first image shows a discrete probability distribution and the second a continuous probability distribution. What is similar and what is different about each visualization?

https://www.youtube.com/watch?time_continue=106&v=PglBg4eb_5M&feature=emb_logo





Discrete Distribution (left) and Continuous Distribution (right).

Quiz Question

Based on the visualizations, which of the following are true about the discrete probability distribution versus the continuous probability distribution?

- The x-axis represents the main variable/event of interest for both visualizations.
- In the discrete visualization, the x-axis variable can only take on certain values such as 1, 2 or 3.
- ☐

In the discrete visualization, the x-axis variable can take on any value such as 3.4 or 0.5.

- In the continuous visualization, the x-axis variable can take on any real number value from -infinity to +infinity.

More terminology

- **Prior** - a prior probability distribution of an uncertain quantity, such as the location of a self-driving car on a road. This is the probability distribution that would express one's beliefs about the car's location **before** some sensor measurements or other evidence is taken into account.
- **Posterior** - the probability distribution of an uncertain quantity, **after** some evidence (like sensor measurements) have been taken into account.

And you'll learn more about this terminology in the upcoming videos!

Localization

https://www.youtube.com/watch?time_continue=1&v=OB6GZxfvESw&feature=emb_logo

Total Probability

https://www.youtube.com/watch?time_continue=11&v=RCEieE2t8U4&feature=emb_logo

- Notebook: 1D Robot World

Probability After Sense

https://www.youtube.com/watch?v=aWQMJQQmNGw&feature=emb_logo

In the above scenario, a robot starts out without any information about where it is in a 1D world and has a uniform distribution of location probabilities: [0.2, 0.2, 0.2, 0.2, 0.2]. You can assume this is the initial distribution for the quiz questions below.

Question 1 of 2

If the robot senses that it is in a red square, and its sensor says that there is a 60% chance that this is indeed a red square, what is the combined probability that a robot is *sensing* red and *in* a red square?

Another way of asking this is: what is the new probability that the robot is in square x2?

- 0.12
-

QUESTION 2 OF 2

If the robot senses that it has reached a red square, but its sensor says that there is still a 20% chance that this is actually a green square (and the sensor reading is incorrect), what is the combined probability that a robot is *sensing* red but *in* a green square?

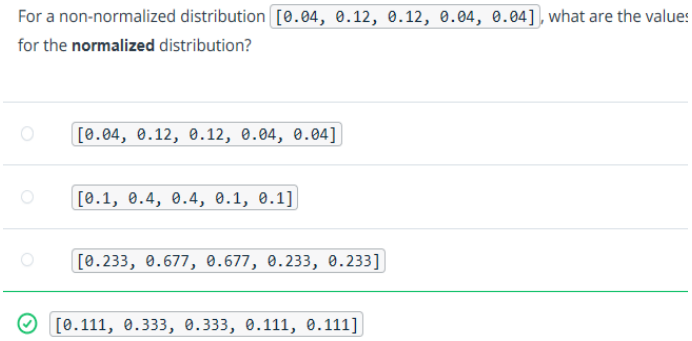
Another way of asking this is: what is the probability that our robot sense red but is actually in square x1?

- 0.04

- Notebook: Probability After Sense

Normalize Distribution

https://www.youtube.com/watch?v=ZGvmFn_u56o&feature=emb_logo



Sense Function

https://www.youtube.com/watch?time_continue=36&v=eIjyrQpDogg&feature=emb_logo

- Notebook: Sense Function

Solution:

https://www.youtube.com/watch?time_continue=7&v=Y5iFxWRTw1c&feature=emb_logo

Normalized Sense Function

https://www.youtube.com/watch?time_continue=4&v=GqWszyHTYas&feature=emb_logo

- Notebook: Normalized Sense Function

Solution:

https://www.youtube.com/watch?v=UX3W8TUKbJ0&feature=emb_logo

Test Sense Function

https://www.youtube.com/watch?time_continue=46&v=gytbuOI9-3g&feature=emb_logo

https://www.youtube.com/watch?v=F8AHaaJVmkw&feature=emb_logo

Multiple Measurements

https://www.youtube.com/watch?time_continue=5&v=gDO4sF8gR9k&feature=emb_logo

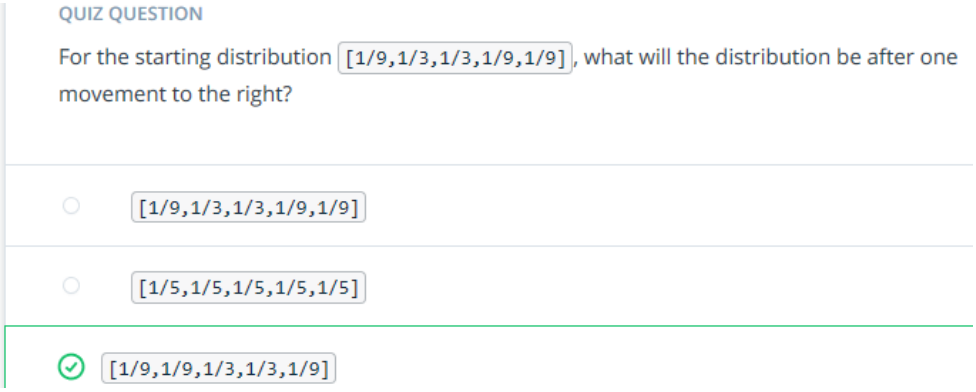
- Notebook: Multiple Measurements

Solution:

https://www.youtube.com/watch?time_continue=3&v=-3qTapGGa-8&feature=emb_logo

Exact Motion

https://www.youtube.com/watch?v=mNXm1wjTumY&feature=emb_logo



Move Function

https://www.youtube.com/watch?v=wfjE0mVADIk&feature=emb_logo

- Notebook move function

Solution:

https://www.youtube.com/watch?v=TnFq6hufsYs&feature=emb_logo

Inexact Motion

https://www.youtube.com/watch?v=hHAwFNsIp1c&feature=emb_logo

For an initial distribution of `[0,1,0,0,0]`, and a motion of **U= 2**, what will the resulting distribution look like when uncertainty in motion is accounted for. You can assume that pUndershoot and pOvershoot are 0.1 and pExact is 0.8.

- `[0, 0, 1, 0, 0]`
- `[0, 0.1, 0.8, 0.1, 0]`
- ✓ `[0, 0, 0.1, 0.8, 0.1]`

Inexact Move Function

https://www.youtube.com/watch?v=68Kao9dkIKA&feature=emb_logo

- Notebook: Inexact Move Function

Sol: https://www.youtube.com/watch?v=QCnPJcNprEU&feature=emb_logo

Limit Distribution

https://www.youtube.com/watch?v=NJvalJwjz18&feature=emb_logo

For a distribution that starts as `[1,0,0,0,0]`, what will this distribution look like after a robot moves infinitely many times (without sensing)? It may help to think about what happens as uncertainty accumulates over 1 motion to the right, then, 2, then 1000, and so on!

If you don't get this answer right away, that's okay, we'll get more into this in the next couple videos!

- `[1,0,0,0,0]`
- `[0,0,0,0,0]`
- ✓ `[0.2,0.2,0.2,0.2,0.2]`

Move Twice

https://www.youtube.com/watch?v=sKiumVTdpgY&feature=emb_logo

https://www.youtube.com/watch?v=oqlgQa1IdcY&feature=emb_logo

Move 1000

https://www.youtube.com/watch?time_continue=3&v=x2o1g3J-1nw&feature=emb_logo

- Notebook: Multiple Moves

Sense and Move

https://www.youtube.com/watch?time_continue=5&v=v2dYzm6-YVs&feature=emb_logo

- Notebook: Sense and Move Cycle

Sol: https://www.youtube.com/watch?v=1s2dRczcu1A&feature=emb_logo

Sense and Move 2

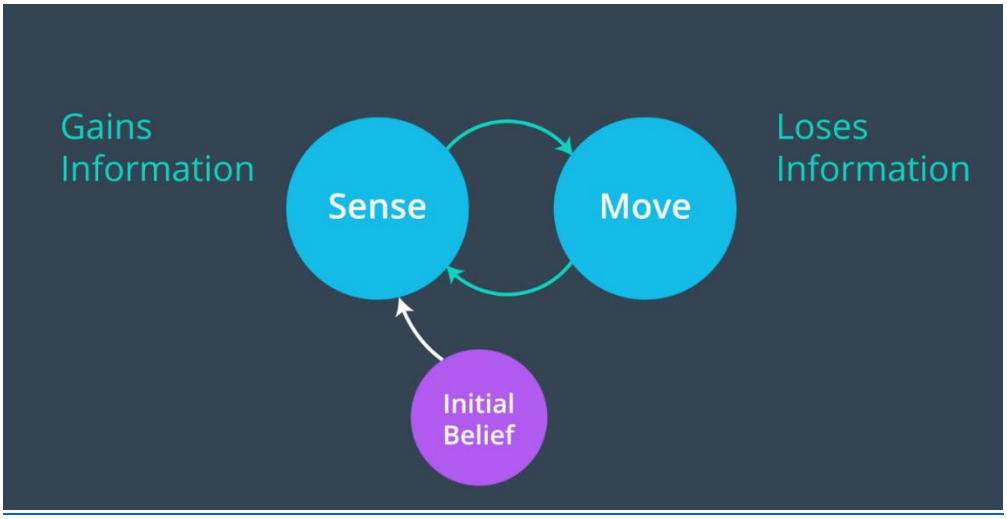
https://www.youtube.com/watch?v=-wT7h9Gdm_8&feature=emb_logo

https://www.youtube.com/watch?v=7gI2GF-laOQ&feature=emb_logo

Localization Summary

Localization

Now, you've learned the foundation of all localization techniques! You know that:, first, a robot starts out with some certainty/uncertainty about its position in a world, which is represented by an initial probability distribution, often called the initial belief or the **prior**. Then it cycles through sensor measurements and movements.

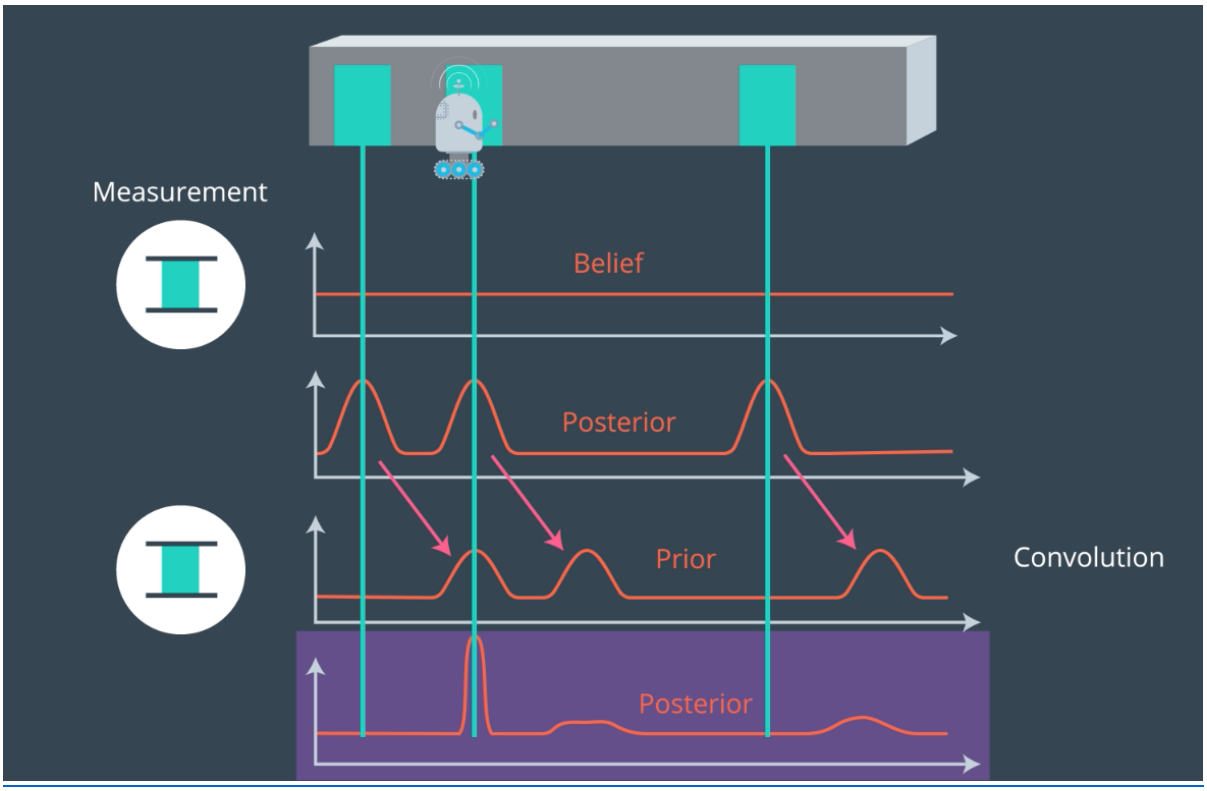


Sense/move cycle.

Sense/Move Cycle

1. When a robot senses, a **measurement update** happens; this is a simple multiplication that is based off of Bayes' rule, which says that we can update our belief based on measurements! This step was also followed by a **normalization** that made sure the resultant distribution was still vald (and added up to 1).
2. When it moves, a **motion update** or prediction step occurs; this step is a convolution that shifts the distribution in the direction of motion.

After this cycle, we are left with an altered **posterior** distribution!



A move sense cycle in action, with an initial belief at the top.

Elective: Learn C++

Now that you've seen how to program a histogram (aka Monte Carlo) localization filter, we have provided an elective section: **C++Programming**, which shows you how to translate this code into efficient C++ code.

C++ logo.

Why Learn C++ ?

If you are looking for a job as a computer vision engineer (rather than just a software developer or researcher), you may find that many job posting require that you know image analysis, deep learning techniques, SLAM, **and** have proficiency in C++. This is because C++ is the language of hardware; most machines like robots, mobile phones, NVIDIA hardware, or autonomous vehicles, run on C++.

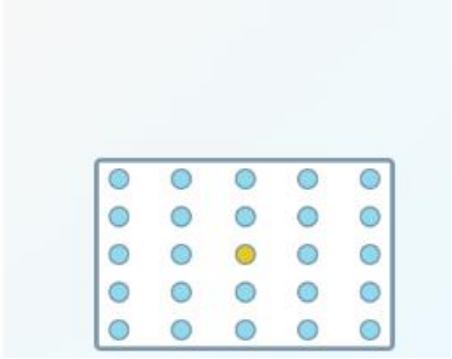
C++ is closer to machine code and can run much faster on hardware than Python can and this is critical for time-sensitive applications. So, should you want to learn more about C++ programming, check out the Elective section to learn the basics of C++ and implement a histogram filter!

This elective course is a very good starting point for learning C++, especially if you are more comfortable with Python and want to expand your knowledge of programming languages. So, if you have any extra time, I highly recommend it!

Below is a video describing what is taught in this elective section!

https://www.youtube.com/watch?v=lR3PH3bL-9U&feature=emb_logo

Lesson 3: 2D Histogram Filer



https://www.youtube.com/watch?v=uaWZNKGTtgM&feature=emb_logo

Project Overview - Two Dimensional Histogram Filter

In this project you will work with and add to an existing code base. You will use a Jupyter Notebook to explore an existing codebase for a 2d Histogram filter.

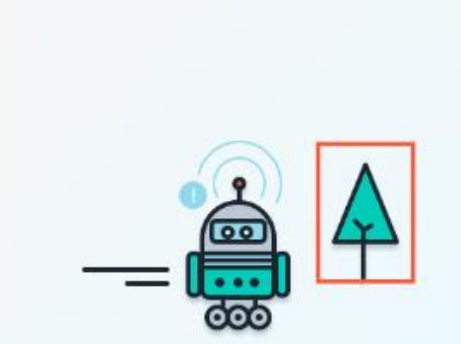
- Notebook - Two Dimensional Histogram Filter

Lesson 4: Intro to Kalman Filters

LESSON 4

Introduction to Kalman Filters

Learn the intuition behind the Kalman Filter, a vehicle tracking algorithm, and implement a one-dimensional tracker of your own.



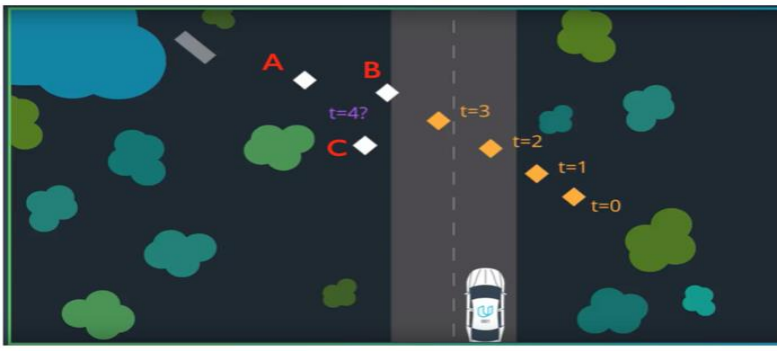
https://www.youtube.com/watch?v=uaWZNKGTtgM&feature=emb_logo

Introduction:

https://www.youtube.com/watch?v=XZL934YQ-FQ&feature=emb_logo

Tracking Intro

https://www.youtube.com/watch?v=C73G7vfVNQc&feature=emb_logo



t = 4 possible locations.

QUIZ QUESTION

Out of the three labelled spots in the image above, which point is the moving bject most likely to be at at time, t = 4?

- A
- B

Inferring Velocity

From the earlier positions, we can infer the velocity (indicated by the pink arrow in the image) of the object; the velocity appears to be u and to the left by a fairly consistent amount between each time step.

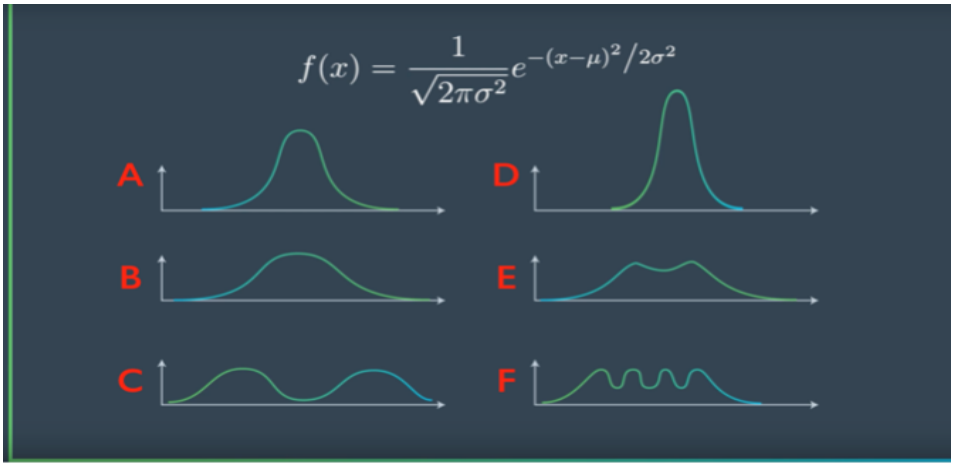
Assuming no drastic change in velocity occurs, we predict that at time t = 4, the object will be on this same trajectory, at point B.

Kalman Filter

A Kalman filter gives us a mathematical way to infer velocity from only a set of measured locations. In this lesson, we'll learn how to create a 1D Kalman filter that takes in positions, like those shown above, takes into account uncertainty, and estimates where future locations might be and the velocity of an object!

Gaussian Intro

https://www.youtube.com/watch?v=S2v1CExswT4&feature=emb_logo



Labelled graphs.

QUIZ QUESTION

Out of all the options in the above image, which of these plots are Gaussians? Select *all* options that apply.

- [x] A
- [x] B
- [ ] C
- [x] D

Gaussian Characteristics

Gaussians are exponential function characterized by a given **mean**, which defines the location of the peak of a Gaussian curve, ad a **variance** which defines the width/spread of the curve. All Gaussian are:

- **symmetrical**
- they have **one peak**, which is also referred to as a "unimodal" distribution, and * they have an exponential drop off on either side of that peak

Options C, E, and F all have multiple peaks and so are *not* Gaussians.

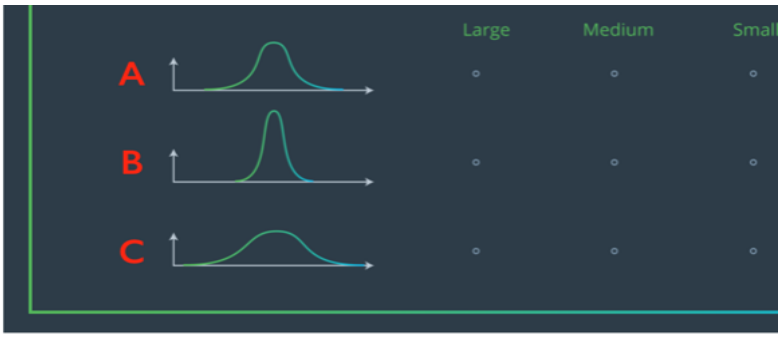


Image of three different Gaussians.

QUESTION 1 OF 2

For the three Gaussians pictured above, which has the largest, smallest, and in-variance? Match their letter label with large/medium/small variance.

