- IMU: Inertial Motion unit
  - acceleration along x, y, z
  - rotation rates around α, γ, ζ (pitch, roll, yaw)

- Vehicle heading and displacement

- Navigation Sensors
  - **Odometers**:
    Measures how far the vehicle has traveled by counting the wheel rotations.
    Useful for measuring distance traveled (displacement) but are biased because of changing tire diameters
    "trip odometer" can be reset by vehicle operator.
  - **IMU**
    measures heading, rotation rate, and linear acceleration using magnetometer, rate gyros, accelerometer

- Average Speed: $v_{avg} = \frac{\Delta q}{\Delta t}$

(1:30 h)

L2: Accelerators, rate gyros and Integrals

- Use integral of acceleration to find velocities and integral of velocities to find displacement.
- **Rate Gyros** measure **angular velocity** $\dot{\theta}$
  Integrating the angular velocity gives us **heading** θ
- yaw rate: $\dot{\theta}$

- Real Data is always flawed!
- accelerometers are often **biased**
  - careful calibration
  - better sensors
  - i.e. they measure some non-zero value when the acceleration is zero.
- The effect of **Error Accumulation** is drastic when integrating over longer time intervals and/or double integrating!
- In order to avoid error accumulation, we should use acceleration data only over short time intervals.

L3: Two Dimensional Robot Motion and Trigonometry
(1:24 h)

Computer Vision and Machine Learning

L1: Computer Vision and Classification (3:41 h)

- **Spacial Coherancy:**
  - As long as the data has spatial coherency we can apply CV techniques to process it
  - The data for CV algorithms does not need to necessarily come from cameras.
- **Spatially Coherent Data:**

  Any data that predictably varies over space.

  Like sound for example. The volume of the sound can give us spatial information. The louder the sound, the nearer the source.
- SDCs use in addition to cameras

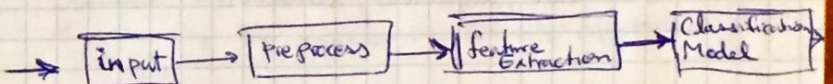
  Sonars, Radars, and lidars

  Radar D. and R. ↙ Light Detection and Ranging ↘

- Radars, Lidars ⟶ long-range detection

  Cameras ⟶ rich sensory input
- Radar, Lidars → **Active sensors:** they sense the environment by transmitting energy.

  Cameras → **Passive sensors:** can only sense the environment based on energy (photons) already in existing in the scene.
- Part of a good design in a CV system is knowing what not to do, even if it is possible.
- Image Classification pipeline



- cropping image `img[a:-a, b:-b, :]`

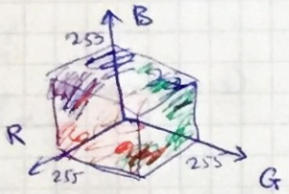
  a = height pixels to crop
  b = width pixels to crop
- mask:

  lower-green = [0, 100, 0]
  upper-green = [100, 255, 100]

  `mask = cv2.inRange(img, lower-green, upper-green)`
  `masked_img = np.copy(img)`
  `masked_img[mask != 0] = [0,0,0]` → black [0,0,0]

- Masking using simple RGB threshold would not work well under
  - variant light conditions
  - needs consistent green color

- Color Spaces
  - RGB color space



  - HSV color space
    - Hue, Saturation, Value
      - Hue → color
      - Saturation → intensity
      - Value → Brightness
  - HLS
    - Hue, lightness, saturation

Consider brightness as a separate component

- Because HSV and HLS separate color and brightness into separate channels they are very useful in image processing.

- HSV conversion:

```
hsv = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
h = hsv[:,:,0]   <- remains consistent under diff. lighting conditions.
s = hsv[:,:,1]   } vary a lot under diff. lighting condition !
v = hsv[:,:,2]   }
```


- Classification Labels:
  - categorical (e.g. dog, day, ...) vs. numerical values
    - integer encoding
    - one-hot encoding

- Two main Types of Features in a image:
  - Color-based features
  - Shape-based features

- High-pass filter:

  Detect big changes in intensity over a small area.

0	-1	0
-1	4	-1
0	-1	0

  - The weights sum to zero
  - The center weight is the highest
  - The neighbor weights are negative

  } Detect Edges

High-pass filter Kernel

Edge: where intensity changes abruptly

- The edge detection filter computes the difference (or change) between neighboring pixels.

  It is an approximation of Derivative of image over space

- A kernel for detecting horizontal lines:

-1	2	-1
0	0	0
1	2	1

- finds the difference between the top and bottom edges surrounding a given pixel.

- **Sobel Filter**: for edge detection

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

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

Taking an approx. of **Image Derivative**

- **Frequency** in an Image is the **Rate of change**

- Images change in space. A high frequency image is where the intensity changes a lot, and the level of brightness changes quickly from one pixel to another.
- A low frequency image is a one where brightness changes slowly or is relatively uniform.
- Most images have both high frequency and low frequency components.
  - High frequency components correspond to edges of the object in an image

- **Roberts Cross Operators**:

(AKA: Roberts Cross)

- simple, quick to compute, 2-D spatial gradient measurements on an image
- highlights the regions of high spatial frequency, which often correspond to edges.

- Spacial Domain
- Frequency Domain

$$G_x \begin{pmatrix} +1 & 0 \\ 0 & -1 \end{pmatrix} \qquad G_y \begin{pmatrix} 0 & +1 \\ -1 & 0 \end{pmatrix}$$

- diagonal gradients. Designed to respond maximally to edges at 45°
- due to small kernel size, susceptible to noise

- **Histograms**:

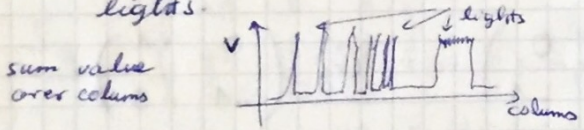
```
hsv = cv2.cvtColor(rgb_img, cv2.COLOR_RGB2HSV)

h_hist = np.histogram(hsv[:,:,0], bins=32, range=(0,180))
s_hist = np.histogram(hsv[:,:,1], bins=32, range=(0,256))
v_hist = np.histogram(hsv[:,:,2], bins=32, range=(0,256))

bin_edges = h_hist[1]
bin_centers = (bin_edges[1:] + bin_edges[0:len(bin_edges)-1])/2

fig = plt.figure(figsize=(12,3))
plt.subplot(131)
plt.bar(bin_centers, h_hist[0])
plt.xlim(0,180)
plt.title("H Histogram")
⋮ # rest two graphs for S and V histograms
```

- Summation to create Feature Vectors

  - To keep spatial ~~information~~ it is also useful to to sum up pixel values alongs columns or rows.

    ⇒ Spike in various colors or intensity values over space.

Example: A night image has often many high intensity lights.