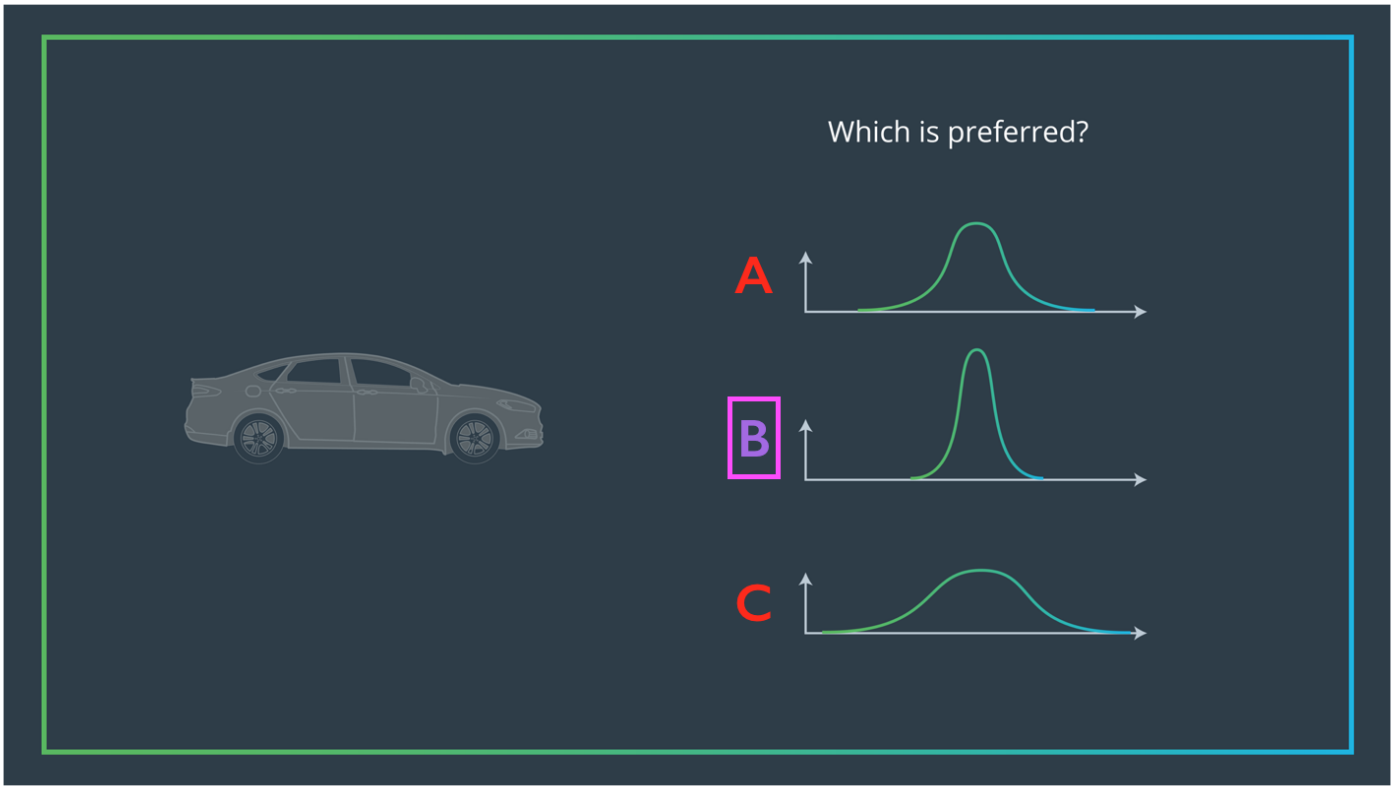
Submit to check your answer choices!

GAUSSIAN LABEL	VARIANCE
A	Medium
B	Small
C	Large

Variance

The variance is a measure of Gaussian spread; the smallest spread is Gaussian B, the largest is Gaussian A. You may also notice that larger variances correspond to shorter Gaussians.

Variance is also a measure of certainty; if you are trying to find something like the location of a car with the *most* certainty, you'll want a Gaussian whose mean is the location of the car and with the smallest uncertainty/spread. As seen in the answer, below.



Smaller variance = more certain measurement.

Maximizing a Gaussian

Below, you'll see a code implementation of a Gaussian function with some specified mean, mu, and variance sigma^2. What value of x would maximize this function?

https://www.youtube.com/watch?v=fRYtUP0P4Lg&feature=emb_logo

See the answer, below!

https://www.youtube.com/watch?v=2cD8T65E-jM&feature=emb_logo

Quiz: Shifting the Mean

https://www.youtube.com/watch?v=gfBdoCFborg&feature=emb_logo

QUIZ QUESTION

After combining the two Gaussians above, where will the new mean be? Choose one of the four options pictured above.

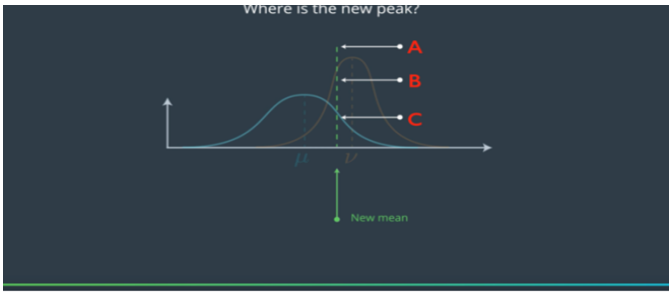
- A
- B
- C

Answer: Shifting the Mean

https://www.youtube.com/watch?time_continue=10&v=L8vNIKvpJ1s&feature=emb_logo

Quiz: Predicting the Peak

https://www.youtube.com/watch?time_continue=19&v=_fGH3xJMxdM&feature=emb_logo



QUIZ QUESTION

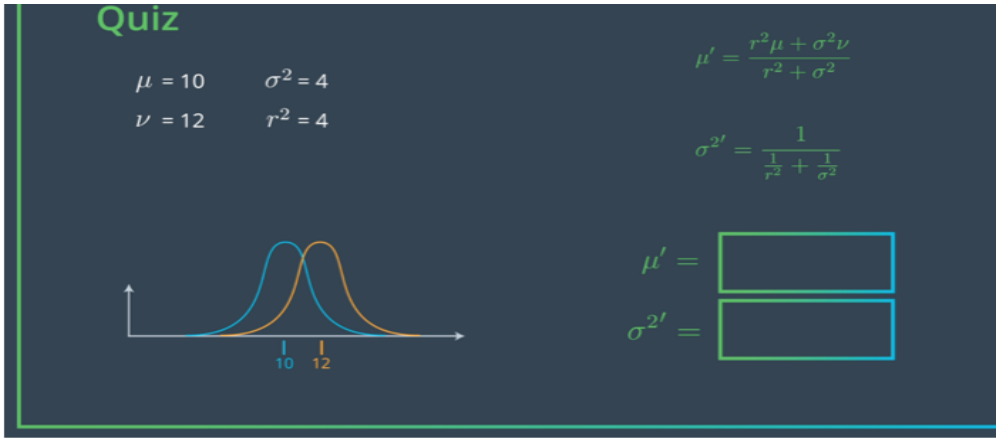
For the newly created Gaussian, given the new mean, how tall do yo think the new peak will be? Recall that the height of a Gaussian is related to its variance term and "certainty." Choose one of the three given options.

- A

https://www.youtube.com/watch?v=mcwr6FcP2Vc&feature=emb_logo

Quiz: Parameter Update

https://www.youtube.com/watch?v=UUXETqShme4&feature=emb_logo



Given the above two Gaussians, with `mu = 10` and `nu = 12` and both squared variances = 4. Answer the following questions about the newly created Gaussian (made from a product of the given two).

What is the new value of the mean, mu-prime?

11 RESET

What is the new squared variance, sigma^2-prime?

8

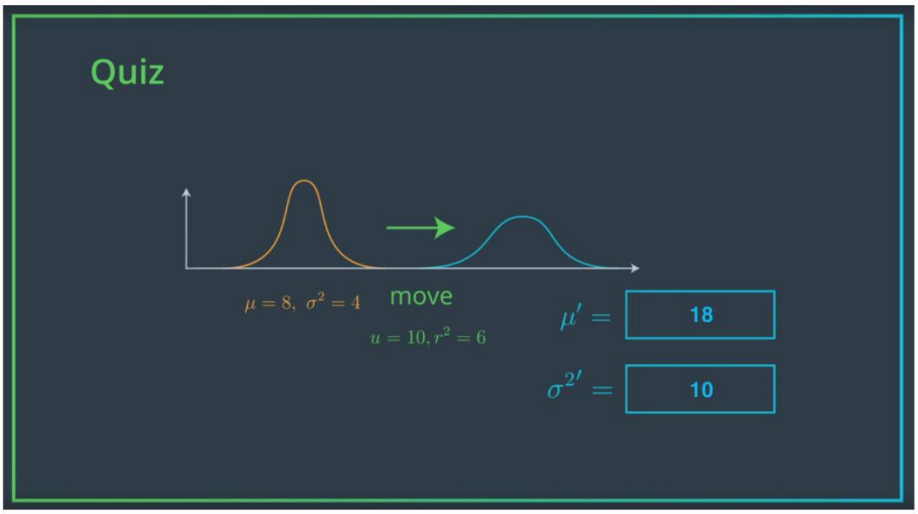
https://www.youtube.com/watch?v=vl6GkkEgY4M&feature=emb_logo

- Notebook: New Mean and Variance

https://www.youtube.com/watch?v=SwxRWZaC1FM&feature=emb_logo

Quiz: Gaussian Motion

https://www.youtube.com/watch?v=LFPT0R3VaPs&feature=emb_logo



Motion Update

A motion update is just an addition between parameters; the new mean will be the old mean + the motion mean; same with the new variance!

Predict Function

https://www.youtube.com/watch?time_continue=17&v=DV2cX9W0tT8&feature=emb_logo

- Notebook: Predict Function

https://www.youtube.com/watch?v=AMFig-sYGfM&feature=emb_logo

Kalman Filter Code

https://www.youtube.com/watch?time_continue=1&v=3xBycKfnCOQ&feature=emb_logo

- Notebook: 1D Kalman Filter

https://www.youtube.com/watch?v=X7cixvcogl8&feature=emb_logo

A Break from Kalman Filters

At this point, you've implemented a one dimensional Kalman Filter and you probably have some idea of how a multi-dimensional Kalman Filter might work.

But in order to actually make (or even use) a Kalman Filter in a 2D or 3D world (or "state space" in the language of robotics) we will first need to learn more about how we can keep track of robot motion and what exactly we mean when we use this word "state".

Motion Models and State

The next lesson will be a concise introduction to representing the position and motion of an object (an object's **state**) using a mathematical representation of motion: **a motion model**.

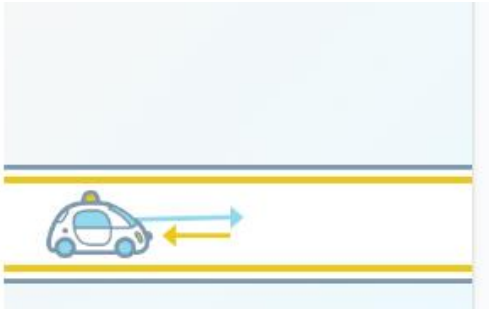
After we go over motion for a self-driving car, we will come back to the *multidimensional* Kalman Filter!

Lesson 5: Representing State and Motion

LESSON 5

Representing State and Motion

Learn about representing the state of a car in a vector that can be modified using linear algebra.



https://www.youtube.com/watch?v=4SiMoSTf4rQ&feature=emb_logo

Localization

All self-driving cars go through the same series of steps to safely navigate through the world.
You've been working on the first step: **localization**. Before cars can safely navigate, they first use sensors and other collected data to best estimate where they are in the world.

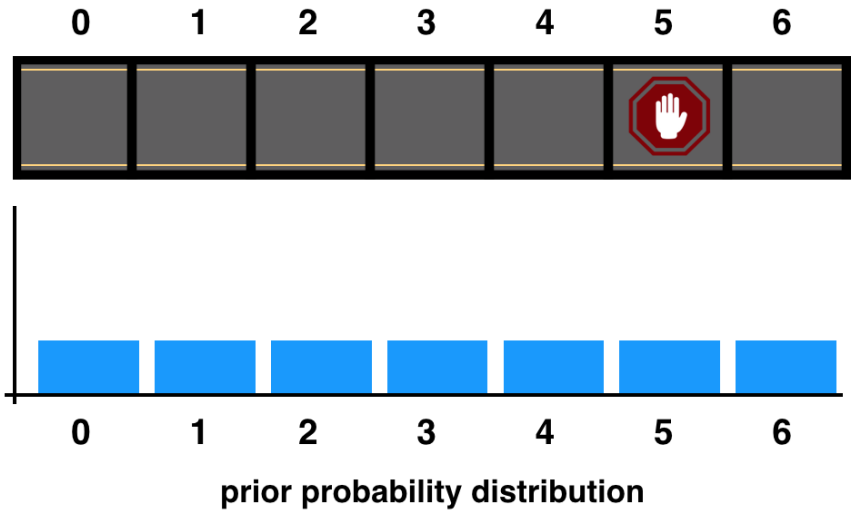
Kalman Filter

Let's review the steps that a Kalman filter takes to localize a car.

1. Initial Prediction

First, we start with an initial prediction of our car's location and a probability distribution that describes our uncertainty about that prediction.

Below is a 1D example, we know that our car is on this one lane road, but we don't know its exact location.

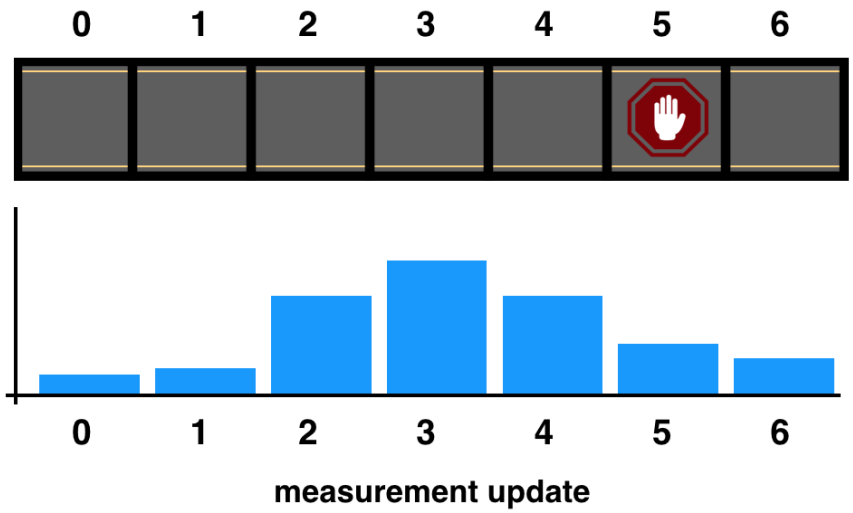


A one lane one and an initial, uniform probability distribution.

2. Measurement Update

We then sense the world around the car. This is called the measurement update step, in which we gather more information about the car's surroundings and refine our location prediction.

Say, we measure that we are about two grid cells in front of the stop sign; our measurement isn't perfect, but we have a much better idea of our car's location.

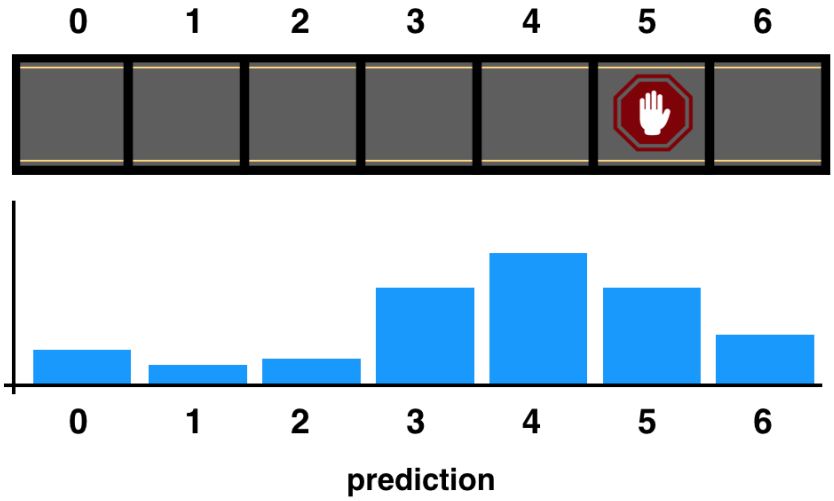


[Measurement update step.]

3. Prediction (or Time Update)

The next step is moving. Also called the time update or prediction step; we predict where the car will move, based on the knowledge we have about its velocity and current position. And we shift our probability distribution to reflect this movement.

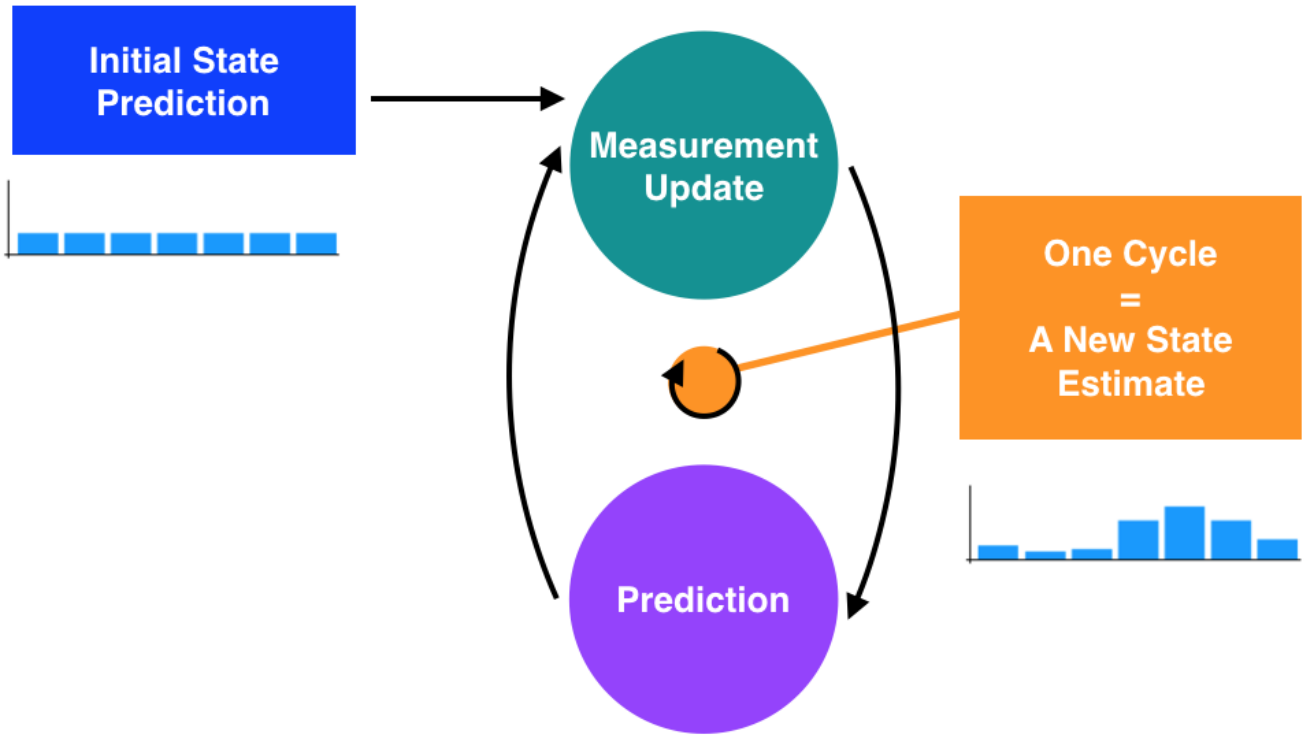
In the next example, we shift our probability distribution to reflect a one cell movement to the right.



[Prediction step.]

4. Repeat

Then, finally, we've formed a new estimate for the position of the car! The Kalman Filter simply repeats the sense and move (measurement and prediction) steps to localize the car as it's moving!



[Kalman Filter steps.]

The Takeaway

The beauty of Kalman filters is that they combine somewhat inaccurate sensor measurements with somewhat inaccurate predictions of motion to get a filtered location estimate **that is better than any estimates that come from *only* sensor readings or *only* knowledge about movement.**

What is State?

When you localize a car, you're interested in only the car's position and it's movement.

This is often called the **state** of the car.

- The state of any system is a set of values that we care about.

In the cases we've been working with, the state of the car includes the car's current **position, x,** and its **velocity, v**.

In code this looks something like this:

x = 4

vel = 1

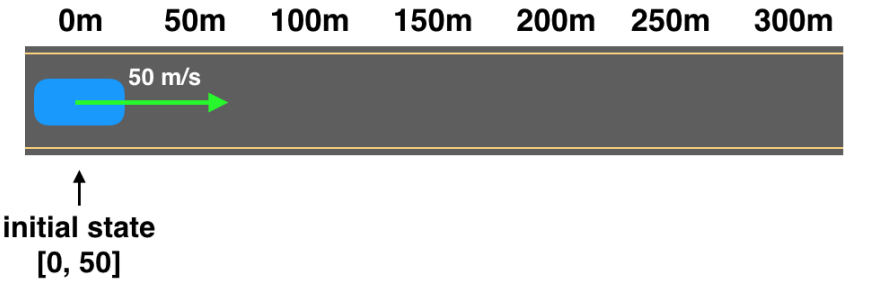
state = [x, vel]

Predicting Future States

The state gives us most of the information we need to form predictions about a car's future location. And in this lesson, we'll see how to represent state and how it changes over time.

For example, say that our world is a one-lane road, and we know that the current position of our car is at the start of this road, at the 0m mark. We also know the car's velocity: it's moving forward at a rate of 50m/s. These values are it's *initial state*.

state = [0, 50]



Using our notation for state, what do you think the new state of the car will be after 3 seconds pass?

- [0, 50]
- [50, 50]
- ✓ [150, 50]

Motion Models

https://www.youtube.com/watch?v=qSdbn_PVQnk&feature=emb_logo

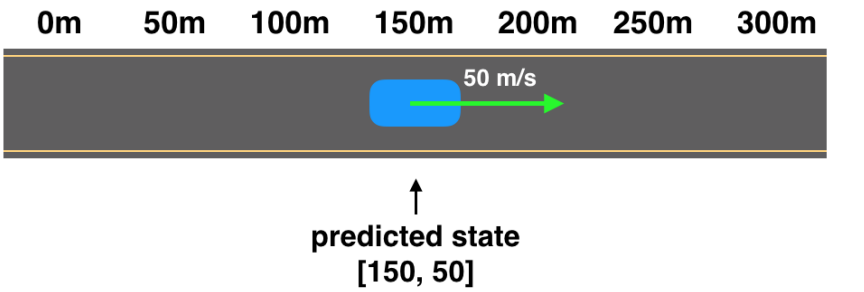
Predicting State

Let's look at the last example.

The initial state of the car is at the 0m position,and it's moving forward at a velocity of 50 m/s. Let's assume that our car keeps moving forward at a constant rate.

Every second it moves 50m.

So, after three seconds, it will have reached the **150m mark** and its velocity will not have changed (that's what a constant velocity means)!



Predicted state after 3 seconds have elapsed.

Its new, predicted state will be at the position 150m, and with the velocity still equal to 50m/s.

predicted_state = [150, 50]

Motion Model

This is a reasonable prediction, and we made it using:

1. The initial state of the car and
2. An assumption that the car is moving at a constant velocity.

This assumption is based on the physics equation:

distance_traveled = velocity * time

This equation is also referred to as a **motion model**. And there are many ways to model motion!

This motion model assumes *constant* velocity.

In our example, we were moving at a constant velocity of 50m/s for three seconds.

And we formed our new position estimate with the distance equation: 150m = 50m/sec*3sec.

The Takeaway

In order to predict where a car will be at a future point in time, you rely on a motion model.

Uncertainty

It's important to note, that no motion model is perfect; it's a challenge to account for outside factors like wind or elevation, or even things like tire slippage, and so on.

But these models are still very important for localization.

Next, you'll be asked to write a function that uses a motion model to predict a new state!

In this quiz, you're asked to write a function that uses a motion model to predict a new state, given some initial parameters!

predict.py solution.py

```
# The predict_state function should take in a state
# and a change in time, dt (ex. 3 for 3 seconds)
# and it should output a new, predicted state
# based on a constant motion model
# This function also assumes that all units are in m, m/s, s, etc.

def predict_state(state, dt):
    # Assume that state takes the form [x, vel] i.e. [0, 50]

    ## Calculate the new position, predicted_x
    ## Calculate the new velocity, predicted_vel
    ## These should be calculated based on the contant motion model:
    ## distance = x + velocity*time

    predicted_x = state[0] + state[1] * dt
    predicted_vel = state[1]

    # Constructs the predicted state and returns it
    predicted_state = [predicted_x, predicted_vel]
    return predicted_state

# A state and function call for testing purposes - do not delete
# but feel free to change the values for the test variables
test_state = [10, 3]
test_dt = 5

test_output = predict_state(test_state, test_dt)
```

A Different Model

https://www.youtube.com/watch?time_continue=2&v=Mh0g-SMpMI4&feature=emb_logo

More Complex Motion

Now, what if I gave you a more complex motion example?

And I told you that our car starts at the same point, at the 0m mark, and it's moving 50m/s forward, but it's *also* slowing down at a rate of 20m/s^2. This means it's acceleration = -20m/s^2.

Car moving at 50m/s and slowing down over time.

Acceleration

So, if the car has a -20 m/s^2 acceleration, this means that:

- If the car starts at a speed of 50m/s
- At the next second, it will be going 50-20 or 30m/s and,
- At the *next* second it will be going 30-20 or 10m/s.

This slowing down is also *continuous*, which means it happens gradually over time.

New Model, New State

For the next two quizzes, I want you to keep in mind this question: **Where will the car be after 3 seconds?**

I also want to ask you:

- What variables do you need to solve this problem? In other words, what values should be included in the state? And...
- What motion model should we use to solve this problem

Kinematics

Kinematics is the study of the motion of objects. Motion models are also referred to as kinematic equations, and these equations give you all the information you need to be able to predict the motion of a car.

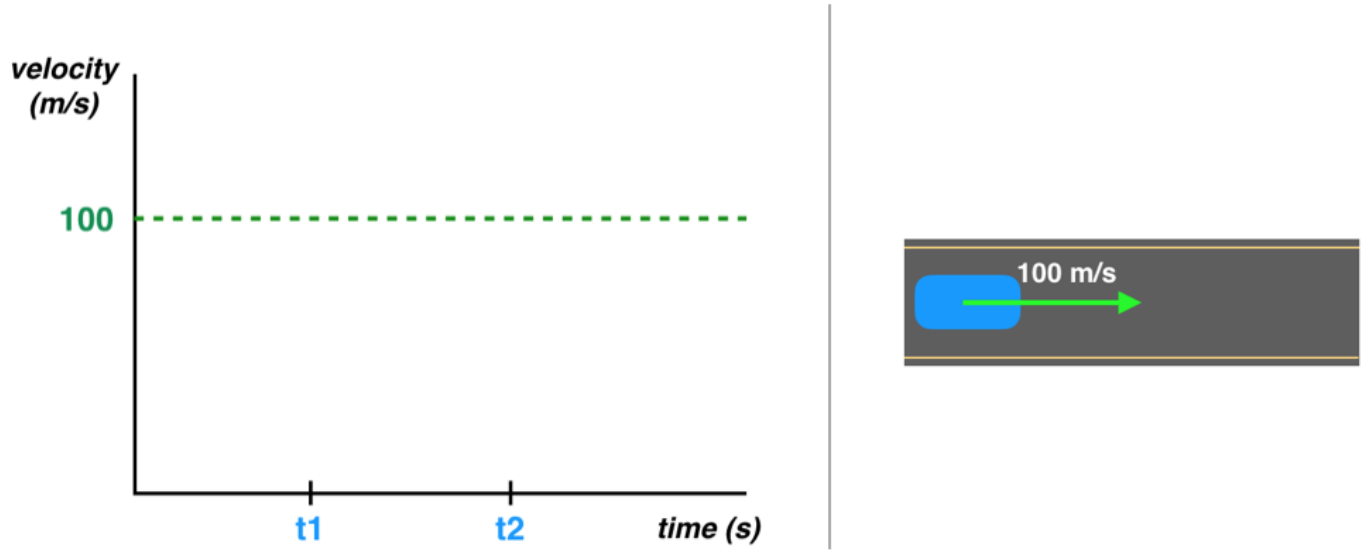
Let's derive some of the most common motion models!

Constant Velocity

The constant velocity model assumes that a car moves at a constant speed. This is the simplest model for car movement.

Example

Say that our car is moving 100m/s, and we want to figure out how much it has moved from one point in time, t1, to another, t2. This is represented by the graph below.



(Left) Graph of car velocity, (Right) a car going 100m/s on a road

Displacement

How much the car has moved is called the **displacement** and we already know how to calculate this!

We know, for example, that if the difference between t2 and t1 is one second, then we'll have moved `100m/sec*1sec = 100m`. If the difference between t2 and t1 is two seconds, then we'll have moved `100m/sec*2sec = 200m`.

The displacement is always = `100m/sec*(t2-t1)`.

Motion Model

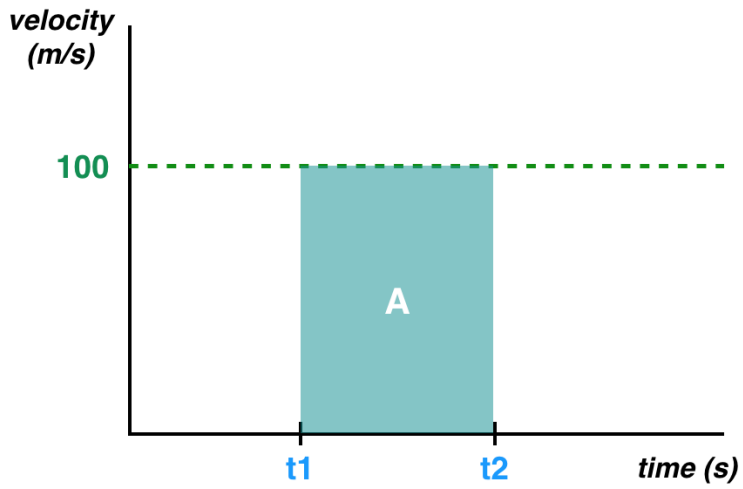
Generally, for constant velocity, the motion model for displacement is:

```
displacement = velocity*dt
```

Where `dt` is calculus notation for "*difference in time.*"

Area Under the Line

Going back to our graph, displacement can also be thought of as the area under the line and within the given time interval.



The area under the line, A, is equal to the displacement!

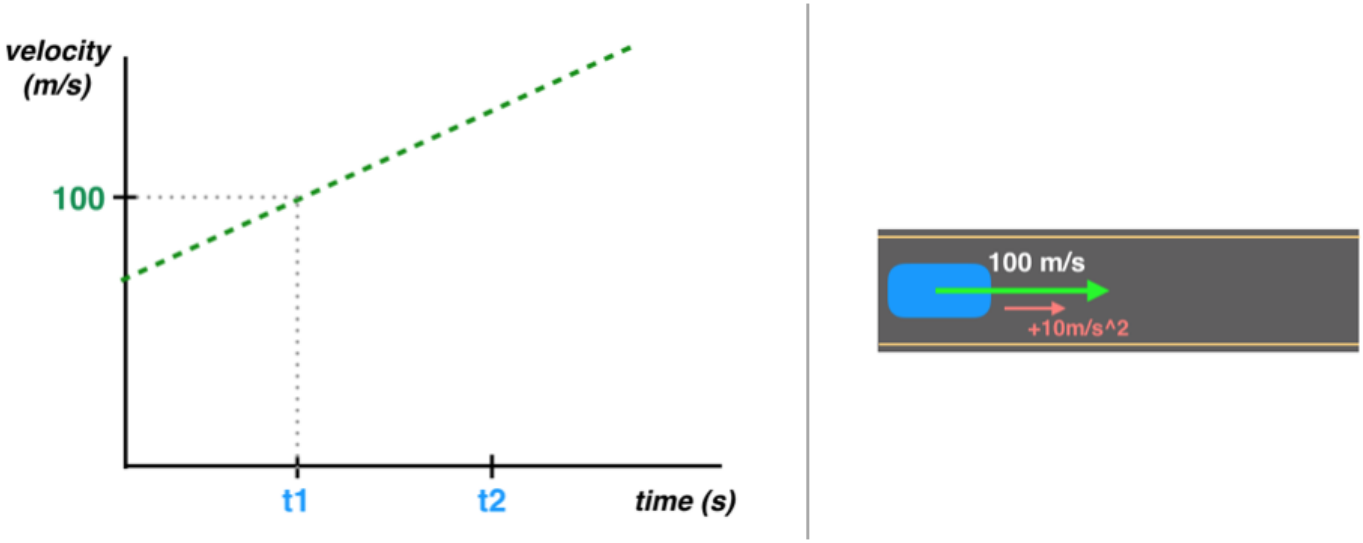
So, in addition to our motion model, we can also say that the displacement is equal to the area under the line!

```
displacement = A
```

Constant Acceleration

The constant acceleration model is a little different; it assumes that our car is constantly accelerating; its velocity is changing at a constant rate.

Let's say our car has a velocity of 100m/s at time t1 and is accelerating at a rate of 10m/s210m/s^210m/s^2.



Changing Velocity

For this motion model, we know that the velocity is constantly changing, and increasing +10m/s each second. This can be represented by this kinematic equation (where dv is the change in velocity):

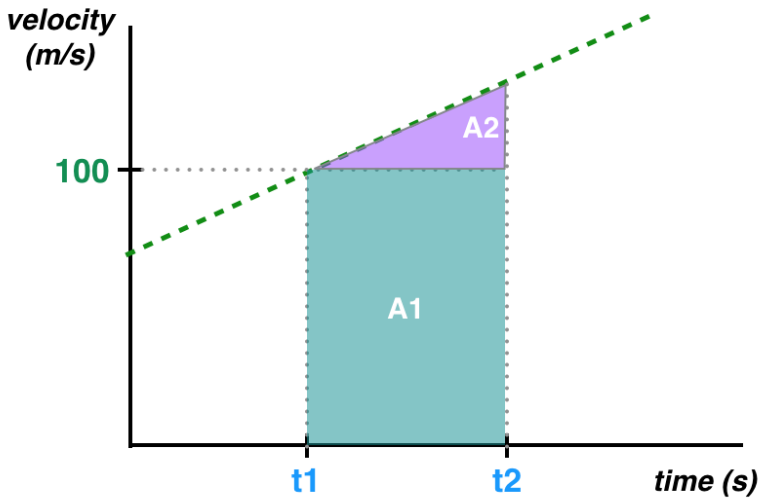
```
dv = acceleration*dt
```

At any given time, this can also be written as the current velocity is the initial velocity + the change in velocity over some time (dv):

```
v = initial_velocity + acceleration*dt
```

Displacement

Displacement can be calculated by finding the area under the line in between t1 and t2, similar to our constant velocity equation but a slightly different shape.



Area under the line, A1 and A2

This area can be calculated by breaking this area into two distinct shapes; a simple rectangle, A1, and a triangle, A2.

A1 is the same area as in the constant velocity model.

```
A1 = initial_velocity*dt
```

In other words, `A1 = 100m/s*(t2-t1)`.
A2 is a little trickier to calculate, but remember that the area of a triangle is `0.5*width*height`.
The width, we know, is our change in time (t2-t1) or `dt`.

And the height is the change in velocity over that time! From our earlier equation for velocity, we know that this value, dv, is equal to: `acceleration*(t2-t1)` or `acceleration*dt`
Now that we have the width and height of the triangle, we can calculate A2. Note that `**` is a Python operator for an exponent, so `**2` is equivalent to `^2` in mathematics or squaring a value.

```
A2 = 0.5*acceleration*dt**2
```

Motion Model

This means that our total displacement, A1+A2 ,can be represented by the equation:

```
displacement = initial_velocity*dt + 0.5*acceleration*dt**2
```

We also know that our velocity over time changes according to the equation:

```
dv = acceleration*dt
```

And these two equations, together, make up our motion model for constant acceleration.

Quantifying State

Different Motion Models

Constant Velocity

In the first movement example, you saw that if we assumed our car was moving at a constant speed, 50 m/s, we came up with one prediction for it's new state: at the 150 m mark, with no change in velocity.

 # Constant velocity case

 # initial variables

x = 0

velocity = 50

initial_state = [x, velocity]

 # predicted state (after three seconds)

 # this state has a new value for x, but the same velocity as in the initial state

dt = 3

new_x = x + velocity*dt

predicted_state = [new_x, velocity] *# predicted_state = [150, 50]*

For this constant velocity model, we had:

- initial state = [0, 50]
- predicted state (after 3 seconds) = [150, 50]

Constant Acceleration

But in the second case, we said that the car was slowing down at a rate of 20 m/s^2 and, after 3 seconds had elapsed, we ended up with a different estimate for its state.

To solve this localization problem, we had to use a different motion model and **we had to include a new value in our state: the acceleration of the car**.

The motion model was for constant acceleration:

- distance = velocity*dt + 0.5*acceleration*dt^2 and
- velocity = acceleration*dt

The state includes acceleration in this model and looks like this: [x, velocity, acc].

 # Constant acceleration, changing velocity

 # initial variables

x = 0

velocity = 50

acc = -20

initial_state = [x, velocity, acc]

 # predicted state after three seconds have elapsed

 # this state has a new value for x, and a new value for velocity (but the acceleration stays the same)

dt = 3

new_x = x + velocity*dt + 0.5*acc*dt**2

new_vel = velocity + acc*dt

predicted_state = [new_x, new_vel, acc] *# predicted_state = [60, -10, -20]*

For this constant *acceleration* model, we had:

- initial state = [0, 50, -20]
- predicted state (after 3 seconds) = [60, -10, -20]

As you can see, our state and our state estimates vary based on the motion model we used and how we assumed the car was moving!

How Many State Variables?

In fact, how many variables our state requires, depends on what motion model we are using.

For a constant velocity model, x and velocity will suffice.

But for a constant acceleration model, you'll also need our acceleration:acc.

But these are all just models.

The Takeaway

For our state, we always choose **the smallest representation** (the smallest number of variables) that will work for our model.

Reflect

What are some applications or challenges you can think of, in which the *color* of a car would be an important state variable?

Things to think about

You can imagine, if you are a quality assurance engineer, you have to make sure the color of a car is correct before shipping. Another example would be for marketing; you have to make sure that the color of the car matches the logo of the brand or sports team you are trying to match. In these cases, color is very important and would be a valuable state variable!

Lesson Outline

The one unifying theme in this lesson is representing and predicting state, but there are two threads that we'll use to explore this idea.

1. Object-Oriented Programming

On the programming side, we'll use something called Object-Oriented Programming as a way to represent state in code. We'll use variables to represent state values and we'll create functions to change those values.

2. Linear Algebra

On the mathematical side we'll use vectors and matrices to keep track of state and change it.

We'll learn all the necessary math notation and code, as we learn more about predicting the state of a car!

https://www.youtube.com/watch?v=jh7wLGXrm3E&feature=emb_logo

Always Moving

https://www.youtube.com/watch?v=EQBQlHvxAQA&feature=emb_logo

Self-driving cars constantly monitor their state. So, movement and localization have to occur in parallel.

If we use a Kalman filter for localization, this means that as a car moves, the Kalman filter has to keep coming up with new state estimates. This way, the car *always* has an idea of where it is.

Always Predicting State

In the code below, you are given a predict_state function that takes in a current state and a change in time, dt, and returns the new state estimate (based on a constant velocity model).

It will be up to you to use this function repeatedly to find the predicted_state at 5 different points in time:

- the initial state
- the predicted state after 2 seconds have elapsed
- the predicted state after 3 *more* seconds have elapsed
- the predicted state after 1 *more* second has elapsed
- the predicted state after 4 *more* seconds have elapsed

To first three states have been given to you in code.

multiple_predictions.py | functions.py | solution.py

```python
from functions import predict_state

# predict_state takes in a state and a change in time, dt
# So, a call might look like: new_state = predict_state(old_state, 2)

# The car starts at position = 0, going 60 m/s
# The initial state:
initial_state = [10, 60]

# After 2 seconds:
state_est1 = predict_state(initial_state, 2)

# 3 more seconds after the first estimated state
state_est2 = predict_state(state_est1, 3)

## Use the predict_state function
## and the above variables to calculate the following states

## 1 more second after the second state estimate
state_est3 = predict_state(state_est2, 1)

## 4 more seconds after the third estimated state
state_est4 = predict_state(state_est3, 4)
```

multiple_predictions.py | functions.py | solution.py

```python
#---- predict state function --#
def predict_state(state, dt):
    # Assumes a valid state had been passed in
    # Assumes a constant velocity model
    x = state[0]
    new_x = x+state[1]*dt

    # Create and return the new, predicted state
    predicted_state = [new_x, state[1]]
    return predicted_state
```

Creating a Car Object

To create a Car, which was named carla in the example. I have to:

1. Import our car file and define a car's initial state variable, and
2. Call car.Car(); a special function that initializes a Car object, and pass in the initial state variables.

The state is defined by a position: [y, x] and a velocity, which has vertical and horizontal components: [vy, vx]. And lastly, we had to pass in a world, which is just a 2D array.