Self Driving Car Engineer :)

Projects:

Part 1:
- ✓ - Finding Lane Lines
- ✓ - Advanced Lane Finding
- ✓ - Traffic Sign Classifier
- ✓ - Behavioral Cloning
- ✓ - Extended Kalman Filters

Part 2:
- ✓ - Kidnapped Vehicle
- - Highway Driving
- - PID Controllers
- ○ - Linked In Profile
- ○ - GitHub Profile
- - Programming a real self Driving Car

Part 1: computer vision, Deep learning, Sensors (EKF, UKF) classification Behavior Cloning (end-to-end Learning)

Part 2:
Localization, Path Planing, Control, Sys. Integration

Lesson 1: Introduction

Sebastian Thron

- 2003 DARPA Challenge (Stanley
- 2009 the Stanley team joined Google

- David Silver (worked at Ford)
- Ryan Keenan (astro physisist)

- Two distinct approaches to Autonomous Development:
  - Robotic Approach

    Fuse sensors to measure vehicles Surroundings and then navigate accordingly
  - Deep learning

    learn to Mimic human behavior using deep learning.

- Content
  1. Computer Vision, Deep Learning, Sensor Fusion
  2. Localization, Path Planning, Control, System Integration

Projects:

P1 • Find lane Markings

Lane Markings tell you where to drive

P2 • Advanced lane finding

P3 • Traffic Sign Classifier

P4 • Behavioral Cloning

clone human behavior using deep learning on camera images

P5 • Extended Kalman Filter

Sensor fusion

P6 • Kidnapped Vehicle

localization

P7 • Path Finding

Path planning: come up with a sequence of actions. City navigation, Parking lot navigation

P8 • PID controller

Control: steer the actuators to meet an objective

P9 • Put your code in a real SLD car

1

2

Extra Cricular

- Additional Content

  Unscented Kalman Filters, Model Predictive Control, Advanced Deep Learning / Semantic Segmentation, Functional Safety

- Autonomous Systems Intervie
- Technical Interview Prep

Career Portal

- Career track (tips)
- Career Services

- Feedback on Resume, Cover letter, LinkedIn, GitHub
- Career track tips, informational interviews
- Udacity profile for Talent Program.

Udacity Talent Program

- graduate
- complete Udacity Profile
- Upload resume and make it Public.

A Timeplan for Studying and Objectives in the next days

[I don't download the course Material for the time being to save time]

- L4: Computer Vision foundamentals (3h 46m)
- P1: finding lanes 28-30. March 2020 (2 days)
- L6: Camera Calibration (2 hours) 31. March
- L7: Gradients and Color Spaces (2h) 01. April
- L8: Advanced CV (2h) 02. April
- P2: Advanced Lane Finding 03. April - 06. April (3 days)
- L10: Neural Networks (3h) 07. April, 22:00 - 09. April, 22:00
- L11: Tensor Flow (2h) 09. April 22:00 - 11. April 21:30
- L12: Deep NN (2h) 11. April 21:30 - 12. April 20:15
- L13: CNNs (2h) 12. April 20:15 - 14. April 23:30
- L14: LeNet for traffic signs (2h) 15. April

12th
- P3: Traffic sign classifier 15. April 21:30 - 24. April 23:30 - 26. April (9 days)
- L16: Keras (2h) 26. April 18:30 - 20:30
- L17: Transfer Learning (1h 30) 26. April 20:30 - 27. April 23:59

15th
- P4: Behavioral ~~Cloning~~ Cloning 28. April - 05. May (7 days)
- L20: Sensors (0:30m) 05. May 23:00 - 00:30
- L21: Kalman Filters (0:32m) 06. May 21:00 - 08. May 21:00
- L23: Geometry & Trigonometry Refresher (8 min)

L24: Extended Kalman Filters (4 hours) (5 days) May 09 22:00 May 14 21:00

19th P5: Extended Kalman Filters Project May 14 21:00 May 18 20:00

Localization, Path Planning, Control, and System Integration

L1: Intro to Localization (19 min) 19. May 18:00

L2: Markov localization (3 h) 20 May 17:00 22 May 20:00

L3: Motion Models (30 min) 21. May

L4: Particle Filters (3 hours) 24. May 21:30 25. May 20:30

L5: Implementation of a particle filter (1 h) 26. May

26th P6: Kidnapped Vehicle 27 May - 28 May ~~17:00~~

L7: Search (1 h 30 m) May 28./29. May

L8: Prediction (1 h 30 m) 01. June 02. June 17:00

L9: Behavior Planing (2 h) 02. June 18:00 03. June 20:00

L10: Trajectory Generation (2 h) (5 days) 03. June 08. June

03 May P7: Highway Driving 09. June 22. June (14 days)

L12: PID control (2 h) 26. June

P8: PID controller 29. June 19:30 30. June

{ GitHub, LinkedIn }

72

← L16: Autonomous Vehicle Architecture (30 m) 01. July

L17: Introduction to ROS (1 h) 01 July

L18: Packages and Catkin Workspace (1 h) 02. July 20:00 22:00

L19: Writing ROS nodes (1 h) 03. July

P9: Program an Autonomous Vehicle 08. July 14. July

15th May L21: Complete the Program 15. July

L4: Computer Vision Fundamentals (3h:46m)

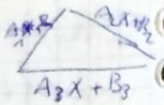
- Feature that could help in identifying lane lines in an image of a highway
  - color, shape, orientation, position in image
- image thresholding
- Region ~~of~~ Masking

  masking the region of interest in the image under the assumption that the camera is placed at a fixed position in ~~the~~ front of the car and

  => the lane lines will always appear in the same general region of the image

73

- np.polyfit( ) => $A_1 x + B_1$



np.meshgrid( ) => XX, YY

region_thresholds = (YY > (A_1 * XX + B_1))
& (YY > (A_2 * XX + B_2))
& (YY < (A_3 * XX + B_3))

- Both masks

result_image [~color_thresholds & region_thresholds] = [255, 0, 0]

draws the lane lines in the RoI red.

- Canny Edge detection
  - gray → gradient → threshold for following the gradients.
  - find edges by tracing out the pixels that follow the strongest gradients.
  - rapid changes in brightness in a gray scale image are edges.
  - gradient $\frac{df}{dxdy} = \Delta$(pixel value)

    (a measure of changes in the image function)

image: $f(x,y)$ = pixel value

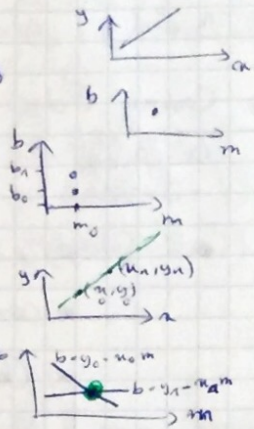
$\frac{df}{dx} = \Delta$(pixel)

- computing the gradient (derivitive with respect to x and y) gives us thick edges.
- Canny Edge detections gives us thin edges as pixels
  - Steps:
    - convert image to gray
    - Gaussian Blur : suppress noise and spurious gradients by averaging
    - kernel_size = 3

      blur_gray = cv2.GaussianBlur(gray, (kernel_size, kernel_size), 0)
    - run Canny

      cv2.Canny(blur_gray, low_thresh, high_threshold)
    - ratio between $\frac{\text{low\_threshold}}{\text{high\_threshold}} \approx 1:2$ or 1:3
    - - gradient intensities over high_thresh are "sure edges"
      - gradient intensities below low_thresh are "none edges"
      - for each point in between, do connectivity check: keep those that are connected

- Paul Hough $y = mx + b$
  - image space line $y = m_0 x + b_0$
  - Hough space (m_0, b_0)
  - parallel lines in Hough space
  - A point in (x,y) image space corresponds to a line in Hough space



$$b = y_0 - x_0 m$$
$$b = -x_0 m + y_0$$

- An intersection point of two lines in Hough space corresponds to a line passing through two points in image space.