Imports and Defining initial variables

 # Import statements

import numpy

import car

 # Declare initial variables

 # Create a 2D world of 0's

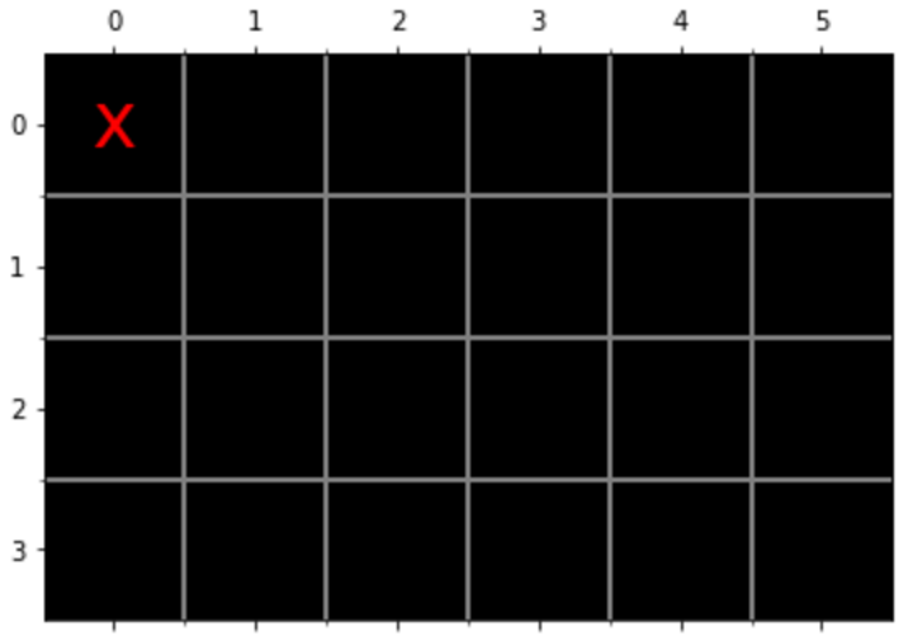
height = 4

width = 6

```
world = np.zeros((height, width))


# Define the initial car state
initial_position = [0, 0] # [y, x] (top-left corner)
velocity = [0, 1] # [vy, vx] (moving to the right)
```

Creating and Visualizing a Car!

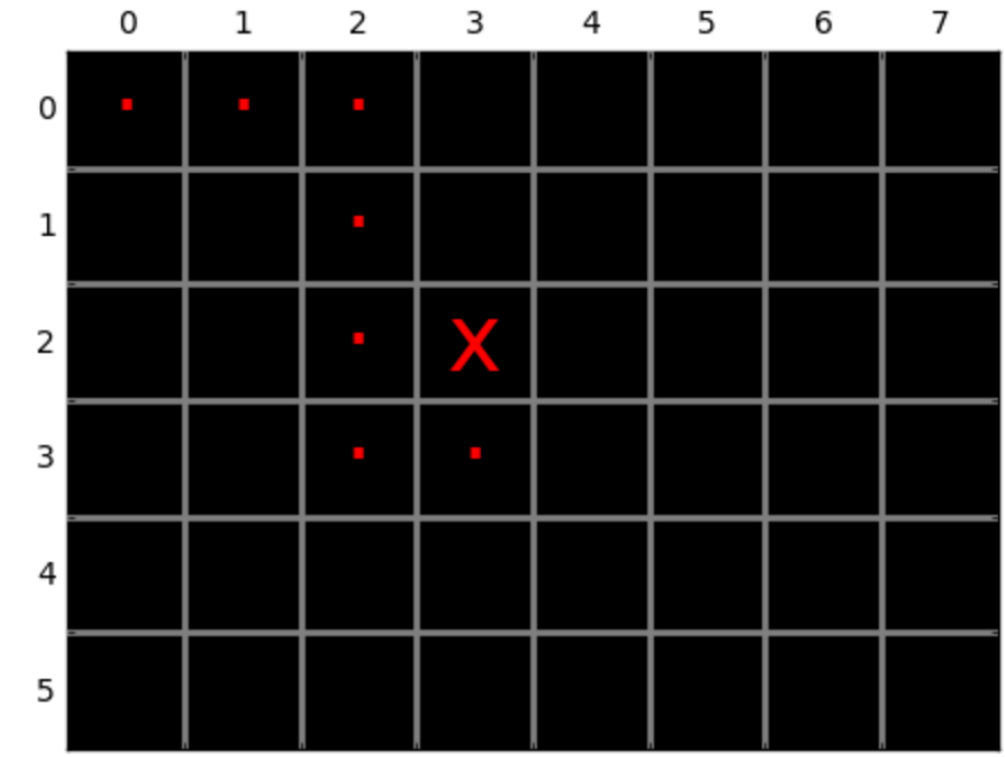
```
# Create a car object with these initial params
carla = car.Car(initial_position, velocity, world)


# Display the world
carla.display_world()
```



Carla's initial state at position [0,0]

Car movement

Carla can also move in the direction of the velocity and turnleft with the functions: move() and turn_left().



https://www.youtube.com/watch?v=SnfhGZ76h7Y&feature=emb_logo

- Notebook Interacting with a Car Object

Adding a turn_right() function to car.py

Your tasks for this car.py file and notebook are:

- Add a turn_right() function to car.py
    - There are a few ways to do this. I'd suggest looking at the code in turn_left() or even using this function.
- Don't forget to update the state as necessary, after a turn!
- Test out your turn_right() function in the notebook (.ipynb) by visualizing the car as it moves, and printing out the state of the car to see if it matches what you expect!
- If you get stuck, you can check out car_solution.py for some tips.
- Run the final cell of the notebook to check whether your solution matches ours as well!

https://www.youtube.com/watch?v=iltQBIpbCSw&feature=emb_logo

Overloading

Now that we've seen how to create a class and talked about some of the fundamental functions and variables that classes contain, let's look at something new!

The double underscore __X__

You've seen a couple of examples of functions that have a double underscore, like:

__init__

__repr__

These are **special functions** that are used by Python in a specific way.

We typically don't call these functions directly, as we do with ones like move() and `turn_left().

Instead, **Python calls them *automatically*** based on our use of keywords and operators.

For example, __init__ is called when we create a new object and __repr__ is called when we tell Python to print the string representation of a specific object!

Another example: __add__

All of these special functions have their names written between double underscores __, and there are many of these types of functions! To see the full list of these functions, check out **the Python documentation**.

For example, we can define what happens when we add two car objects together using a + symbol by defining the __add__ function.

```
def __add__(self, other):
    # Create an empty list
    added_state = []

    # Add the states together, element-wise
    for i in range(self.state):
        added_value = self.state[i] + other.state[i]
        added_state.append(added_value)
```

```
    return added_state
```

The above version, adds together the state variables! Or.. you may choose to just print out that adding cars is an invalid operation, as below.

```
def __add__(self, other):
    # Print an error message and return the unchanged, first state
    print('Adding two cars is an invalid operation!')
    return self.state
```

Operator Overloading

When we define these functions in our class, this is called **operator overloading**.

And, in this case, overloading just means: *giving more than one meaning to* a standard operator like addition.

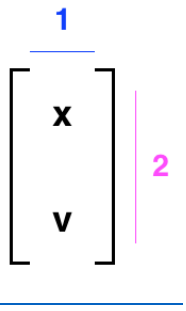
Operator overloading can be a powerful tool, and you'll not only see it pop up again and again in classes, but it is useful for writing classes that are intuitive and simple to use. So, keep this in mind as you continue learning, and **let's get some practice with overloading operators!**

https://www.youtube.com/watch?v=st26ov_TVwM&feature=emb_logo

State Vector

https://www.youtube.com/watch?v=DRRuQMYo800&feature=emb_logo

A state vector is a column of values whose dimensions are 1 in width and M in height. This vector should contain all the values that we are interested in, and for predicting 1D movement, we are interested in the position, x, and the velocity, v.



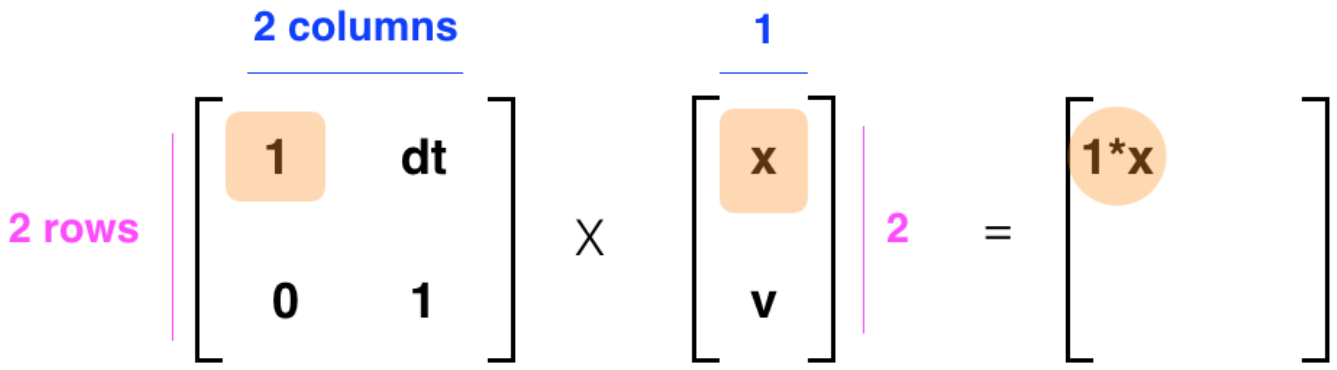
An example of a 2x1 state vector that contains variables: x and v.

Efficiently predicting state

With a state vector, we can predict a new state in just one matrix multiplication step.

Matrix multiplication

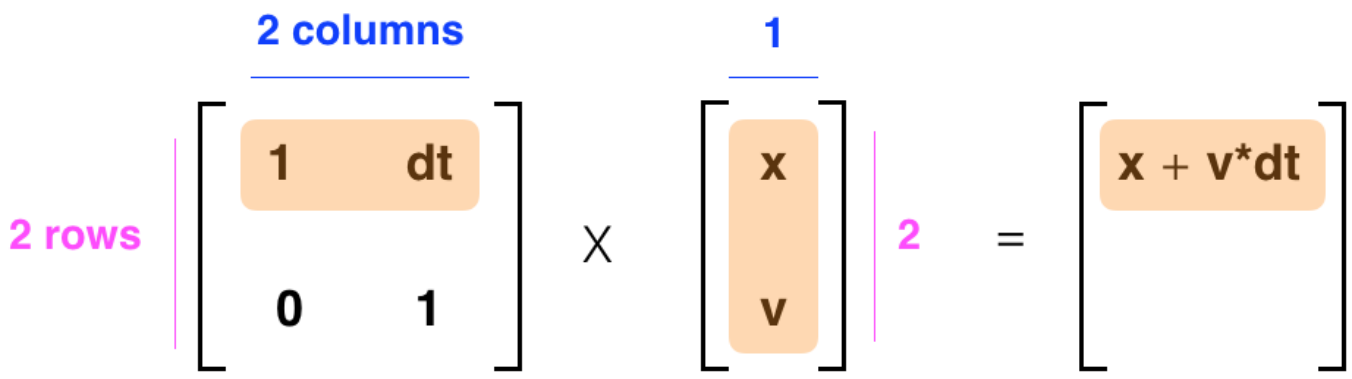
Matrix multiplication multiplies two grids of numbers; multiplying the rows in the first matrix, by the columns in the second. One step in this process is pictured, below.



The start of matrix multiplication for these 2x2 and 2x1 matrices.

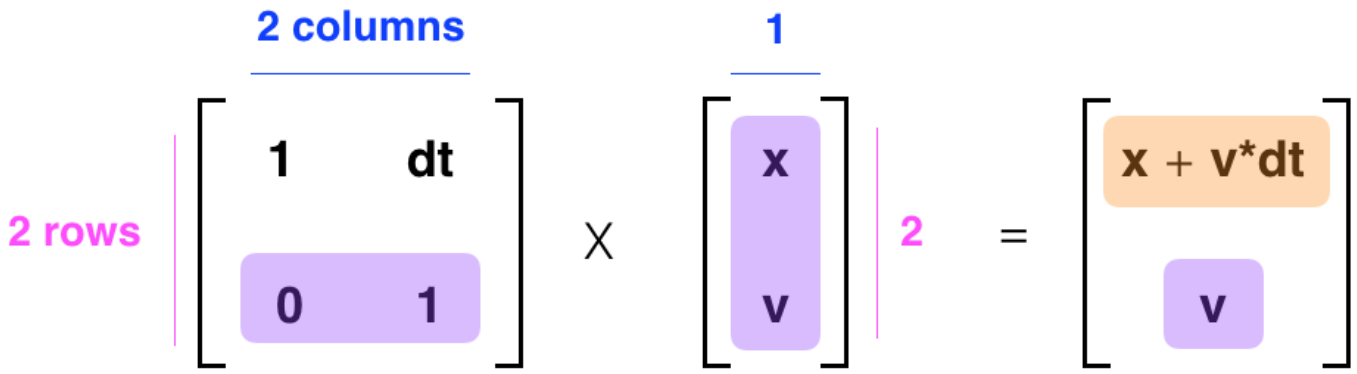
Summing step

Once a whole row and column have been multiplied, matrix multiplication sums those values to form a single, new value in a resulting matrix.



Summation step: x + v*dt

Then it moves on to the next row, and this process repeats.



Completed matrix multiplication!

You can see that this creates a new 2x1 vector, with two values in it that may look familiar! These are just our equations for our constant velocity motion model. So, matrix multiplication let's us create a new, predicted state vector in just one multiplication step!

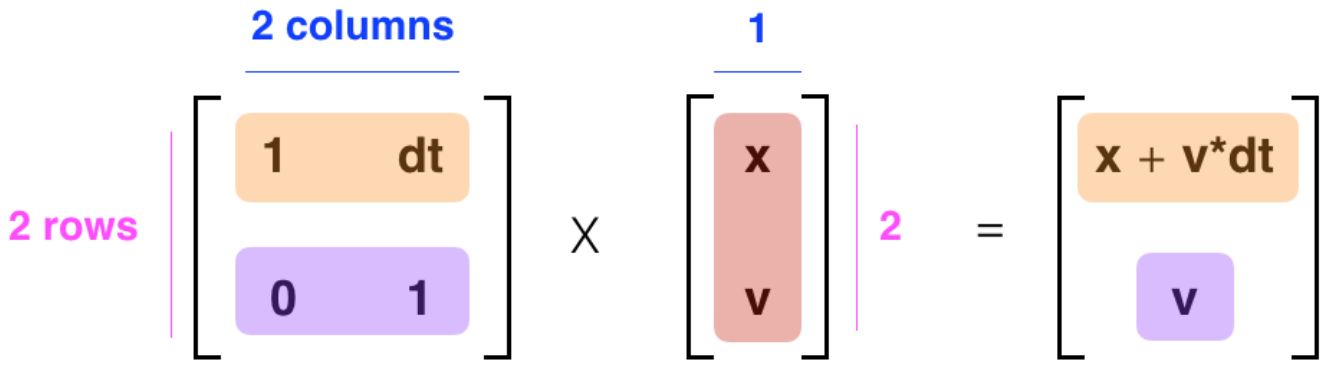
In fact, this is such a common way to predict a new state, that the 2x2 matrix on the left is often called the state transformation matrix.

Matrix Multiplication

Now that you've seen how Matrix multiplication is used in state transformation, let's go through some concrete examples, and test your knowledge!

Remember that multiplying two matrices involves a couple steps:

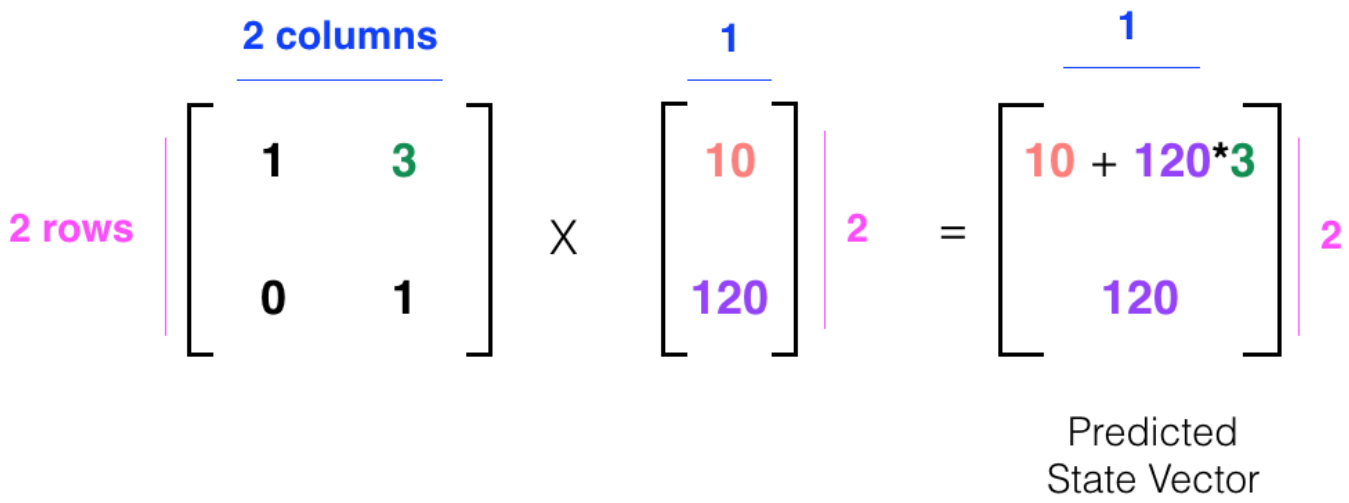
1. It multiplies each row in the first matrix with the columns in the second, element-wise (that means the number of columns in the first matrix and rows in the second **must be equal**).
2. It sums up those multiplied values to form a new value in a new matrix at a specific location.
    - **ex.** If we multiply the *first row* of the first matrix by the *first column* of the second, this new, summed value belongs in the *first row and first column* of the resulting matrix.
    - **ex.** If we multiply the *second row* of the first matrix by the *first column* of the second, this new, summed value belongs in the *second* row and first column of the resulting matrix.
3. Matrix multiplication takes practice, so take a look at [this page] for more examples!



[State transformation by matrix multiplication.]

Let's consider the case where the position of a self-driving car is at: x = 10 , and velocity,v = 120.

Where would you predict that the car with be in 3 seconds, using matrix multiplication? Well, we'd simply plug in the numbers for x, v, and dt in the transformation equation:



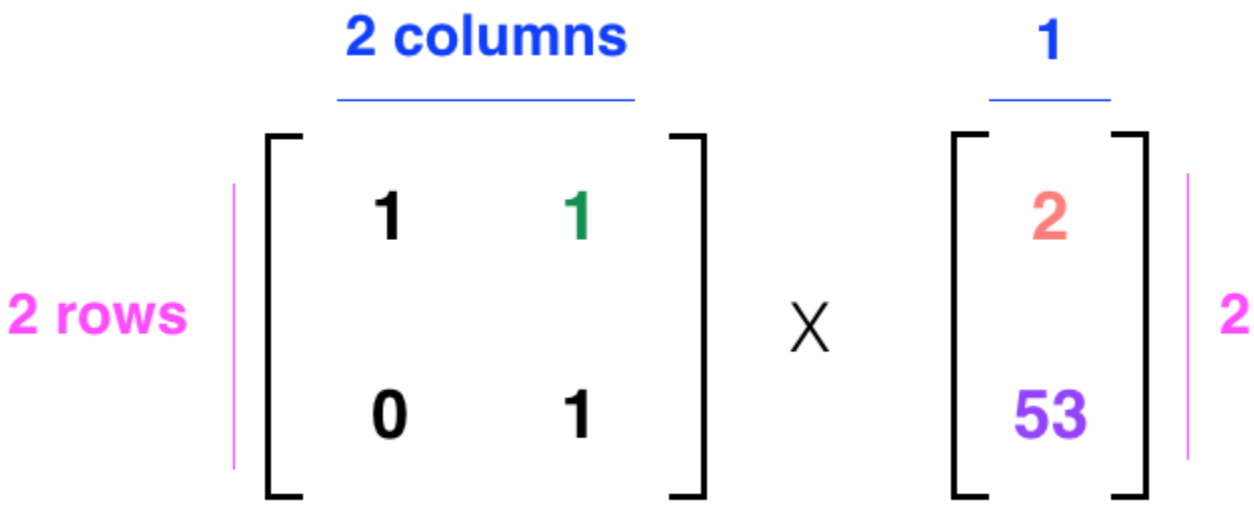
[New, predicted state vector.]

Predicted State Vector

We'd get a new, predicted state vector with x = 10+120*3. and a constant velocity.

x = 370 and v = 120

Let's try a few more examples.



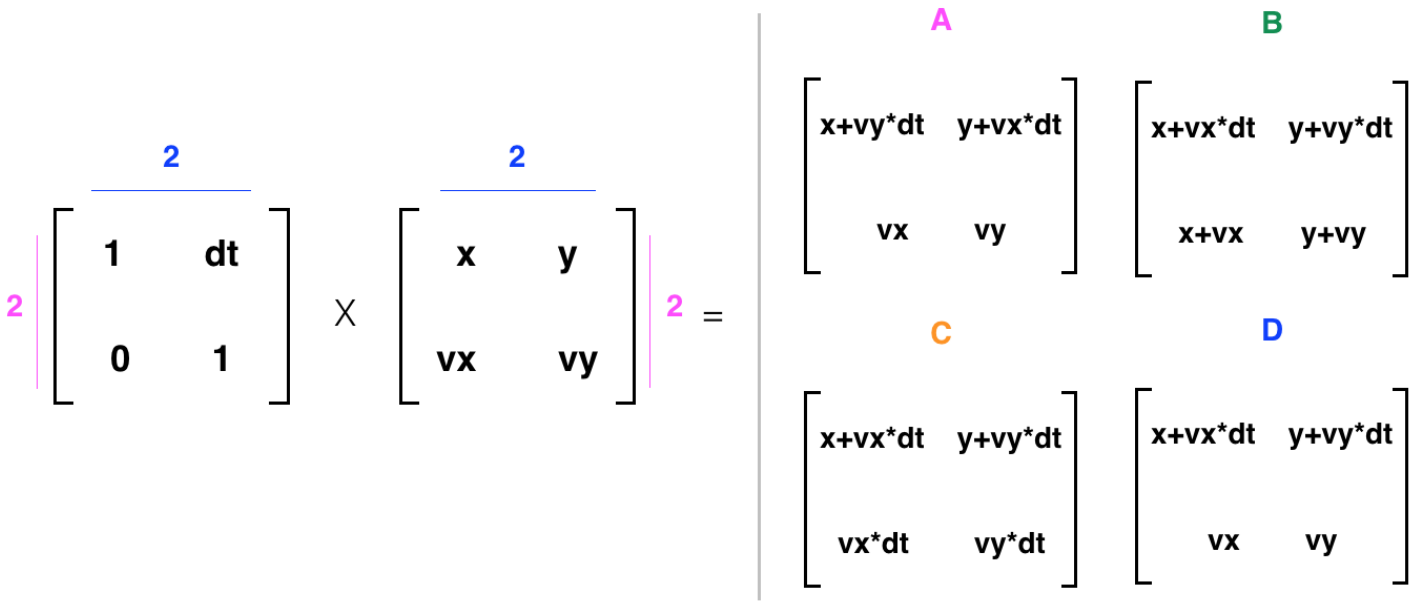
dt = 1, x = 2, and v = 53

Question 1 of 2

What is the resulting predicted state vector for the values: dt = 1, x = 2, and v = 53? Imagine these lists and column vectors with two values: [x, v].

- [55, 53]

What about for a 2x2 matrix that's **not a state vector**? Take a look at the scenario below and the options for the resulting matrix.



Question 2 of 2

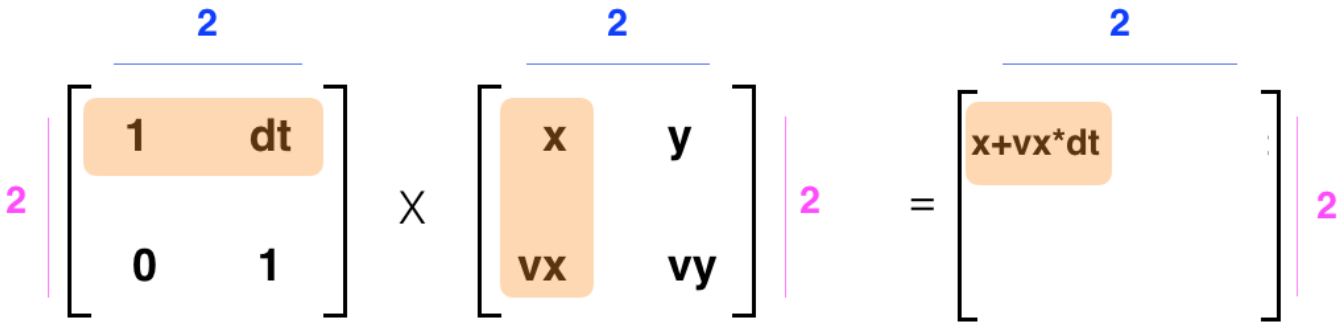
Which of the above 2x2 matrices, A-D, is the correct, resulting matrix?

- D

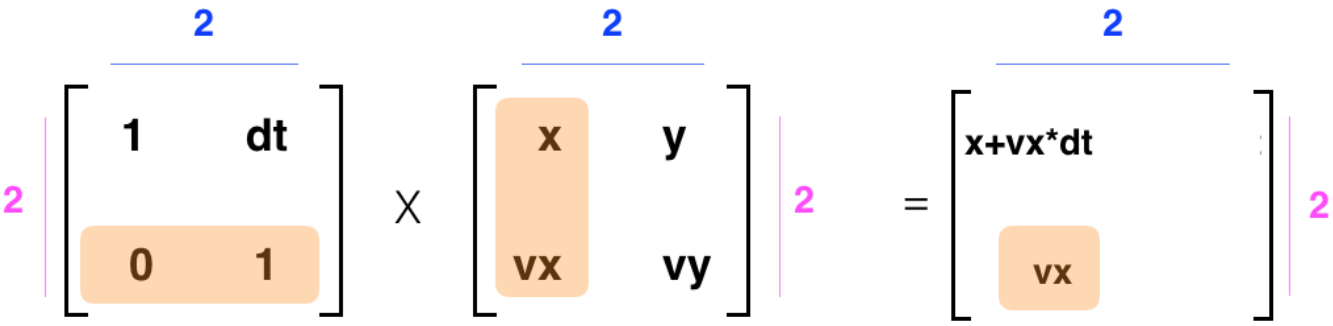
Matrix Multiplication

Let's walk through that last quiz example, step-by-step.

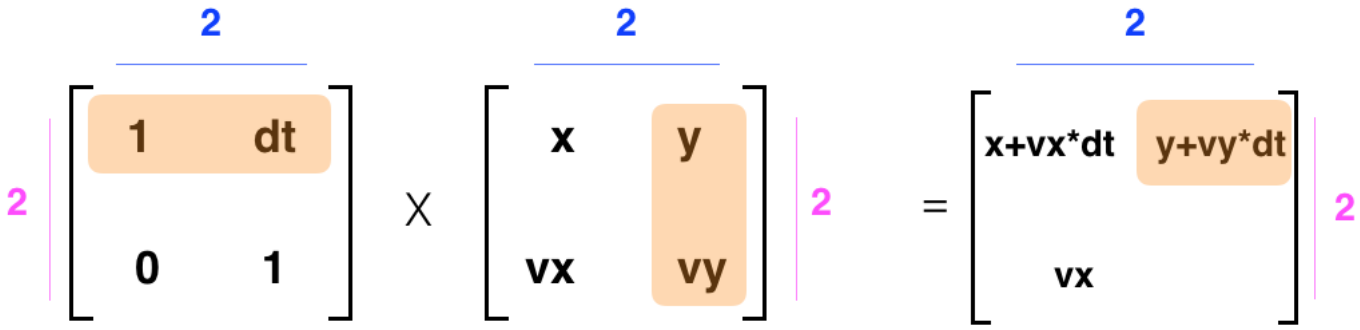
- Multiply the first row by the first column and sum.



- Then, the second row, by the first column.



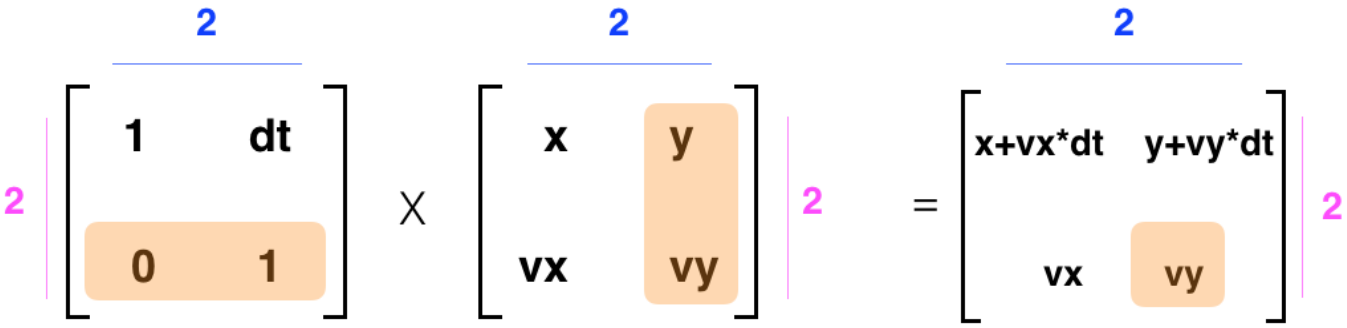
- Then *back* to the first row, this time, multiplied by the second column.



And, finally the last step:

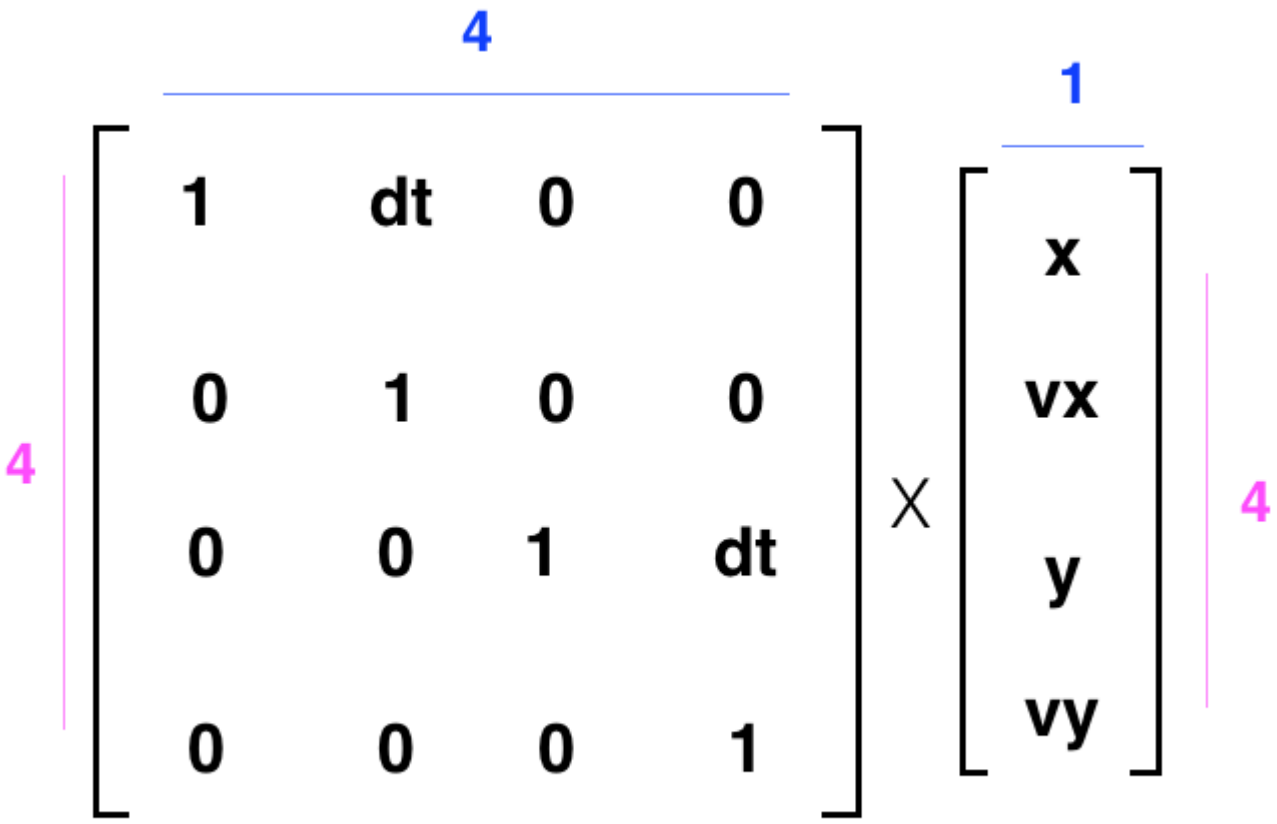
- The last row multiplied by the last column.

To get our complete, resulting matrix!



Constant velocity

This kind of multiplication can be really useful, if x and y are not dependent on one another. That is, there is a separate and constant x-velocity and y-velocity component. For real-world, curved and continuous motion, we still use a state vector that is one column, so that we can handle any x-y dependencies. So, you'll often see state vector and transformation matrices that look like the following.



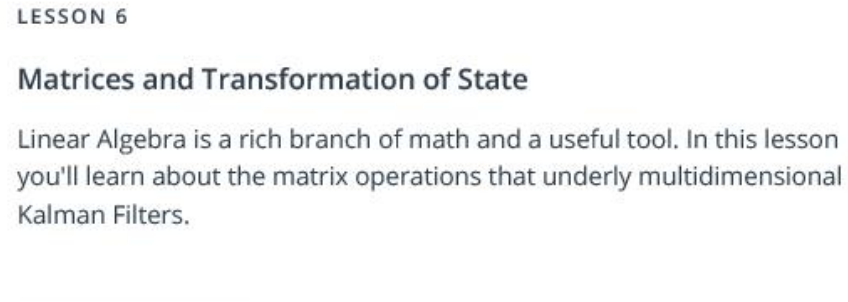
State vector equivalent

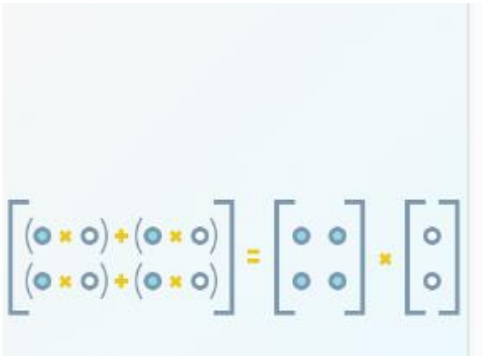
These extra spaces in the matrix allow for more detailed motion models and can account for a x and y dependence on one another (just think of the case of circular motion). So, **state vectors are always column vectors**.

- Notebook: Modify Predict State

https://www.youtube.com/watch?v=nruxu8pr6i8&feature=emb_logo

Lesson 6: Matrices and Transformation of state





https://www.youtube.com/watch?v=6xupqulu0bc&feature=emb_logo

Kalman Filter Prediction

https://www.youtube.com/watch?v=DjoBJNLzhj8&feature=emb_logo

https://www.youtube.com/watch?v=OelWLjfdSyw&feature=emb_logo

Another Prediction

https://www.youtube.com/watch?v=wdSkHh-6U4Y&feature=emb_logo

https://www.youtube.com/watch?v=RfzQ5MbVNeQ&feature=emb_logo

More Kalman FIlters

https://www.youtube.com/watch?time_continue=19&v=5QYGm4D9z6Y&feature=emb_logo

A Note on Notation

In the previous video (and in the next) you saw the following equation:

x′=x+x˙x' = x + \dot{x}x′=x+x˙

Translated into plain speech, this says

the x position **after** motion (x′x'x′) is equal to the x position **before** motion (xxx) plus the velocity in the x direction (x˙\dot{x}x˙).

If you read through that statement, you might notice that it doesn't quite make sense because it doesn't take into account the **duration** of motion. If I drive for 10 seconds I go farther than if I only drive for 1 second!

In the previous video we are assuming that the duration of motion (typically called Δt\Delta tΔt) is equal to 1 second. The "complete" version of the equation above would be

$$x' = x + \dot{x}\Delta t$$

Symbol	Meaning
x	x position **before** motion
x'	x position **after** motion
$\dot{x}$	velocity in x direction
Δt	duration of motion "delta t"

Kalman Filter Design

https://www.youtube.com/watch?time_continue=27&v=7C_tsAr8PNM&feature=emb_logo

What You've Learned

So far, you've really done a lot in this section of the course. You've:

- Programmed a **histogram filter**
- Learned about the sense/move cycle in a Kalman Filter
- Implemented a 1D Kalman Fillter

- Learned how to represent a car's motion and **state** in a **vector** that could be transformed

And you've just learned a but about 2D and multidimensional Kalman Filters.

Linear Algebra and the Kalman Equations

Now, Sebastian said "not to worry" about the complex Kalman equations that he's written, but this course is meant to be a good foundation for real-world programming challenges. So, the rest of this lesson will be about how to really understand these equations. Along the way, you'll learn a bit about how to approach a challenge like this and learn a lot of linear algebra that will allow you to read any equations like the Kalman filter equations with relative ease!

If you are already comfortable with linear algebra, feel free to skip around this section, otherwise, let's learn more about these equations and the powerful tool: linear algebra!

The Kalman Filter Equations

https://www.youtube.com/watch?time_continue=16&v=X9UUpk5URuw&feature=emb_logo

Simplifying the Kalman Filter Equations

https://www.youtube.com/watch?v=UpC0D-SEtD0&feature=emb_logo

Kalman Filter Equations Fx versus Bu

Consider this specific Kalman filter equation: x'=Fx+Bux' = Fx + Bu$x'=Fx+Bu$.

This equation is the move function that updates your beliefs in between sensor measurements. Fx models motion based on velocity, acceleration, angular velocity, etc of the object you are tracking.

B is called the control matrix and u is the control vector. Bu measures extra forces on the object you are tracking. An example would be if a robot was receiving direct commands to move in a specific direction, and you knew what those commands were and when they occurred. Like if you told your robot to move backwards 10 feet, you could model this with the Bu term.

When you take the self-driving car engineer nanodegree, you'll use Kalman filters to track objects that are moving around your vehicle like other cars, pedestrians, bicyclists, etc. In those cases, you would ignore BuBuBu because you do not have control over the movement of other objects. The Kalman filter equation becomes x'=Fxx' = Fx$x'=Fx$.

The Rest of the Lesson

The rest of this lesson will introduce you to **vectors**, **matrices** and the **operations** associated with them. Most of the instruction will be in the form of code demonstrations.

Depending on your background you may need to adjust your pace through the rest of this lesson. If you have a mathematical background then there's a good chance you'll move very quickly through this content. If you haven't worked with math in a while you may need to go more slowly.

A lot of people have bad memories with math and trying to learn something like linear algebra can sometime bring those memories back. If you start to feel stressed just step back, take a breath and remember that math is just another thing to learn. You can do it!

If you get stuck, don't hesitate to ask for help in the Study Groups or Knowledge. You've got classmates and staff who will be thrilled to help you learn

Representing State with Matrices

The State Vector

You just learned how to represent a self-driving car's state using a motion model. It turns out that matrices provide a very convenient and compact form for representing a vehicle's state.

Let's go back to the constant velocity motion model:

distance=velocity×timedistance = velocity \times timedistance=velocity×time

The vehicle's state is represented by the two variables distance and velocity. If you were going to store these two variables in Python, you'd probably use a list like this:

state = [distance, velocity]

That Python code looks a lot like a mathematical concept called a vector. A vector is essentially a list where each element in the list contains some information.

You could imagine that a state vector could have even more information. In a two-dimensional world, the state could have a distancexdistance_xdistancex, distanceydistance_ydistancey, velocityxvelocity_xvelocityx and a velocityyvelocity_yvelocityy.

In Python, the list would look like this:

state = [distance_x, distance_y, velocity_x, velocity_y]

Or an even more complex model could include information about the turning angle of the vehicle and the turning rate:

state = [distance_x, distance_y, velocity_x, velocity_y, angle, angle_rate]

State Vector in a One-Dimensional World

For now, consider the one-dimensional model with just distance and velocity.

state = [distance, velocity]

How did you calculate the distance and velocity of the vehicle over time when the velocity was constant? There were two different equations.

{distance=velocity×timevelocity=velocity \begin{cases} distance = velocity \times time \\ velocity = velocity \end{cases} {distance=velocity×timevelocity=velocity

Now, think about these equations in terms of state. For convenience, you can represent distance as xxx, velocity as vvv, and time as ttt.

Initial State

When the vehicle first starts moving, you can consider that t=t0t = t_0t=t0, x=x0x = x_0x=x0 and v=v0v = v_0v=v0. So the state vector is state0=[x0,v0]state_0 = [x_0, v_0]state0=[x0,v0].

First Time Step

What about after a certain amount of time has passed and now you are at a time t1t_1t1?

At t1t_1t1, the state vector is:

state1=[x1,v1]state_1 = [x_1, v_1] state1=[x1,v1]

According to the model formulas,

x1=x0+v0×(t1−t0)x_1 = x_0 + v_0 \times (t_1 - t_0)x1=x0+v0×(t1−t0)

and since velocity is constant:

v1=v0v_1 = v_0v1=v0.

Second Time Step

Then after the next time step t2t_2t2, the state vector is: state2=[x2,v2]state_2 = [x_2, v_2]state2=[x2,v2]

where

x2=x1+v0×(t2−t1)x_2 = x_1 + v_0 \times (t_2 - t_1)x2=x1+v0×(t2−t1)

and since velocity is constant v2=v0v_2 = v_0v2=v0.

A Better Way

The math so far is not too hard, right? You have a distance equation and a velocity equation. You plug in the previous state into each equation, the time lapse, and you get the new velocity and the new distance.

But imagine what will happen as your self-driving car model gets more complex. What happens when you have to take into account an x-direction, y-direction, x and y velocities, and steering angle and angular velocity? Or what about an even more complex model like a drone or helicopter that also has a z-direction?

Instead of updating your equations one by one, you can actually use vectors and matrices to do all of the calculations in just one step.

Updating State with Matrix Algebra

If you look back at the equation that updates the distance, you'll notice that distance depends on the previous distance, the initial velocity, and how much time has elapsed since the distance formula was updated. You end up with a generic function:

xt+1=xt+v0×(tt+1−tt)x_{t+1} = x_{t} + v_0 \times (t_{t+1} - t_{t}) xt+1=xt+v0×(tt+1−tt)

You can also write tt+1−ttt_{t+1} - t_{t}tt+1−tt as:

Δt\Delta tΔt

For a constant velocity model, the generic velocity equation becomes: vt+1=vtv_{t+1} = v_{t}vt+1=vt

How could you combine the x and v equations into one matrix algebra expression? The matrix algebra would look like this:

[xt+1vt+1]=[1Δt01]×[xtvt] \begin {bmatrix} x_{t+1} \\ v_{t+1} \end {bmatrix} = \begin {bmatrix} 1& \Delta t \\ 0 & 1 \end{bmatrix} \times \begin {bmatrix} x_{t} \\ v_{t} \end{bmatrix} [xt+1vt+1]=[10Δt1]×[xtvt]

Don't worry if you're not sure what this expression means or how to multiply these matrices. You will learn how in this lesson.