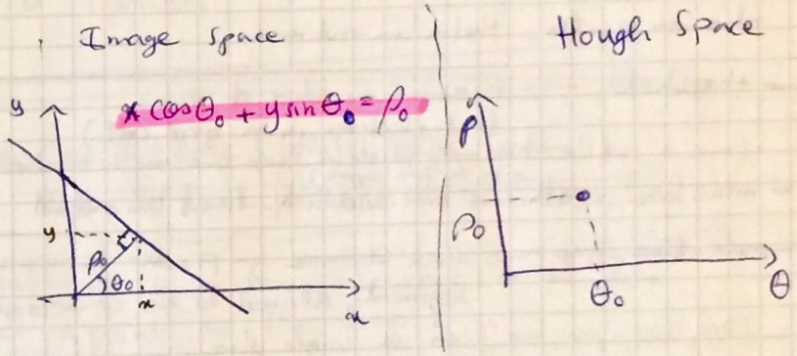
- Strategy to find line in image space, find intersection of lines in Hough space.

⇒
- First do Canny to detect edges.
- Every point on an edge corresponds to a line in Hough space
- Where many lines intersect in Hough space, we have found a line in image space that connects all those points.

Problem! Vertical lines have infinite slope. in (m, b) representation.

$$y = mx + b$$

⇒ Let's represent a line in image space in polar coordinates



- this representation has the advantage that we can represent any line (vertical lines too).
- now each point in image space corresponds to a sin curve in Hough space
- A line of points in Imag space, translates to a whole bunch of sin curves in Hough space

⇒ find intersection of sin curves

- cv2.HoughLinesP(masked_edges, rho, theta, threshold, np.array([]), min_line_length, max_line_gap)

  - rho } distance and angular resolution of the grid in Hough space
  - theta } Search for intersections

  rho in pixels, theta in radians

  - threshold ⇝ minimum number of votes (intersection in a grid cell)
    (i.e. how many points in image space must be associated with the line segment)
  - min_line_width ⇝ min width of lines in output
  - max_line_gap ⇝ max distance in pixels between segments allowed to be connected in a single line.

- cv2.fillPoly(img, pts, color)
  pts: clockwise or counterclockwise list of points defining the polygon.

- Parameter Tuning → a big challenge!

P1: Finding Lane Lines

28-30.03.20

- Gray, blur
- Canny
- RoI Mask
- Hough
- Group segments
- fit lane line based on segments
- draw lane lines

- cv2.fitLine(points, DIST, param, r_eps, a_eps)
  cv2.fitLine(mid_points, cv2.DIST_L2, 0, 0.01, 0.01)

returns a vector colinear to the line
a point on the line: [vx, vy, x0, y0]

the line equation

$(x, y) = (x_0, y_0) + t\,(v_x, v_y)$ for t from -∞ to +∞

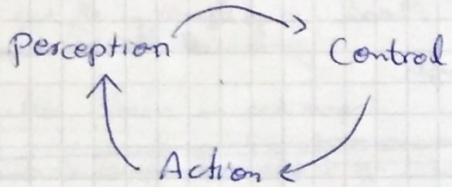
we need to find t in image bondaries to clip the line:

```
t0 = (0 - y0) / vy
t1 = (img_height - y0) / vy
p0 = (line[2:4] + t0 * line[0:2]).astype(np.uint32)
p1 = (line[2:4] + t1 * line[0:2]).astype(np.uint32)
cv2.line(img, p0, p1, thickness, color)   # draw the line
```

L6: Camera Callibration

31.03.20

Robotic Cycle:



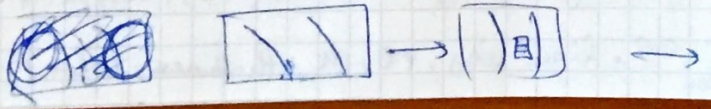
- 80% of the challenge of building a SDC is perception — Seb. Thron.

- Why use cameras over Radars and Lidars?

Sensors	Spatial Resolution	Cost	3D
Radar + Lidar	Low	$$$	Yes
Single Camera	High	$	No

- To steer the car we need to know how much the lane is curving.

↝ transform perspective to above



- But to do that we have to first fix the camera distortion

- Camera's don't create perfect images
  Objects near the edges of the camera are stretched and skewed

- IMPORTANT:

  If the lens is distorted, we will get the wrong measurement for lane curvature.
  So our steering commands will be wrong !

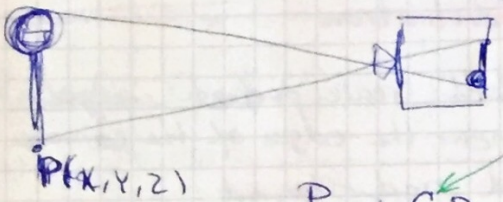
- Distortion results in X :(
  - change of apparent geometry of objects
    ↳ size, shape
  - make objects appear closer or farther away
  - make object appereance change depending on where the object is in the field of view.

- Pinhole Camera Model

↝

← Pinhole Camera Model

- image plane Π
- principal point $c \in \Pi$
- center of projection
- optical axis
- focal length



$P \sim CP$ ← Camera Matrix

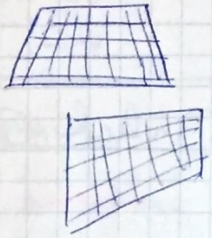
$3D \rightsquigarrow 2D$

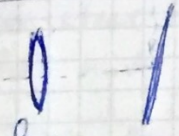
- Real cameras don't use pinholes, but lenses to create images faster. (by bending multiple light rays at a time)
- lenses cause distortion, because light is bent to much or too little at the edges of the lens.
  $\Rightarrow$ lines appear more curved than they are

Radial Distortion

- most common type of distortion

Tangential Distortion



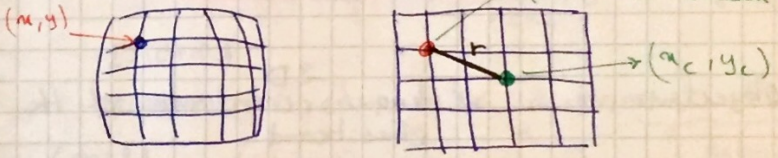


- Most distortions can be captured by 5 numbers: the distortion coefficients

$$(k_1, k_2, \underbrace{p_1, p_2}_{\text{tangential}}, \underbrace{k_3}_{\text{radial}})$$

represent radial and tangential distortions in an image.