Notation

The matrix algebra equation you just saw is actually one part of the Kalman filter update equation.

Traditionally, the matrix operation:

[xt+1vt+1]=[1Δt01]×[xtvt] \begin {bmatrix} x_{t+1} \\ v_{t+1} \end {bmatrix} = \begin {bmatrix} 1& \Delta t \\ 0 & 1 \end{bmatrix} \times \begin {bmatrix} x_{t} \\ v_{t} \end{bmatrix} [xt+1vt+1]=[10Δt1]×[xtvt]

is represented by this notation for Kalman filters: x^k|k−1=Fx^k−1|k−1 \mathbf{\hat{x}_{k|k-1}} = \mathbf{F} \mathbf{\hat{x}_{k-1|k-1}} x^k|k−1=Fx^k−1|k−1

where x^\mathbf{\hat{x}}x^ is the state vector and F\mathbf{F}F is the matrix

[1Δt01] \begin {bmatrix} 1& \Delta t \\ 0 & 1 \end{bmatrix} [10Δt1]

k−1\mathbf{k - 1}k−1 is the previous step and k\mathbf{k }k is the current step.

You will see in the next part of the lesson why the notation contains k−1|k−1\mathbf{k-1|k-1}k−1|k−1 and k|k−1\mathbf{k|k-1}k|k−1 .

This notation can get a little bit confusing. For example, what is the difference between xxx and x^\mathbf{\hat{x}}x^?

The regular xxx would usually represent distance along the x-axis; on the other hand, the bold x^\mathbf{\hat{x}}x^ indicates a vector. In the one-dimensional case being discussed here, the x^\mathbf{\hat{x}}x^ vector contains two variables: distance along the x-axis and velocity; hence x^=[xv]\bold{\hat{x}} = \begin{bmatrix} x \\ v \end{bmatrix}x^=[xv].

Why is there a capitalized bold F\mathbf{F}F instead of f\mathbf{f}f? The capitalized, bold F\mathbf{F}F tells you that this variable is a matrix.

Kalman Equation Reference

We're just including this here in case you want to refer back to the Kalman Filter equations at any time.
Feel free to move along :)

Variable Definitions

$\hat{\mathbf{x}}$ - state vector

$\mathbf{F}$ - state transition matrix

$\mathbf{P}$ - error covariance matrix

$\mathbf{Q}$ - process noise covariance matrix

$\mathbf{R}$ - measurement noise covariance matrix

$\mathbf{S}$ - intermediate matrix for calculating Kalman gain

$\mathbf{H}$ - observation matrix

$\mathbf{K}$ - Kalman gain

$\tilde{\mathbf{y}}$ - difference between predicted state and measured state

$\mathbf{z}$ - measurement vector (lidar data or radar data, etc.)

$\mathbf{I}$ - Identity matrix

Prediction Step Equations

$$\hat{\mathbf{x}}_{\mathbf{k}|\mathbf{k}-1} = \mathbf{F}_{\mathbf{k}}\hat{\mathbf{x}}_{\mathbf{k}-1|\mathbf{k}-1}$$

$$\mathbf{P}_{\mathbf{k}|\mathbf{k}-1} = \mathbf{F}_{\mathbf{k}}\mathbf{P}_{\mathbf{k}-1|\mathbf{k}-1}\mathbf{F}_{\mathbf{k}}^{\mathbf{T}} + \mathbf{Q}_{\mathbf{k}}$$

Update Step Equations

KALMAN GAIN

$$\mathbf{S}_{\mathbf{k}} = \mathbf{H}_{\mathbf{k}}\mathbf{P}_{\mathbf{k}|\mathbf{k}-1}\mathbf{H}_{\mathbf{k}}^{\mathbf{T}} + \mathbf{R}_{\mathbf{k}}$$

$$\mathbf{K}_{\mathbf{k}} = \mathbf{P}_{\mathbf{k}|\mathbf{k}-1}\mathbf{H}_{\mathbf{k}}^{\mathbf{T}}\mathbf{S}_{\mathbf{k}}^{-1}$$

UPDATE STATE VECTOR AND ERROR COVARIANCE MATRIX

$$\tilde{\mathbf{y}_{\mathbf{k}}} = \mathbf{z}_{\mathbf{k}} - \mathbf{H}_{\mathbf{k}}\hat{\mathbf{x}}_{\mathbf{k}|\mathbf{k}-1}$$

$$\hat{\mathbf{x}}_{\mathbf{k}|\mathbf{k}} = \hat{\mathbf{x}}_{\mathbf{k}|\mathbf{k}-1} + \mathbf{K}_{\mathbf{k}}\tilde{\mathbf{y}_{\mathbf{k}}}$$

$$\mathbf{P}_{\mathbf{k}|\mathbf{k}} = (\mathbf{I} - \mathbf{K}_{\mathbf{k}}\mathbf{H}_{\mathbf{k}})\mathbf{P}_{\mathbf{k}|\mathbf{k}-1}$$

What is a vector? Physics versus Computer Programming

You might have learned at some point that a vector is a measurement or quantity that has both a **magnitude** and a **direction**. Examples might be distance along a y-axis or velocity towards the north-west.

But in computer programming, when we say "vector" we are just referring a **list of values**.

These two ways of thinking about vectors are actually deeply related, but for this Nanodegree we're going to look at vectors from a computer science point of view.

Vectors, Motion Models and Kalman Filters in Self-Driving Cars

In a physics class, you might have one vector for position and then a separate vector for velocity. But in computer programming, a vector is really just a list of values.

When using the Kalman filter equations, the bold, lower-case variable x\mathbf{x}**x** represents a vector in the computer programming sense of the word. The x\mathbf{x}**x** vector holds the values that represent your motion model such as position and velocity.

In a basic motion model, the vector x\mathbf{x}**x** will contain information about position and linear velocity like: x=[x,y,vx,vy]\mathbf{x} = [x, y, v_x, v_y]**x**=[*x*,*y*,*vx*,*vy*]. In a physics class, these might be represented with two different vectors: a position vector and a velocity vector.

A more complex motion model might take into account yaw rate, which considers a rotational angle and angular velocity around the center of the vehicle like x=[x,y,vx,vy,ψ,ψ˙]\mathbf{x} = [x, y, v_x, v_y, \psi, \dot{\psi}]**x**=[*x*,*y*,*vx*,*vy*,*ψ*,*ψ*˙].

So in order to use the Kalman filter equations and execute object tracking, you have to be familiar with vectors and how to write programs with them.

Vectors in Python

In Python, you can represent a vector with lists. So a vector like [5,9,10,2,20]\begin{bmatrix}5, 9, 10, 2, 20\end{bmatrix}[5,9,10,2,20] could be represented with

myvector = [5, 9, 10, 2, 20]

Vector Indexing

If you wanted to access values inside the vector, you would use indexing. The first value, which in this case is 5, would be accessed by

myvector[0]

The second value:

myvector[1]

And so on.

Assigning Values to Vectors

If you wanted to change a value in the vector, you could use this syntax:

myvector[3] = 19

which would change the fourth element from 20 to 19.

To add a value to the end of a vector, you can use the .append() method:

myvector = [5, 9, 2, 20]

myvector.append(18)

and the resulting vector would be [5, 9, 2, 20, 18].

Vector Math in Python

There are a few vector operations you'll want to become familiar with in order to use the Kalman filter equations:

- vector addition
- scalar multiplication
- the dot product

Vector Addition

To add two vectors together, you sum them element by element. For example,

v1=[a,b,c,d]\mathbf{v_1} = [a, b, c, d]v1=[a,b,c,d]

v2=[w,x,y,z]\mathbf{v_2} = [w, x, y, z]v2=[w,x,y,z]

The sum of these two vectors would then be:

v1+v2=[a+w,b+x,c+y,d+z]\mathbf{v_1} + \mathbf{v_2} = [a + w, b + x, c + y, d + z]v1+v2=[a+w,b+x,c+y,d+z]

Say that you know the current state of your vehicle x=[x,y,vx,vy]\mathbf{x} = [x, y, v_x, v_y]x=[x,y,vx,vy]

If you knew the change in the position and velocity of your vehicle, you could use vector addition to find the new state vector. You will code this in the vector coding exercises.

Vector addition Python Code

How might you execute vector addition using Python? In general, for loops are very useful for accessing values inside of a Python list:

```
for i in range(len(v1)):
    v1[i]
```

would give you access to v1, one element at a time. So you could access each value of v1 and v2, sum them together, and then append each sum to a new vector like:

```
# initialize an empty vector
```

```
vsum = []


# use for loops to take the sum of a value and then append to vsum
for ....
    .....
    vsum.append(value)
```

We are not showing you the full answer because you will be figuring this out in a coding exercise.

Scalar Multiplication

Scalar multiplication involves multiplying each element in a vector by a constant.

Here is a concrete example.

If v=[a,b,c,d]\mathbf{v} = [a, b, c, d]v=[a,b,c,d], then 5v=[5a,5b,5c,5d]5\mathbf{v} = [5a, 5b, 5c, 5d]5v=[5a,5b,5c,5d]

You will also implement scalar multiplication in the coding exercises. Think about how you can use a for loop and the append method to code scalar multiplication.

A use case for scalar multiplication could be changing between units of measurement. For example, if your state were measured in meters, you could use scalar multiplication to convert to feet. Get ready to code this in the vector coding exercises!

Dot Product

The dot product of two vectors is very important for matrix multiplication. If

v1=[a,b,c,d]\mathbf{v_1} = [a, b, c, d]v1=[a,b,c,d]

v2=[w,x,y,z]\mathbf{v_2} = [w, x, y, z]v2=[w,x,y,z],

then the dot product would be

v1·v2=aw+bx+cy+dz\mathbf{v_1 \cdot v_2} = aw+ bx + cy + dzv1·v2=aw+bx+cy+dz

This is another place where a for loop would be useful. You will figure out how to code the dot product in the exercises in the next section.

You'll see how to apply the dot product and why it is useful in the coding exercises.

Summary

Here is a summary of what you will need to know about Python lists to complete the coding exercises:

- assigning a vector - myvector = [5, 9, 10 2, 20]
- accessing a value - myvector[3]

- changing a value - myvector[3] = 15
- appending a value to the end of the list - myvector.append(6)
- and finally, using for loops to access and manipulate vectors:

```
for i in range(len(myvector)):
    myvector[i]
```

Check out this link to learn about other methods that come with Python lists. You might find them useful although you will not need them for the following exercises.

Next, you will practice coding vectors in Python.

Some python maths

- Notebook: 1_vector_coding
- Notebook: 1_vector_coding
- Notebook: 2_matrices_in_python
- Notebook: 3_matrix_addition
- Notebook: 4_matrix_multiplication
- Notebook: 5_matrix_transpose
- Notebook: 7_identity_matrix
- Notebook: 6_inverse_matrix

What to Take Away from this Lesson

You just went through a lot of math! Remember: you don't need to memorize everything that was covered in this lesson.

https://www.youtube.com/watch?v=BRLmFGScL_k&feature=emb_logo

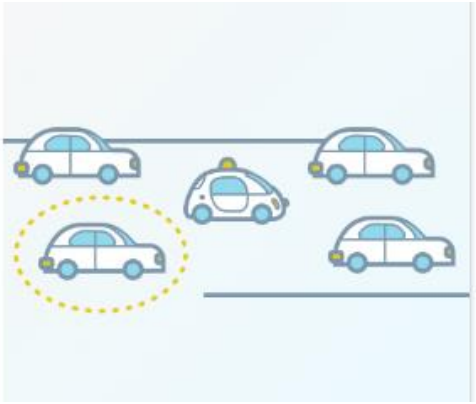
Lesson 7: SLAM

LESSON 7

Simultaneous Localization and Mapping

Learn how to implement SLAM: simultaneously localize an autonomous vehicle and create a map of landmarks in an environment.

VIEW LESSON →



Simultaneous Localization and Mapping

In the previous lessons, you learned all about localization methods that aim to locate a robot or car in an environment given sensor readings and motion data, and we would start out knowing close to nothing about the surrounding environment. In practice, in addition to localization, we also want to build up a model of the robot's environment so that we have an idea of objects, and landmarks that surround it *and* so that we can use this map data to ensure that we are on the right path as the robot moves through the world!

In this lesson, you'll learn how to implement SLAM (Simultaneous Localization and Mapping) for a 2 dimensional world! You'll combine what you know about robot sensor measurements and movement to create locate a robot *and* create a map of an environment from only sensor and motion data gathered by a robot, over time. SLAM gives you a way to track the location of a robot in the world in real-time and identify the locations of landmarks such as buildings, trees, rocks, and other world features.

https://www.youtube.com/watch?v=UVkkDPJshgM&feature=emb_logo

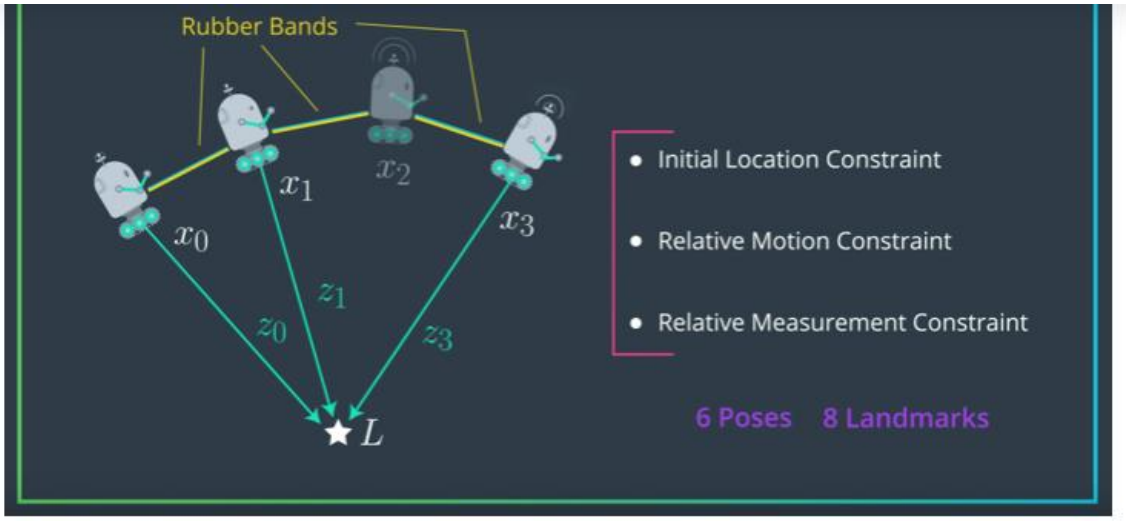
QUIZ:

https://www.youtube.com/watch?v=OGBC9HrVqd4&feature=emb_logo

QUIZ QUESTION

For a series of robot motions and landmark locations, how many constraints will there be for 6 total poses and 8 landmarks?

- 12
- ✓ 14
- 32
- 48



Constraints

For our 6 poses, we have:

- 1 initial location constraint
- 5 additional, relative motion constraints, and finally,
- 8 relative measurement constraints for the 8 landmark locations

Adding all of these up gives us a total of **14** constraints.

Now, consider the image above, with 4 poses (including the initial location `x0`) and one landmark. We can use this same math and say that there are 5 total constraints for the given image, but in practice there are usually many more measurements and motions!

Implementing Constraints

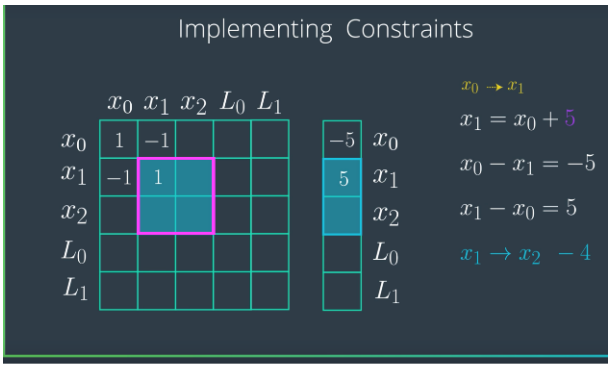
You also may have noticed that not all of these constraints will provide us with meaningful information, such as in our example: we do not have a measurement between the pose `x3` and the landmark location. Next, let's see how we can represent these constraints in a **matrix** and their relationships with values in that matrix and a constraint **vector**.

Constraint Matrices

Next, you'll see how to implement relationships between robot poses and landmark locations. These matrices should look familiar from the section of linear algebra, but I also find it helpful to think of the values in these matrices as **weights** kind of like you've seen in convolutional kernels, only these weights imply how much a pose or landmark should be weighted in a set of equations.

https://www.youtube.com/watch?v=2V3ZF08TcX8&feature=emb_logo

See below for the quiz.



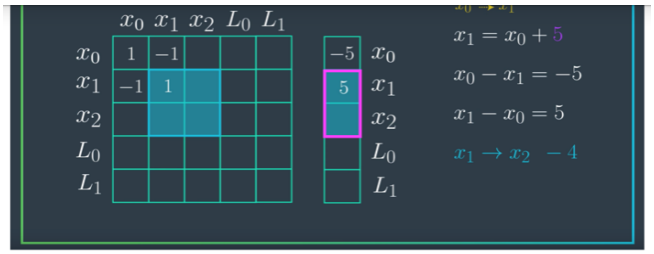
QUESTION 1 OF 2

For the highlighted values in the square above, what will the new weights relating x1 and be after this motion update? They start as:

```
[ 1  0 ]
[ 0  0 ]
```

Which can also be read as: `[ 1 0 ], [ 0 0 ]`

- `[ 0 1 ], [ -1 1 ]`
- `[ 1 -1 ], [ 0 0 ]`
- ✓ `[ 2 -1 ], [ -1 1 ]`



QUESTION 2 OF 2

For the highlighted values in the small rectangle above, what will the new values relating x1 and x2 be after this motion update? They start as:

```
[ 5 ]
[ 0 ]
```

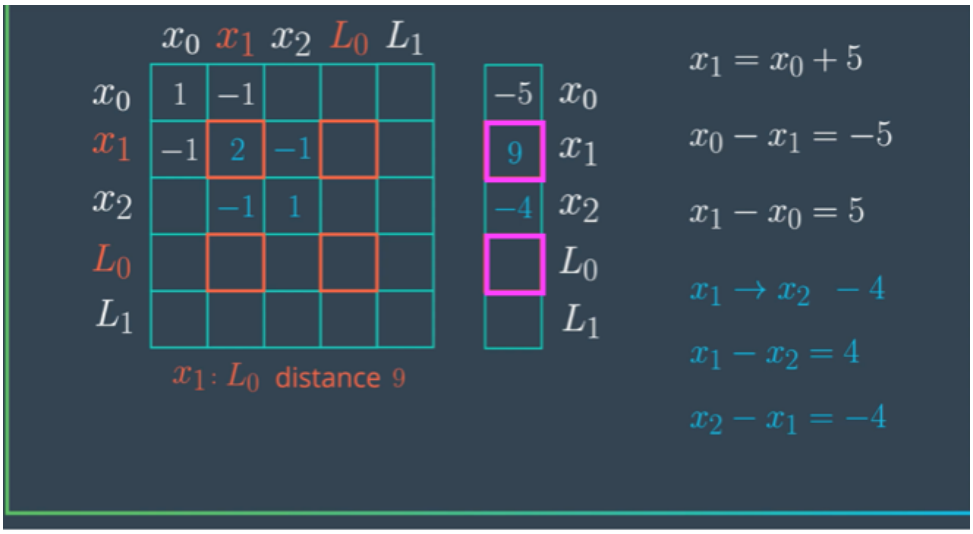
Which can also be read as: `[ 5 ], [ 0 ]`

- `[ 0 ], [ 5 ]`
- `[ 1 ], [ 4 ]`
- `[ 1 ], [ -4 ]`
- `[ 9 ], [ 5 ]`
- ✓ `[ 9 ], [ -4 ]`

https://www.youtube.com/watch?v=uLmGavXEN64&feature=emb_logo

QUIZ:

https://www.youtube.com/watch?v=_JUCLtoh1CE&feature=emb_logo



QUIZ QUESTION

For the two highlighted values in the column vector above, what will there new value be after this landmark relationship is added to these constraint matrices? Right now the values are 9, 0.

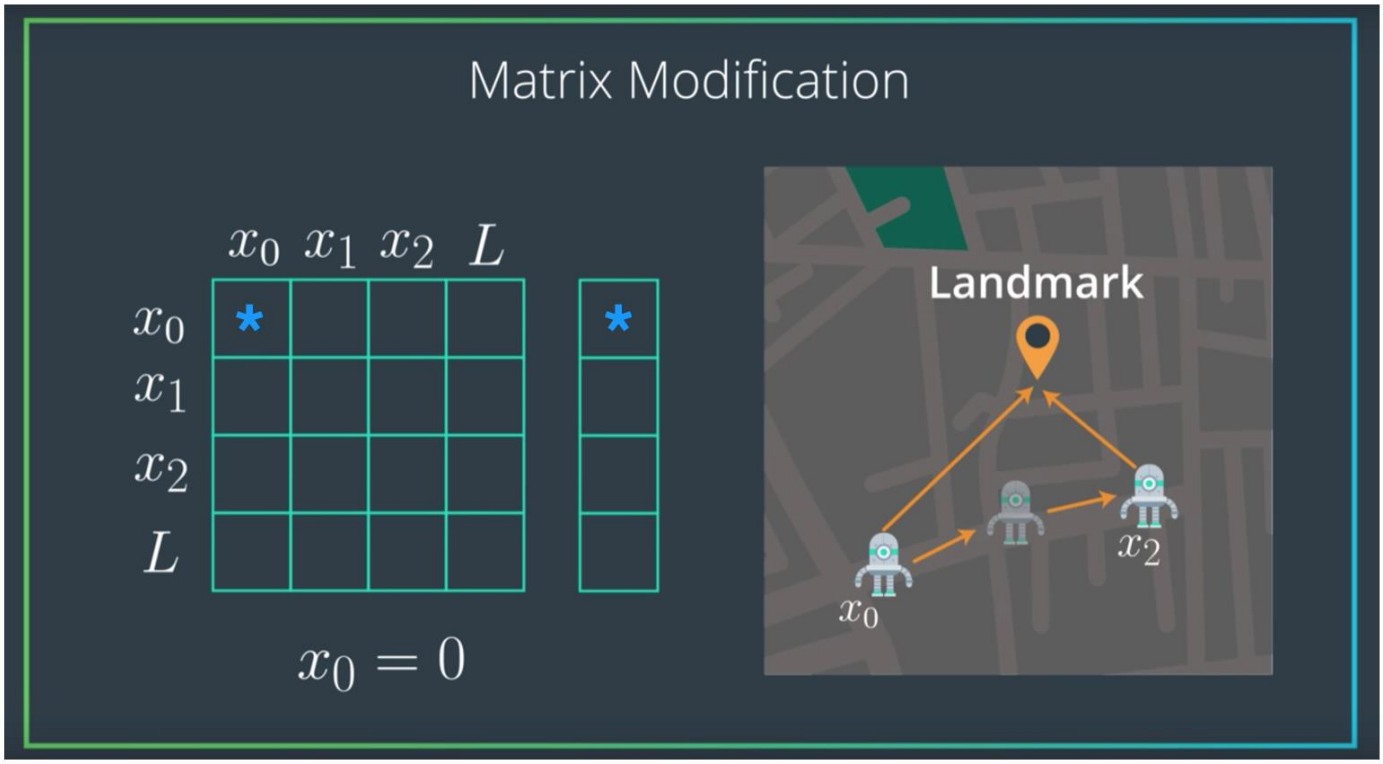
- 9, 0
- 18, 9
- -5, 9

✓ 0, 9

https://www.youtube.com/watch?time_continue=2&v=vDfQNdUSclA&feature=emb_logo

Quiz: Matrix Modification

https://www.youtube.com/watch?v=e3zpWlM9IRg&feature=emb_logo



Mark the Relationships

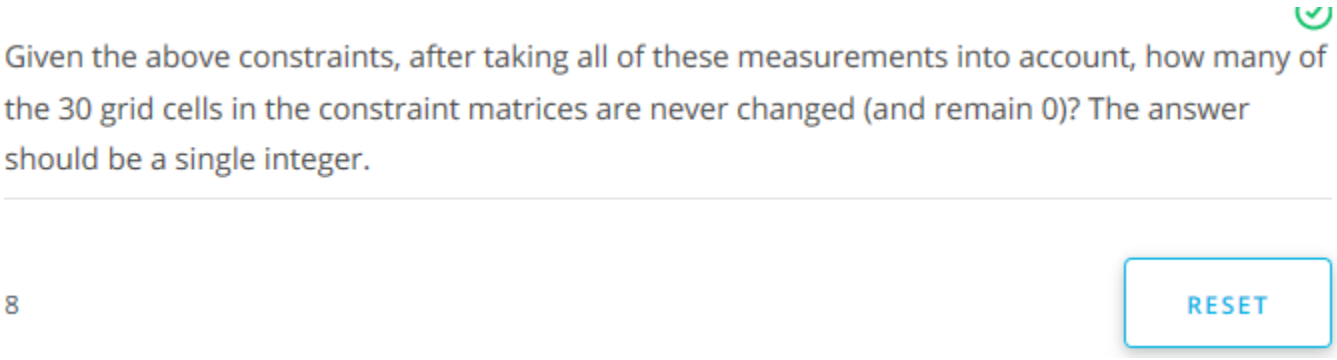
For this quiz, take out a piece of paper and draw these 4x4 and 4x1 grids. Draw a dot (or other mark) wherever there is a relationship between poses and/or landmarks. If nothing will change about a cell, it will remain 0 and you can leave it blank.

An example has been started in the image below, note that the initial constraint has been marked for you!

https://www.youtube.com/watch?v=ntK0oz35VPQ&feature=emb_logo

Quiz: Untouched Fields

https://www.youtube.com/watch?v=V9qbWSqqKwg&feature=emb_logo

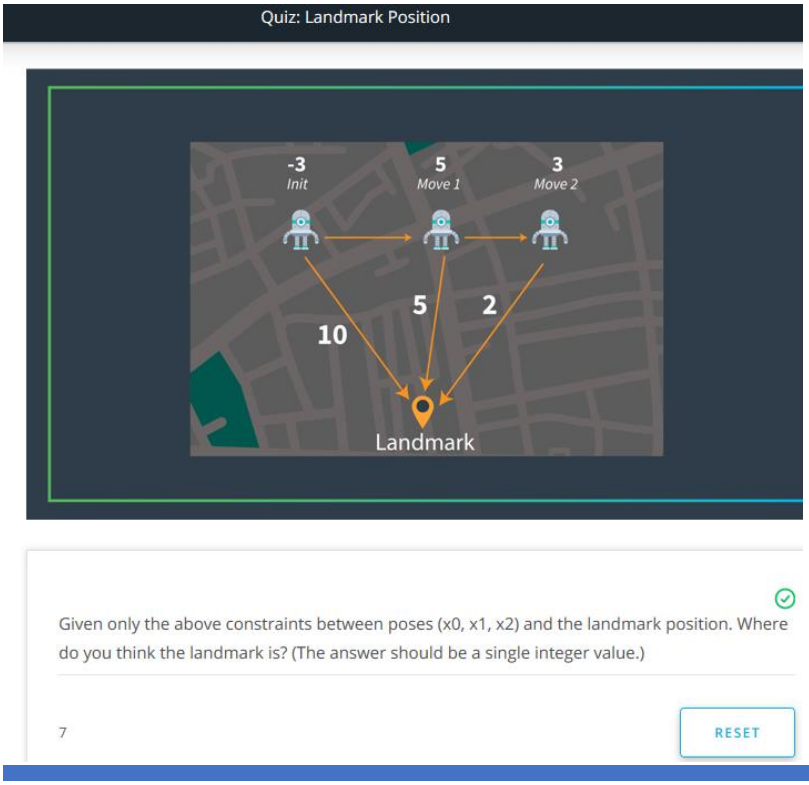


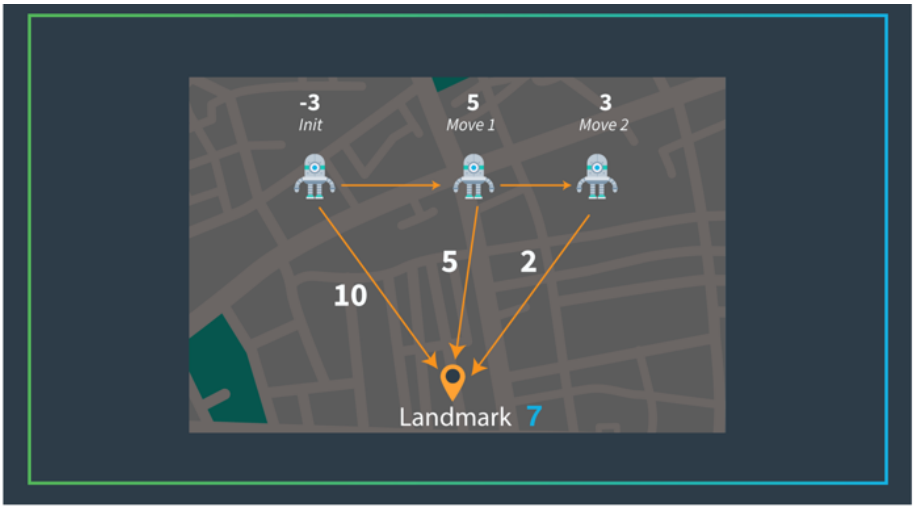
https://www.youtube.com/watch?v=cKFEdCwBGqo&feature=emb_logo

Quiz: Omega and Xi

https://www.youtube.com/watch?v=KqCBoAaa5jQ&feature=emb_logo

- Notebook: Omega and Xi





Landmark Position

The landmark, L, is at the value 7.

Think about the relationship between x0 and L. x0 is at -3 and sees L a distance of 10 away: `L = -3 + 10`.

And if we keep adding poses and sensor measurements this remains consistent. As x0 moves to x1, we get `L = -3 + 5 + 5`

And the final loop all the way to x2: `L = -3 + 5 + 3 + 2`

- **Notebook: Including Sensor Measurements**

Quiz: Introducing Noise

https://www.youtube.com/watch?v=E_OI5DinFA0&feature=emb_logo

For the above scenario, what is the effect on `x0`?

- ○ x0 < -3
- ✓ x0 = -3

https://www.youtube.com/watch?v=X5kh82oke5w&feature=emb_logo

Confident Measurements

https://www.youtube.com/watch?v=WRANOBm89I4&feature=emb_logo

- Notebook: Confident Measurements

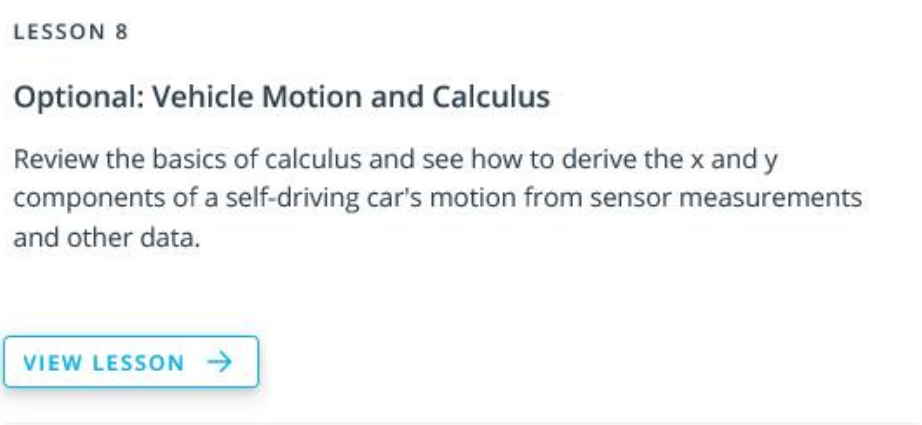
Implementing SLAM

Now you have all the information you need to complete a 2D version of SLAM. You know how to update constraint matrices and incorporate strength and noise into a model of motion and sensor measurements.

Next, you'll be given the following variable and asked to complete an implementation of SLAM for your final project, Congratulations on getting this far!

https://www.youtube.com/watch?v=Ptx33qEaUQQ&feature=emb_logo

Lesson 8: Vehicle Motion and Calculus



Introduction to Odometry

Vehicle Motion and Controls

This is an optional lesson that will give you more practice with reading sensor data from a self-driving car and determining the car's trajectory and path over time using calculus. This type of motion analysis is used in autonomous vehicle control systems, and you may find this section fun and useful if that is of interest to you!

It is **optional** because none of the following material will be necessary for you to complete the final project in this course, so if you are not interested or want to wait and come back to this material, that is up to you!

Next, hear from Sebastian about how vehicle controls and *odometry* work.

https://www.youtube.com/watch?v=vWgG0d2HOVE&feature=emb_logo

Navigation Sensors

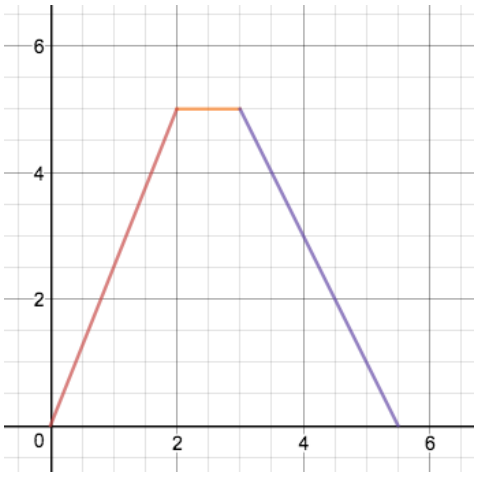
We will be discussing the following sensors in this course:

- **Odometers** - An odometer measures how far a vehicle has traveled by counting wheel rotations. These are useful for measuring distance traveled (or *displacement*), but they are susceptible to **bias** (often caused by changing tire diameter). A "trip odometer" is an odometer that can be manually reset by a vehicle's operator.
- **Inertial Measurement Unit** - An Inertial Measurement Unit (or **IMU**) is used to measure a vehicle's heading, rotation rate, and linear acceleration using magnetometers, rate gyros, and accelerometers. We will discuss these sensors more in the next lesson.
- Notebook: Plotting Position vs Time

Interpreting Position vs. Time Graphs

The graph below shows position on the **vertical axis** (in meters) vs time on the **horizontal axis** (in seconds) for a car moving in one dimension in a parking lot. The next few questions refer to this graph.

https://www.youtube.com/watch?v=NAEasXHN_PU&feature=emb_logo



Which of the following is the best verbal description of the vehicle's motion?

- The car drove up a hill which goes up, then flat, then down.
- The car drove up and to the right, then turned to the right, then drove down and to the right.
- ✓ The car drove forward, stopped, then backed up.

What was the car's average speed from $t = 0$ to $t = 2$?

- 0 m/s
- 2 m/s
- ✓ 2.5 m/s

QUESTION 3 OF 3

What was the car's average speed from t=0 to t=5.5?

- ✓ 0 m/s

A "Typical" Calculus Problem

If you've taken calculus before you probably have vague recollections of terms like "the chain rule" or "the product rule" or "the quotient rule"... these are all techniques for calculating the derivative of a function when you know the function's algebraic form. Calculating the derivative of a function is also known as differentiation. And in a typical calculus class you would spend a LOT of time learning how to differentiate various functions...

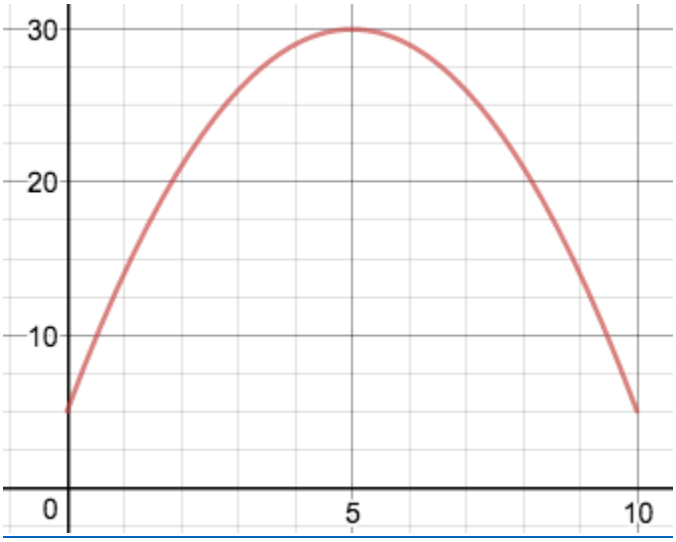
Below you will see a differentiation problem that is similar to what you might find in a typical calculus class. Read through this problem and think about what it's asking. Afterwards I'm going to explain why we are **not** going to spend time solving these kinds of problems in this course.

Differentiation Example Problem

1. A self driving car's position is described as a function of time from t=0t=0*t*=0 to t=10t=10*t*=10 by the following equation:

x(t)=−t2+10t+5x(t) = -t^2 + 10t + 5*x*(*t*)=−*t*2+10*t*+5

What is the derivative, x˙(t)\dot{x}(t) *x*˙(*t*) of this function? A graph of x(t)x(t) *x*(*t*) is included below.

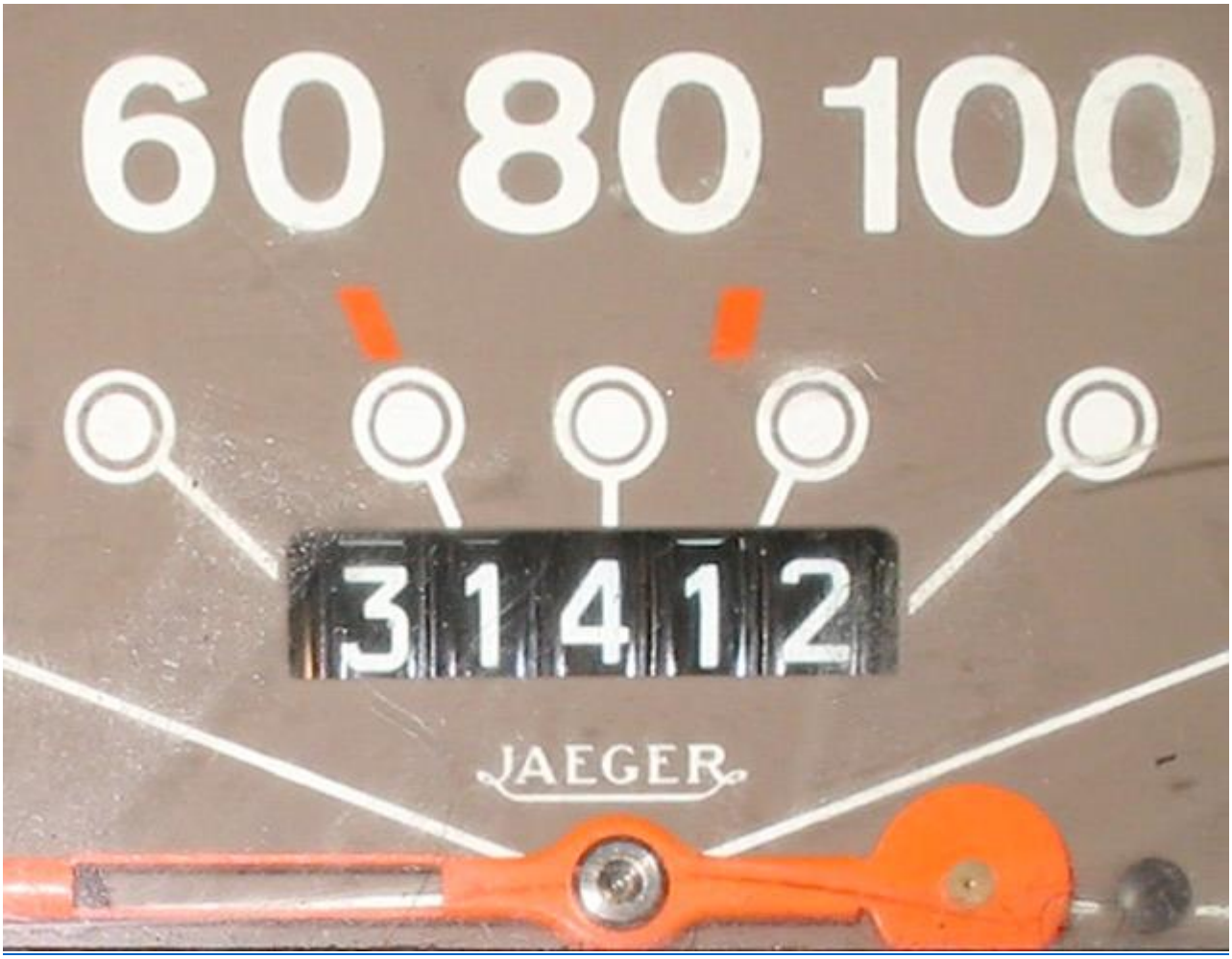


Graph of x(t)=−t2+10t+5x(t) = -t^2 + 10t + 5*x*(*t*)=−*t*2+10*t*+5

https://www.youtube.com/watch?v=0ww_q51P8uY&feature=emb_logo

How Odometers Work

A mechanical odometer works by coupling the rotation of a vehicle's wheels to the rotation of numbers on a gauge like this:



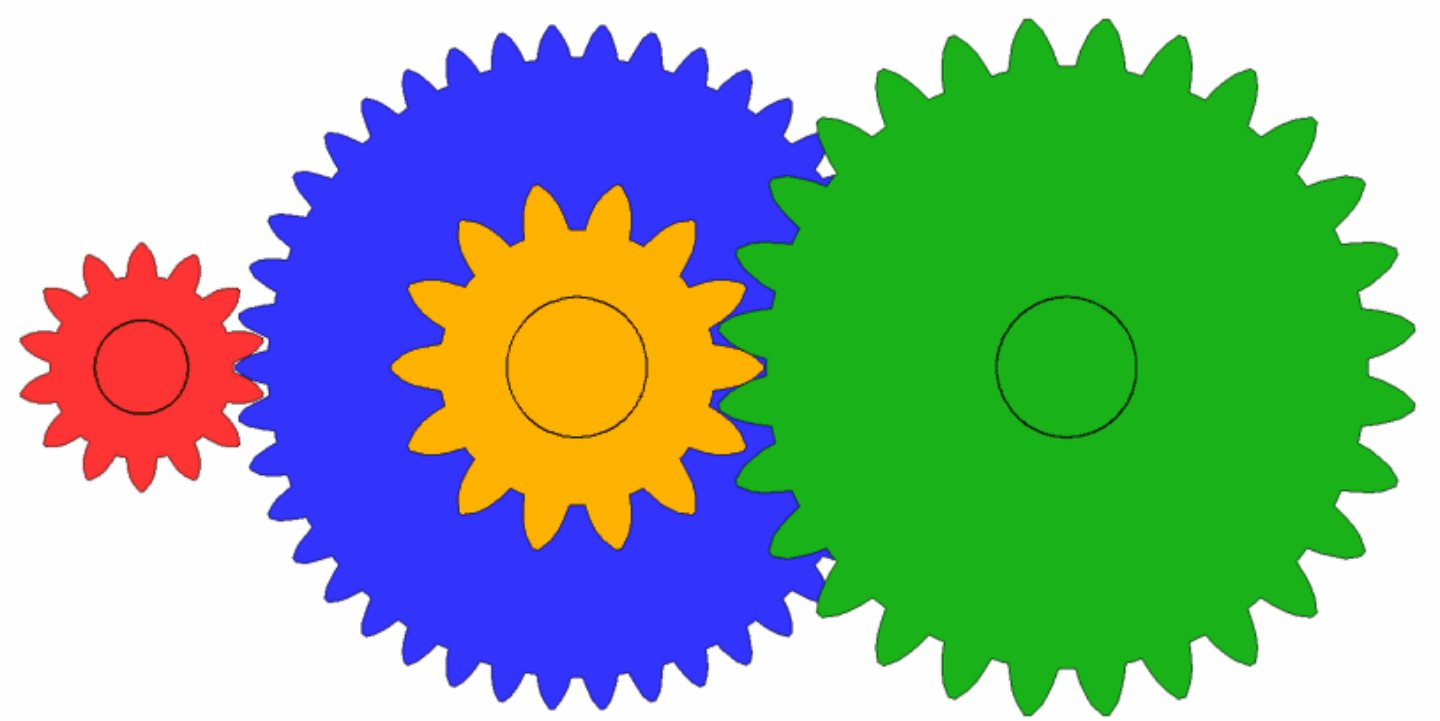
Each of these numbers is written on a dial which has the numbers 0 - 9 written on it.