- sometimes even more coefficients are required to describe the distortion, in severe cases.
- distortion coeffs $\rightsquigarrow$ Cam. Calibr. $\rightsquigarrow$ undistorted Image.
- Distortion correction:



(x, y) a point in distorted image

(x_{corr}, y_{corr}) same point in corrected image

r: distance of the corrected point to center of distortion, which is often center of the image. (x_c, y_c) is also called distortion center.

k_3: is required to correct major distortions and is often near zero for normal cameras. We take k_3 also into account.

Radial Distortion correction:

$$x_{distorted} = x_{corr}(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

Tangial Distortion Correction

$$x_{corr} = x + [2p_1xy + p_2(r^2 + 2x^2)]$$
$$y_{corr} = y + [2p_2xy + p_1(r^2 + 2y^2)]$$

- We need a known, regular, high contrast pattern for calibration (correcting for the distortion effect)

  => Chessboard

- cv2.findChessboardCorners() (img, pattern_size, flag=None)
- cv2.drawChessboardCorners (img, pattern_size, corners, ret)
- It is recommended to use at least 20 images to get reliable calibration.

[x, y, z]

objectPoints # known 3D coordinates of the chessboard

[0,0,0] to [6,4,0] for a 7x5 (8 cols, 6 rows) chessboard

the z coordinate of all object points will be zero, because they all lie on the same flat surface.

objp = np.zeros((7*5, 3), np.float32) → 35 x 3

objp[:, :2] = np.mgrid[0:7, 0:5].T.reshape(-1,2)

imPoints = # corners ⟷ 35 x 2

- glob ⇝ read images with a given filename pattern

- cv2.calibrateCamera(objPoints, imPoints, gray.shape[::-1], None, None)

  size of the image ← gray.shape[::-1]

  returns ret, mtx, dist, rvecs, tvecs

  mtx: (intrinsic) camera matrix
  dist: distortion coeffs
  rvecs: rotation vectors
  tvecs: translation vectors
  extrinsic

- cv2.undistort(img, mtx, dist_coeffs, None, mtx)

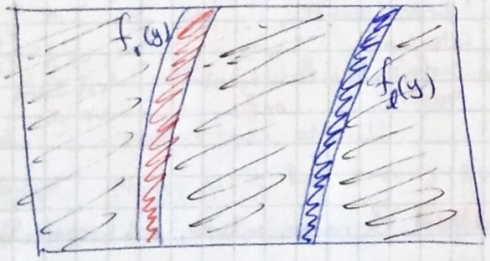
  ↳ destination image (undistorted)

Lane Curvature

- We can calcate the steering angle with information about speed and dynamics of the car and how much the lane is curving.

Steps:

1. detet the lane lines using masking and thresholding
2. perform a perspective transform to get a birds eye view of the lane
   Rectify the image
3. Fit a (2D?) polynomial to the lanes in Birds eye view.



r = right
l = left

$$f_r(y) = A_r y^2 + B_r y + C_r$$

$$f_l(y) = A_l y^2 + B_l y + C_l$$

Perspective:

- We can characterize perspective using 3D coordinates with objects with larger Z (depth distance) with respect to camera, appear smaller in the 2D image
- A Perspective Transform essentially changes the apparent z coordinate of the object points, to change the 2D representation of the object.
- A perspective transform warps the image and effectively drags points towards the camera or pushes them away.
- A perspective transform lets us change our perspective to view a scene from different view points or angles:
  - side of the camera
  - below
  - above
- We do perspective transform using correspondence points from two images

  → linear transformation

  from one image to the other.

- select 4 points that define a rectangle on the original image (input image, source image)

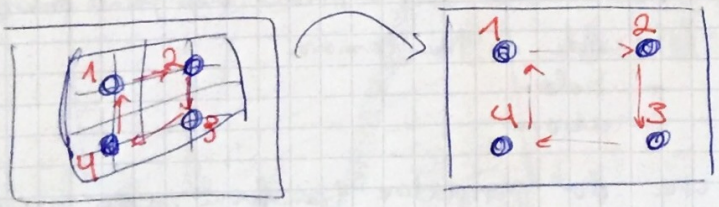
  → 4 points are enough to define a linear transform

M =
- cv2.getPerspectiveTransform(src, dest)

interpolation method for missing points

- cv2.warpPerspective(img, M, img_size, cv2.INTER_LINEAR)

- Select 4 points that make a Rectangle in SRC image
- Select/define their corresponding 4 points in dest imag



- color and Gradient thresholds, a way for determining where certain objects occur in images that have already been undistorted and perspective transformed.

L7: Gradients and Color Spaces

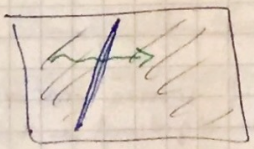
- Canny gives us a lot of edges on scenary and surrounding cars that we end up discarding.
- The lane lines are close to vertical. This we know a priori.

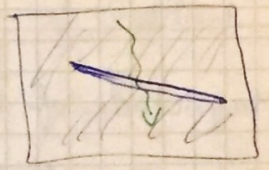
  => Take advantage of this fact by using gradients smarter to detect steep edges.
- Canny takes derivative w.r.t. x and y
- Sobel operator

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

gradient in x direction

gradient in y direction





- gradient in x direction emphasizes more the vertical lines.

  gradient in y direction emphasizes more the horizontal lines

```
gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0)
sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1)
```

- Absolute value of Sobel x

```
abs_sobelx = np.absolute(sobelx)
```

- Convert the abs value image to 8 bit

```
scaled_sobel = np.uint8(255 * abs_sobelx / np.max(abs_sobelx))
```

- Create a binary threshold to select pixels based on gradient strength:

```
thresh_min = 20
thresh_max = 100
sxbinary = np.zeros_like(scaled_sobel)
sxbinary[(scaled_sobel >= thresh_min) & (scaled_sobel <= thresh_max)] = 1

plt.imshow(sxbinary, cmap='gray')
```

- Magnitude of Gradient: square root of sum of gradient in both directions for

$$abs\_sobelx = \sqrt{(sobelx)^2}$$

$$abs\_sobely = \sqrt{(sobely)^2}$$

$$abs\_sobelxy = \sqrt{(sobelx)^2 + (sobely)^2}$$

- Direction of Gradient

- $\arctan\left(\frac{sobely}{sobelx}\right)$

Each pixel in gradient image has a value for gradient direction between $-\frac{\pi}{2}$, $\frac{\pi}{2}$