But that last digit on the gauge needs to rotate **very slowly** compared to the rotation rate of the vehicle's tires. Typically, a car's wheels will have to complete **750** rotations to move 1 mile. And since there are 10 digits on each dial, that means the last digit should only complete one rotation after the wheels have completed **7,500** rotations!

We can even express this mathematically:

$$\frac{\Delta \theta_\text{odometer}}{\Delta \theta_\text{tires}} = \frac{1}{7,500}$$ $\Delta\theta_\text{tires}\Delta\theta_\text{odometer}=7,5001$

This reduction of rotation rate is accomplished through *gear reduction*. If you look at the blue and green gears in the image below you should get a sense for how that works.



You don't need to remember any of this. Digital odometers don't even work this way (though they are pretty cool too).

I just think animations of gears are too fun to pass up!

- Notebook: Speed from Position Data

Position, Velocity, and Acceleration

Allow me to say the same thing about **position** and **velocity** in 5 different ways.

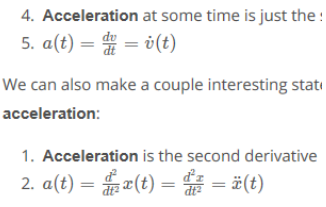
1. **Velocity** is the derivative of **position**.
2. **Velocity** is the *instantaneous rate of change of **position** with respect to **time***.
3. An object's **velocity** tells us how much it's **position** will change when time changes.
4. **Velocity** at some time is just the slope of a line tangent to a graph of **position** vs. **time**
5. v(t)=dxdt=x˙(t)v(t) = \frac{dx}{dt} = \dot{x}(t)v(t)=dtdx=x˙(t)

It turns out you can say the same 5 things about **velocity** and **acceleration**.

1. **Acceleration** is the derivative of **velocity**.
2. **Acceleration** is the *instantaneous rate of change of **velocity** with respect to **time***.
3. An object's **acceleration** tells us how much it's **velocity** will change when time changes.
4. **Acceleration** at some time is just the slope of a line tangent to a graph of **velocity** vs. **time**
5. a(t)=dvdt=v˙(t)a(t) = \frac{dv}{dt} = \dot{v}(t)a(t)=dtdv=v˙(t)

We can also make a couple interesting statements about the relationship between **position** and **acceleration**:

1. **Acceleration** is the second derivative of **position**.
2. a(t)=d2dt2x(t)=d2xdt2=x¨(t) a(t) = \frac{d^2}{dt^2}x(t) = \frac{d^2x}{dt^2} = \ddot{x}(t)a(t)=dt2d2 x(t)=dt2d2x=x¨(t)



We'll explore this more in the next lesson. For now, just know that differentiating position twice gives acceleration!

- Notebook: Implement an Accelerometer

Differentiation Recap

In the last lesson, you learned about the **derivative**. This section is just here to remind you of what you learned.

Understanding the Derivative

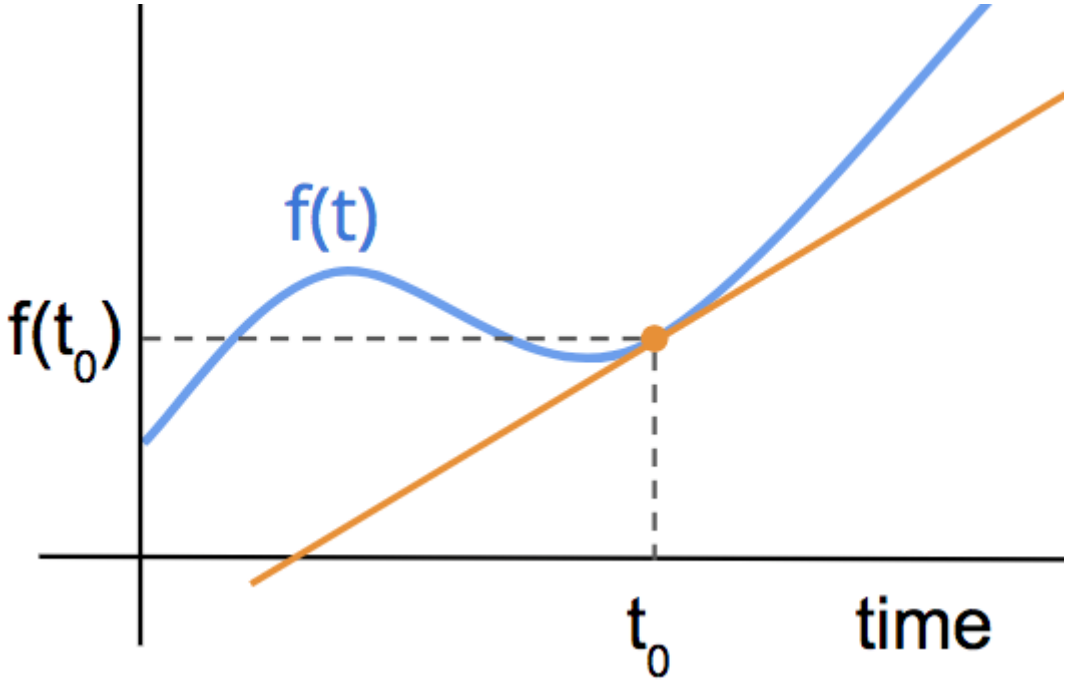
You saw a few ways to understand the derivative:

1. The "Rate of Change" Interpretation

If f(t)f(t)f(t) gives the value of a function at **any** ttt, then f˙(t0)\dot{f}(t_0)f˙(t0) gives the instantaneous rate of change of f(t)f(t)f(t) at the value t=t0t=t_0t=t0.

2. The Graphical Interpretation

The slope of the line tangent to f(t)f(t)f(t) at t=t0t=t_0t=t0 is f˙(t0)\dot{f}(t_0)f˙(t0)



The slope of the orange line is equal to the derivative of *f* at t=t_0

3. The Formal Definition

The *formal* mathematical definition is the following:

The **derivative** of a function f(t)f(t)f(t) is the function f˙(t)\dot{f}(t)f˙(t) (or dfdt\frac{df}{dt}dtdf), and is defined as:

f˙(t)=limΔt→0f(t+Δt)−f(t)Δt\dot{f}(t) = \lim_{\Delta t \to 0}\frac{ f(t + \Delta t) - f(t)}{\Delta t}f˙(t)=Δt→0limΔtf(t+Δt)−f(t)

The **derivative** of a function $f(t)$ is the function $\dot{f}(t)$ (or $\frac{df}{dt}$), and is defined as:

$$\dot{f}(t) = \lim_{\Delta t \to 0} \frac{f(t + \Delta t) - f(t)}{\Delta t}$$

Derivatives and Motion

Position, **velocity**, and **acceleration** are all useful quantities when describing a vehicle's motion and these quantities are related through the derivative.

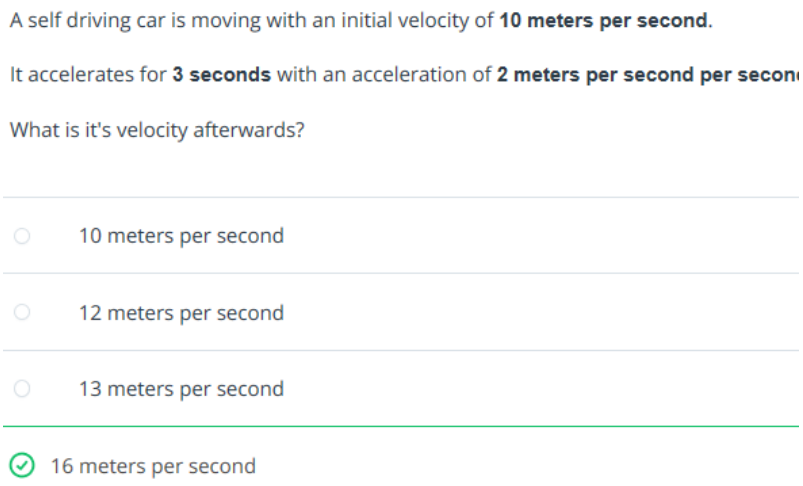
- velocity is the derivative of position
  - v(t)=x˙(t)v(t)=\dot{x}(t)v(t)=x˙(t)
- acceleration is the derivative of velocity and the second derivative of position.
  - a(t)=v˙(t)=x¨(t)a(t) = \dot{v}(t) = \ddot{x}(t)a(t)=v˙(t)=x¨(t)

Coding the Derivative

```
def get_derivative_from_data(position_data, time_data):
    """
    Calculates a list of speeds from position_data and
    time_data.

    Arguments:
      position_data - a list of values corresponding to
        vehicle position

      time_data     - a list of values (equal in length to
        position_data) which give timestamps for each
        position measurement

    Returns:
      speeds        - a list of values (which is shorter
        by ONE than the input lists) of speeds.
    """
    # 1. Check to make sure the input lists have same length
    if len(position_data) != len(time_data):
        raise(ValueError, "Data sets must have same length")

    # 2. Prepare empty list of speeds
    speeds = []

    # 3. Get first values for position and time
    previous_position = position_data[0]
    previous_time     = time_data[0]

    # 4. Begin loop through all data EXCEPT first entry
    for i in range(1, len(position_data)):

        # 5. get position and time data for this timestamp
        position = position_data[i]
        time     = time_data[i]

        # 6. Calculate delta_x and delta_t
        delta_x = position - previous_position
        delta_t = time - previous_time

        # 7. Speed is slope. Calculate it and append to list
        speed = delta_x / delta_t
        speeds.append(speed)

        # 8. Update values for next iteration of the loop.
        previous_position = position
        previous_time     = time

    return speeds
```

Acceleration Basics

https://www.youtube.com/watch?v=ea6b4PZ7YXU&feature=emb_logo



- Notebook: Plotting Elevator Acceleration

Reasoning About Two Peaks

https://www.youtube.com/watch?v=bgPS1GO1EJs&feature=emb_logo

I exaggerated in the video when I said the peaks look like mirror images. In fact the first peak is wider but not as tall (in absolute value) while the second peak is "peakier".

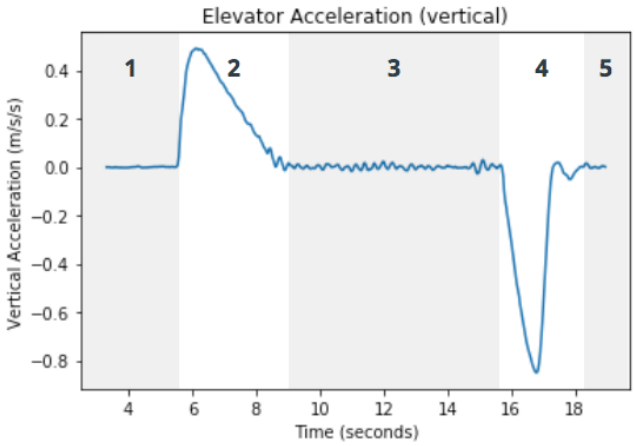
But that just makes it even more amazing that the area under each of these curves still winds up being equal!

The Integral: Area Under a Curve

Finding Area Practice

The following four questions are based on graphs of vehicle velocity vs time. For these graphs the horizontal axis is in units of **seconds** and the vertical axis is in units of **meters / second**.

https://www.youtube.com/watch?time_continue=3&v=Nhpvh2dolcE&feature=emb_logo



https://www.youtube.com/watch?v=QNvgwoIrsIE&feature=emb_logo

Approximating the Integral

https://www.youtube.com/watch?time_continue=1&v=9C05AHzI_I8&feature=emb_logo

- Notebook: Approximating the Integral

- Notebook: Integrating Accelerometer Data

Rate Gyros

https://www.youtube.com/watch?v=TmnecSf80b0&feature=emb_logo

- Notebook: Integrating Rate Gyro Data

Working with Real Data

https://www.youtube.com/watch?v=WvEcWtAz-OQ&feature=emb_logo

- Notebook: Accumulating Errors

Sensor Strengths and Weaknesses

As you just saw, it can be dangerous to rely on accelerometer data for localization since errors have a tendency to accumulate. This is a weakness of accelerometers.

Fortunately, it takes some time for these errors to accumulate. So when they're used over short time intervals accelerometers can be really helpful.

Take a look at one of Uber's early prototypes of a self driving car:

Look at all those sensors perched on the hood! And you can't even see the IMUs and odometers and GPS sensors inside the vehicle!

Each of these sensors has strengths and weaknesses and each contributes to an improved understanding of the vehicle and it's surroundings.

If you choose to take the advanced Self Driving Car Engineer Nanodegree, you'll learn about **sensor fusion**. Sensor fusion is how you use software to stitch together all these disparate data sources into one coherent picture about the vehicle, its motion, and the world around it.

From Calculus to Trigonometry

By now, you should feel comfortable working with sensor data and using what you know about derivatives and integrals to calculate quantities that interest you when it comes to vehicle motion. These kinds of calculations come up all the time in autonomous vehicle navigation and the more you practice these concepts the better you'll become and building up an understanding of motion and controls.

Next, I want to take a step back from calculus and briefly revisit trigonometry and the idea of **motion vectors** from the start of this section. Then, you'll get a chance to practice what you've learned on complete trajectory data!

Trigonometry and Vehicle Motion

https://www.youtube.com/watch?time_continue=4&v=WY3T-9GHI_0&feature=emb_logo

Solving Trig Problems

https://www.youtube.com/watch?time_continue=1&v=qI4i845d7Qg&feature=emb_logo

- Notebook: Keeping Track of x and y

Conclusion:

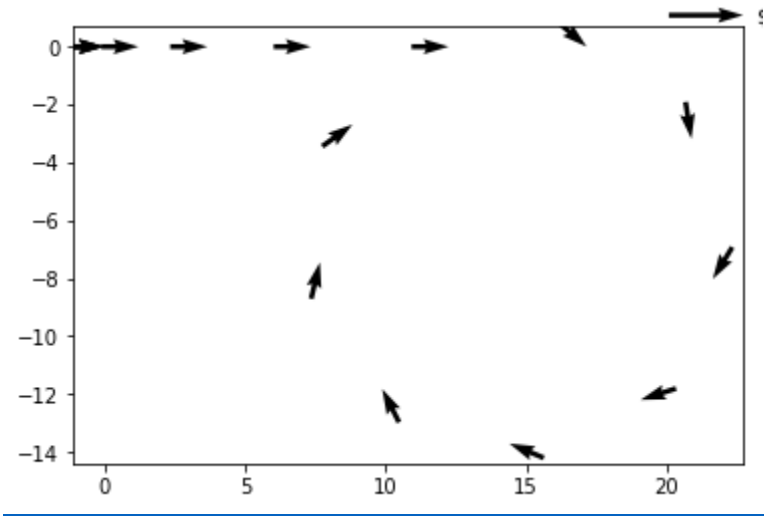
https://www.youtube.com/watch?v=gMbDqd4ItiU&feature=emb_logo

Lab Overview

In this project you will take raw sensor data like this:

timestamp	displacement	yaw_rate	acceleration
0.0	0	0.0	0.0
0.25	0.0	0.0	19.6
0.5	1.225	0.0	19.6
0.75	3.675	0.0	19.6
1.0	7.35	0.0	19.6
1.25	12.25	0.0	0.0
1.5	17.15	-2.829	0.0
1.75	22.05	-2.829	0.0
2.0	26.95	-2.829	0.0
2.25	31.85	-2.829	0.0
2.5	36.75	-2.829	0.0
2.75	41.65	-2.829	0.0
3.0	46.55	-2.829	0.0
3.25	51.45	-2.829	0.0
3.5	56.35	-2.829	0.0

and turn it into plots of vehicle trajectories like this:



Data Note

The above is just example data, and not from real sensors - as you might notice, acceleration of 19.6 m/s^2 from 0.25 to 0.5 seconds would result in a velocity changing from 0 to 4.9 m/s. If this was a consistent velocity, the displacement would be the velocity (4.9) multiplied by the change in time (delta t, or 0.25), resulting in displacement of 1.225, as shown above. However, in reality it would likely be closer to constant acceleration (instead of constant velocity) during this period from 0 to 4.9 m/s velocity, so displacement would actually be 0.6125. We've simplified the calculations for our examples in this lab.

Instructions

Instructions for this project are included in the project notebook in the next section.

Note that this project is **not** submitted and will not be reviewed by a Udacity reviewer. This means that you **can** complete this Nanodegree without completing this project (but we recommend you complete it anyway).

If you want to ask questions about this project or share advice and code, post in the Study Groups or Knowledge.

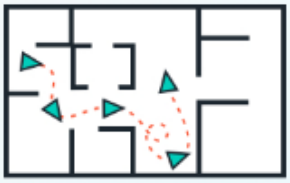
- **Notebook: Reconstructing Trajectories**

Project: Landmark detection and tracking

PROJECT

Project: Landmark Detection & Tracking (SLAM)

Implement SLAM, a robust method for tracking an object over time and mapping out its surrounding environment, using elements of probability, motion models, and linear algebra.



Landmark Detection & Robot Tracking (SLAM)

Project Overview

In this project, you'll implement SLAM (Simultaneous Localization and Mapping) for a 2 dimensional world! You'll combine what you know about robot sensor measurements and movement to create a map of an environment from only sensor and motion data gathered by a robot, over time. SLAM gives you a way to track the location of a robot in the world in real-time and identify the locations of landmarks such as buildings, trees, rocks, and other world features. This is an active area of research in the fields of robotics and autonomous systems.

Below is an example of a 2D robot world with landmarks (purple x's) and the robot (a red 'o') located and found using only sensor and motion data collected by that robot. This is just one example for a 50x50 grid world; in your work you will likely generate a variety of these maps.

Example of SLAM output (estimated final robot pose and landmark locations).

Project Instructions

The project will be broken up into three Python notebooks; the first two are for exploration of provided code, and a review of SLAM architectures, **only Notebook 3 and the robot_class.py file will be graded**:

Notebook 1 : Robot Moving and Sensing

Notebook 2 : Omega and Xi, Constraints

Notebook 3 : Landmark Detection and Tracking

You can find these notebooks in the Udacity workspace that appears in the concept titled **Project: Landmark Detection & Tracking**. This workspace provides a Jupyter notebook server directly in your browser.

You can also choose to complete this project in your own local repository, and you can find all of the project files in this **GitHub repository**. Note that while you are *allowed* to complete this project on your local computer, you are strongly encouraged to complete the project from the workspace.

Evaluation

Your project will be reviewed by a Udacity reviewer against the Landmark Detection & Tracking project **rubric**. Review this rubric thoroughly, and self-evaluate your project before submission. All criteria found in the rubric must meet specifications for you to pass.

Zip file submission

If you are submitting this project from a workspace or as a zip file, it is recommended that you follow the instructions in the final workspace notebook to compress your project files and make sure your submission is not too large. You are *only* graded on Notebook 3, and the file robot_class.py.

Ready to submit your project?

Click on the "Submit Project" button and follow the instructions to submit!

Meets Specifications

Well done on completing the SLAM project, hope you had fun working on it! Happy learning and stay udacious