Orientation of 0 $\Longrightarrow$ vertical line

Orientation of $\pm\frac{\pi}{2}$ $\Longrightarrow$ horizontal line

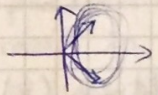
- np.arctan2 ⇝ returns values between $-\pi, +\pi$

but we take abs value of sobelx

$\Rightarrow x \geq 0 \Rightarrow$ arctan2 ⇝ returns between $\pm\frac{\pi}{2}$

- difference atan() and atan2() !

only quadrant I and IV



all quadrants

- the result of direction of Gradient is very noisy though

Color Thresholding:

- RGB thresholding does not work well in variable light conditions or when lane lines have different colors (like white & yellow)
- Color Space
  - RGB

White = (255, 255, 255)
  - HSV Hue, Saturation, Value

  - HLS

    L = lightness

    $0 \leq H \leq 179$

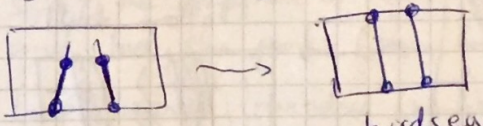
- Advantage of HLS
  - isolates the Lightness (L) component which varies the most under different lighting conditions
  - Hue and Saturation channel stay fairly consistent in shadow or excessive brightness.
- Combining color and gradient thresholding with S channel.

L8: Advanced Computer Vision

01.04.20
21:00

- We have covered so far:
  - Cam. calibration
  - distortion correction
  - Color and gradient threshold
  - Perspective Transform
- Then we will do
  - Detect lane lines
  - Determine the lane curvature

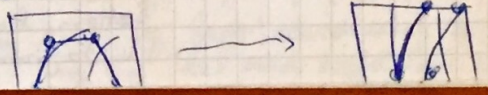
- We first do camera calibration (only once)
- Every frame gets distortion corrected using dist-coeffs.
- We then apply thresholding to get the lane lines clearly visible
- We then do perspective transform. We assume that the road is flat. We choose four points lying on 2 straight lane lines.



birdseye view

We then use this transform for every frame

- To test if the transform is correct, the lane lines should appear parallel, whether or not they are curved.

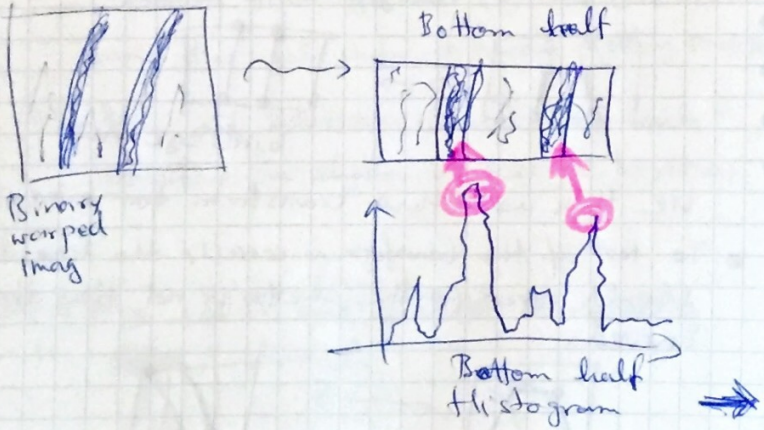


Finding the Lanes: Histogram Peaks

- input: Thresholded warped image
- there are many ways to do finding the lanes
- one method is Peaks in Histograms
- We need to determine: [input: a binary warped image, where the lane lines stand out clearly]
  - which pixels are part of the lines
  - which belong to the left line and which belong to the right line

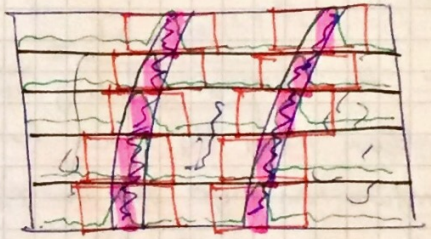
=> possible solution: Plotting the histogram of binary activations in image

take only the bottom half, Lane lines are likely to be most vertical nearest to the car.



Sliding Window

- The two peaks in the histogram are good indicators for the x-position of the base of the lane lines
- We use that starting position as the starting point of our search, for lines.
- We use a sliding window placed around the line centers to follo the lines up to the top of the frame.



Sliding Window

Histogram in each Sw gives us the position of that segment of lane.

Steps:

- split the histogram of base into two (left, right) and find the peaks (bases for left & right)
- Loop through nwindow (number of windows)
  - find the boundaries of window (we don't take the whole width, but a small window, and count all the pixels inside it; If number of pixels is larger than a minpix threshold, we move the window to the mean position of these pixels)
  - Add the (x,y) coordinates of nonzero pixels in the left and right lane lists

- Fit the polynomial, np.polyfit(x, y, deg)
- After tracking (finding) the lanes in one frame using sliding window, we don't need to do the sliding window in every consequitive frame.
- Instead we create a search margin around the found lanes and only search inside that margin.
- Of cource the current sliding window algorithm has a problem when the lanes have sharp curvature and go off the right or left edge of the image. This is related to the "minpix" threshold to move the position of the next sliding window.

Masuring Curvature

- What we have now is a two second order polynomials of lane lines.
- $f(y) = Ay^2 + By + C$ →

- We fit for y instead of x, f(x), because (f(y)) the lane lines are nearly vertical, and if we fit for f(x) chances are that there is same x value for more than one y value.

K : Curvature R : radius of curvature $= \frac{1}{K}$

- At point x of the function f(x) the radius of the curvature is given by

$$R_{curve} = \frac{\left[1 + \left(\frac{dx}{dy}\right)^2\right]^{3/2}}{\left|\frac{d^2x}{dy^2}\right|}$$

- We have a sec. order polynomial ($f(y) = Ax^2 + Bx + C$) =>

$$f'(y) = \frac{dy}{dx} = 2Ax + B$$

$$f''(y) = \frac{d^2y}{dx^2} = 2A$$

$$\Rightarrow R_{curve} = \frac{\left(1 + (2Ay + B)^2\right)^{3/2}}{|2A|}$$

- For the curvature of the lane at the nearest point to the car, put y = imag.shape[0] in the formula.

- The radius we calculated is in pixel space. We need to convert it to real-world space.
  - For this we require to know how long and wide is the section of the road that we are projecting in our warped image
    - we assume the sections for the images in this course that we are projecting is 30 m long and 3.7 meters wide
    - American U.S. road standard
      - minimum lane width = 12 ft = 3.7 m
      - dashed lane lines = 3 m long (10 ft)
- Let's say camera image is 720 pixel in y-dimension and we have roughly 700 relevant pixels in x-dimension.

$$\Rightarrow \quad y_{m\text{-}per\text{-}pix} = 30/720 \quad \# \text{ m/pix in y dimens.}$$

$$x_{m\text{-}per\text{-}pix} = 3.7/700 \quad \# \text{ m/pix in x dim.}$$

~~P2 Addon~~

02.04.20 10 p.m. – 03.04.20

P2: Advanced Lane Finding

- More realistic highway scenario
  - lighting changes
  - color changes
  - under passes
- If you do well in this scenario (only using cameras) you are reaching state of the art in CV and SDCs — Sebastian Thrun
- Expected magnitud of curve ≃ 1 km
- Assumpsion: camera is mounted at the center of the car
  - ⇒ mid point at the bottom of the image is between two lines (lane center)
    - check offset of the lane center from center of the image (converted from pixels to meters) is your distance from center of the lane.
- Define a Line() class to keep track of detected line information
- Sanity check. Detected lanes
  - have similar curvature
  - are separated by approx. right distance
  - are roughly parallel.
- Look Ahead Filter: search from prior in consequent frames. (use)
  - then do sanity check
  - if you lost track for several consequitive frames, start searching from scratch again

- Drawing (and unwarping) the lane lines:

```
warp_zero = np.zeros_like(warped).astype(np.uint8)
color_warp = np.dstack((warp_zero, warp_zero, warp_zero))

# Recast x and y into usable format for cv2.fillPoly()
# The point must be in clockwise or counter clockwise (np.flipud())
pts_left = np.array([np.transpose(np.vstack([left_fitx, ploty]))])
pts_right = np.array([np.flipud(np.transpose(np.vstack([right_fitx, ploty])))])
pts = np.hstack((pts_left, pts_right))

cv2.fillPoly(color_warp, np.int_([pts]), (0,255,0))
unwarp = cv2.warpPerspective(color_warp, M_inv, (img.shape[1], img.shape[0]))
result = cv2.addWeighted(undist, 1, unwarp, 0.3, 0)
plt.imshow(result)
```

Steps

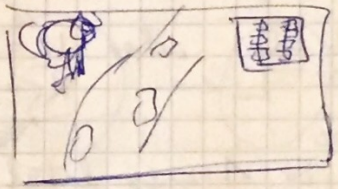
- ✓ Camera Calibration
- ✓ Distortion Correction
- ✓ Gradient, color, etc. thresholding
- ✓ perspective tranform to rectify image
- ✓ Detect lane pixels and fit a polynomial to find lane boundries
- ✓ Determine Lane curvature and vehicle offset from center
- ✓ Warp detected lane boundaries back onto the original image
- ✓ output lane boundaries and estimate of lane curvature and vehicle offset.

- Todo:
  - search ahead
  - line class
  - check for too sharp curves
  - sanity check
  - smoothing

- adaptive thresholding

:o) • Use Jupyter's interactive widgets for parameter tuning :)

- ↗ ipywidgets
- Add binary mini map to the



video to visualize your fitted lines and sliding window

L10: Neural Networks

7. April 22:00 Uhr

Luis Serano, Mat Leonard

- Neural Networks, Tensorflow, DNNs, CNNs
- Linear regression ⟶ fits a line
- Linear regression is good for estimating values on a continous spectrum
- Logistic Regression ⟶ predicting descrete data
- linear reg. ⟶ logistic reg. ⟶ NNs.
- linear decision boundary (a line)

$$W_1 x_1 + W_2 x_2 + b = 0$$

or

$$W\mathbf{X} + b = 0 \quad \text{wher} \quad W = (W_1, W_2), \quad \mathbf{X} = (x_1, x_2)$$

(W: weights, b: bias)

$y = 0$ or 1 ⟶ label

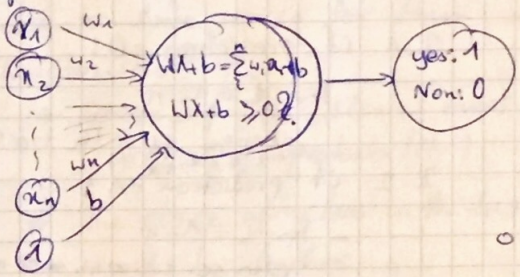
$$\hat{y} = \begin{cases} 1 & \text{if } W\mathbf{X} + b \geq 0 \\ 0 & \text{if } W\mathbf{X} + b < 0 \end{cases}$$ prediction

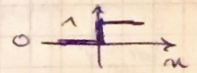
- 3D data ⟹ decision boundary would be a plane

$$W_1 x_1 + W_2 x_2 + W_3 x_3 + b = 0 \rightsquigarrow W\mathbf{X} + b$$

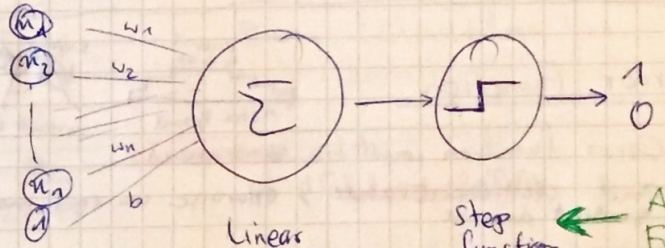
$W = (W_1, W_2, W_3)$
$\mathbf{X} = (x_1, x_2, x_3)$

- N-dimensional data: $x_1, \dots, x_n$
  - ↳ Bondary: $n-1$ dimensional hyperplane
- Perceptron:





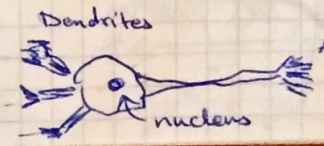
- Step function: $y = f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$



- Perceptron formula

$$f(x_1, \dots, x_n) = \begin{cases} 0 & \text{if } \sum W_i x_i + b < 0 \\ 1 & \text{if } \sum W_i x_i + b \geq 0 \end{cases}$$

- Learning rate

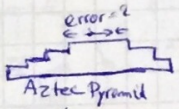


Perceptron Algorithm

1. Start with random weights: $w_1, \dots, w_n, b$
2. For every misclassified point $(x_1, \dots, x_n)$
   - 2.1 if prediction = 0:
     - for $i = 1..n$
       - change $w_i = w_i + \alpha x_i$
     - chang $b = b + \alpha$
   - 2.2 if prediction = 1:
     - for $i = 1..n$
       change $w_i = w_i - \alpha x_i$
     - chage $b = b - \alpha$

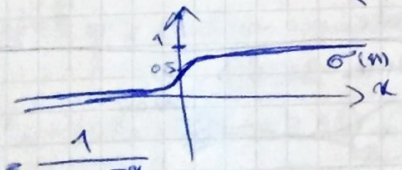
repeat till error small enough or for e.g. 1000 times

Error Function



- Error function must be **continous** and **differentiable**, otherwise we cant use gradient descent
- to have a continous error function we need our predictions to be continous and not descrete

- **Sigmoid function** ⇝ probabilities $0 \le p \le 1$



$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

Multiclass classification

↳ **softmax function**

- When we have more than 2 classes, we need to have scores for them with following charcteristics:
  - Sum of the scores must be 1 (because of probability)
  - The score of class with higher probability must be higher
- let's devide the score by sum of all

Duck	Biver	Walrus
2	1	0
$\frac{2}{2+1+0}$	$\frac{1}{2+1+0}$	$\frac{0}{2+1+0}$

- good idea, but it has a problem:
  what if some scores are negative and we end up with sum being zero?
- Let's take $\exp(x)$ ⇝ it is positive for every number

$$\exp(x) > 0$$

$$\frac{e^2}{e^2+e^1+e^0} \qquad \frac{e^1}{e^2+e^1+e^0} \qquad \frac{e^0}{e^2+e^1+e^0}$$

- **Softmax function:**

  Let's say we have n-classes and a linear ~~function~~ model that gives us the following scores $z_1, \cdots, z_n$

$$p(\text{class } i) = \frac{e^{z_i}}{e^{z_1} + e^{z_2} + e^{z_3} + \cdots + e^{z_n}}$$

- Soft max function for $n = 2$ is the sigmoid function.

```
def softmax(L):      # L a list of numbers
    expL = np.exp(L)
    return np.divide(expL, expL.sum())
```

- one-hot encoding
- Maximum likelihood
  - we pick the model that gives the existing Labels the highest probability.

  ⇒ By maximizing the probability, we can pick the best possible model.

- In order to tell how good or bad a model is, we calculate how likely it is. (its likelihood):
  - we calculate the probability of each point being the color it is according to the model.
  - multiply all these probabilities to obtain the probability, (the likelihood) of whole arrangement

- to find the best model, we need to maximize the $p(\text{all})$ probability. $p(\text{all}) = p_1 \cdot p_2 \cdot p_3 \cdot \cdots \cdot p_n$ for n data points.
- But products are not good! Let's take its log to turn product to sum

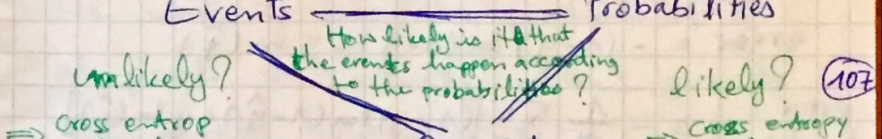
$$\Rightarrow p(\text{all}) = p_1 \cdot p_2 \cdot p_3 \cdot \cdots \cdot p_n$$

$$\Rightarrow \log(P(\text{all})) = \log(p_1 \cdot p_2 \cdot \cdots \cdot p_n) = \sum_i \log(p_i)$$

- we multiply the logs with -1, because $\log(x)$ is negative for $0 < x < 1$ (probabilities)
- This is called Cross Entropy

  Sum of negatives of all probabilities of a model $-\sum_i \log(p_i)$

- A good model has a smaller cross entropy!
- $-\log(p_i)$ for a point i is also a measure of error: the points that are mis classified have a larger value for $-\log(p_i)$
- now our goal is to minimize the cross entropy which is equivalent to maximizing the likelihood of the model



- **Binary Cross Entropy Loss:**

$$CE = -\sum_{i=1}^{m} y_i \ln(p_i) + (1-y_i)\ln(1-p_i)$$

$y_i = 1$ if ~~Class A~~
0 if not ~~Class A~~

- **Multi-Class Cross Entropy**
  - $$CE = -\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij} \ln(p_{ij})$$

$y_{ij} = 0$ or 1

n = number of classes

- **Logistic Regression.**
  - one of the most popular and useful algorithms in ML
  - Building block of all that constitutes DL
  - Logistic Regression:
    - Take your data
    - Pick a random model
    - Calculate the error
    - minimize the error, and obtain a better model
    - repeat

$$\text{error} = -(1-y)\ln(1-\hat{y}) - y\ln(\hat{y})$$

$$\text{error function} = -\frac{1}{m}\sum_{i=1}^{m}\left[(1-y_i)\ln(1-\hat{y}_i) + y_i\ln(\hat{y}_i)\right]$$

- **Multi-Class Entropy Error:** $-\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n} y_{ij}\ln(\hat{y}_{ij})$

- **Gradient Descent**

W = random()
b = random
$\hat{y} = \sigma(WX+b)$
$\nabla E = \left(\frac{dE}{dW}, \frac{dE}{db}\right)$
$\alpha = 0.1$ (learning rate)
$W' = W - \alpha\frac{dE}{dW}$
$b' = b - \alpha\frac{dE}{db}$

- **Derivative of sigmoid** $\sigma'(x) = \sigma(x)(1-\sigma(x))$

$$(e^x)' = e^x$$
$$\left(\frac{1}{u}\right)' = -\frac{u'}{u^2}$$

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right) = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}}\cdot\frac{e^{-x}}{1+e^{-x}}$$
$$= \sigma(x)\cdot(1-\sigma(x))$$

- We are after $\frac{dE}{dW}$ and $\frac{dE}{db}$
- for m data points $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$ the error is $E = -\frac{1}{m}\sum_{i=1}^{m}\left(y_i\ln(\hat{y}_i) + (1-y_i)\ln(1-\hat{y}_i)\right)$

We want to calculate gradient of E at point $x = (x_1, x_2, \ldots, x_n)$ given by the partial derivative: →

← $\nabla E \left(\frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_n}, \frac{\partial E}{\partial b} \right)$

- Let's actually think of the error that each point generates and calculate the derivative of this error. The total error is then the average of errors at all points. The error produced by each point is

$$E = -y \ln(\hat{y}) - (1-y) \ln(1-\hat{y})$$

- In order to calculate the derivative of this error w.r.t. the weights, let's first calculate $\frac{\partial \hat{y}}{\partial w_j}$

$$\hat{y} = \sigma(Wx+b)$$

$$\frac{\partial}{\partial w_j}(\hat{y}) = \frac{\partial}{\partial w_j} \sigma(Wx+b)$$

$$= \sigma(Wx+b)(1-\sigma(Wx+b)) \cdot \frac{\partial}{\partial w_j}(Wx+b)$$

$$= \hat{y}(1-\hat{y}) \cdot \frac{\partial}{\partial w_j}(Wx+b)$$

$$= \hat{y}(1-\hat{y}) \cdot \frac{\partial}{\partial w_j}(w_1 x_1 + \dots + w_j x_j + \dots + w_n x_n + b)$$

$$= \hat{y}(1-\hat{y}) \cdot x_j$$

because the only term in the sum which is not constant w.r.t. w_j is $w_j x_j$, which has derivative x_j

- Now we can go ahead and calculate the derivative of the error E at point x w.r.t. w_j

→

←

$$\frac{dE}{dw_j} = \frac{\partial}{\partial w_j} [-y \log(\hat{y}) - (1-y) \log(1-\hat{y})]$$

$$= -y \frac{\partial}{\partial w_j} \log(\hat{y}) - (1-y) \frac{\partial}{\partial w_j} \log(1-\hat{y})$$

$$= -y \cdot \frac{1}{\hat{y}} \frac{\partial}{\partial w_j} \hat{y} - (1-y) \cdot \frac{1}{(1-\hat{y})} \cdot \frac{\partial}{\partial w_j}(1-\hat{y})$$

$$= y \cdot \frac{1}{\hat{y}} \hat{y}(1-\hat{y}) x_j - (1-y) \cdot \frac{1}{1-\hat{y}} \cdot (-1) \hat{y}(1-\hat{y}) x_j$$

$$= -y(1-\hat{y}) \cdot x_j + (1-y) \hat{y} x_j$$

$$= -(y-\hat{y}) x_j$$

Similarly

$$\frac{\partial E}{\partial b} = -(y-\hat{y})$$

- This has a very important implication:
For a point with coordinates $(x_1, \dots, x_n)$, label y, and prediction $\hat{y}$ the gradient of the error function at that point is $(-(y-\hat{y})x_1, \dots, -(y-\hat{y})x_n, -(y-\hat{y}))$

$$\Rightarrow \nabla E = -(y-\hat{y})(x_1, \dots, x_n, 1)$$

- The gradient is actually a scalar times the coordinates of the point. And this scalar is nothing more than a multiple of the difference between the label and the prediction.

$\Rightarrow$ The closer the prediction to label, the smaller the gradient

or

The farther the Label from prediction, the larger the gradient.

- So exactly like the perceptron algorithm, small gradient means change the coordinates a little bit, and larger gradient means change the coordinates a lot.

- **Gradient Descent Step**

$$w_i' \leftarrow w_i - \alpha[-(y-\hat{y})x_i]$$

$$\Rightarrow w_i' \leftarrow w_i + \alpha(y-\hat{y})x_i$$

and $b' \leftarrow b + \alpha(y-\hat{y})$

$\Delta w_i = \alpha \delta x_i$

δ = error term

Grad. Descent vs. Perceptron Alg.

Same

if x is misclassified

Change

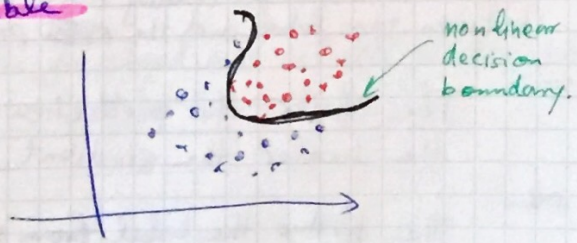
w_i to $w_i + \alpha(y-\hat{y})x_i$

change w_i to $\begin{cases} w_i + \alpha x_i & \text{label positive} \\ w_i - \alpha x_i & \text{label negative} \end{cases}$

- Neural Networks can show their full potential with non linear data

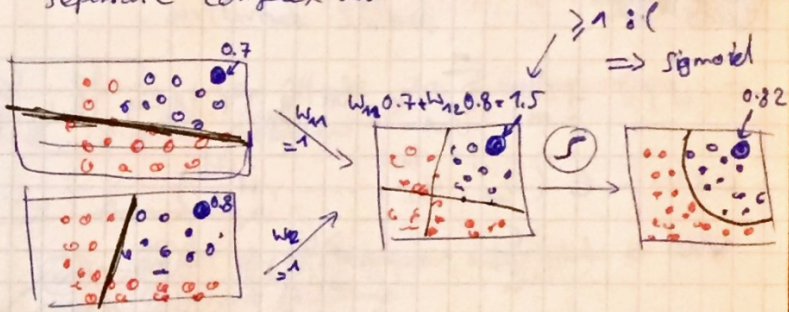
- Linear data : when a line can separate the data

- Non-linearly separable data



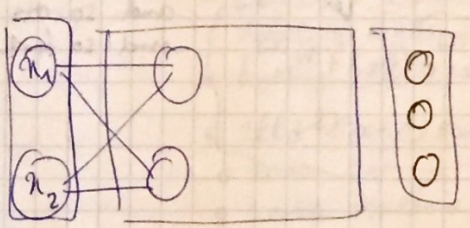
- Lovely explanation :)

We want to build (linear) combinations of several models to achieve new ones that separate complex data



w_{11}, w_{12}

- we can add weights to each model too to make its contribution more important.



One or more output nodes

Multiple output nodes for multi class classification.

input layer Hidden layer output layer ← where linear models get combined to obtain a non linear model

- **Multiple output**
  - softmax to get probability score of each class.

- Feed forward

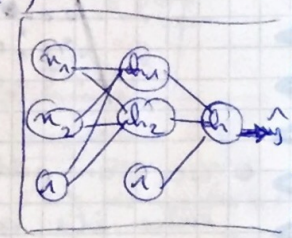
- Backprop
- Chain rule: derivative of composition can be shown as multiplying the partial derivatives.

$$E(W) = -\frac{1}{m}\sum_{i=1}^{m} y_i \ln(\hat{y}_i) + (1-y_i)\ln(1-\hat{y}_i)$$

$$E(W) = E\left(W_{11}^{(1)}, W_{12}^{(1)}, \dots, W_{31}^{(2)}\right)$$

$$\nabla E = \left(\frac{\partial E}{\partial W_{11}^{(1)}}, \dots, \frac{\partial E}{\partial W_{31}^{(2)}}\right)$$

$$\frac{\partial E}{\partial W_{11}^{(1)}} = \frac{\partial E}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial h}\frac{\partial h}{\partial h_1}\frac{\partial h_1}{\partial W_{11}^{(1)}}$$



$$\frac{\partial E}{\partial \hat{y}} = (\hat{y} - y)$$

$$h = W_{11}^{(2)}\sigma(h_1) + W_{21}^{(2)}\sigma(h_2) + W_{31}^{(2)}$$

$$\Rightarrow \frac{\partial h}{\partial h_1} = W_{11}^{(2)}\sigma(h_1)(1-\sigma(h_1))$$

and so on and so forth

LM: TensorFlow

09. April 22:00
2h

- Vincent Vanhoucke: principle scientist at Google's DL and director of robotics.
- DL shines when there are a lot of data and complex problems
- DL Advantages:
  - a lot of techniques, adaptable to many different problems
  - all using common infrastructure
  - and a common language to describe things

- History →
  - LeCun's LeNet end of 90's detecting digits handwriting
  - beginning 2000's: Alex Net
  - 2009 speech recognition
  - 2012 CV
  - 2014 Machine Translation

what changed? [at end of 2000's NNs made a comeback]
- a lot of data
- fast GPUs

- We will use TF to classify images of notMNIST dataset: a dataset of images of English letters A-J

```
import tensorflow as tf
hello_const = tf.constant("hello")
with tf.Session() as sess:
    output = sess.run(hello_const)
    print(output)
```

- TF data is not stored as integers, floats, or strings. The values are encapsulated in an object called **Tensors**
- hello_const ⟶ 0-dimensional string tensor. ⟶ a constant tensor
- A = tf.constant(1234) ⟶ 0-dim int32 tensor
  B = tf.constant([123,456,789]) ⟶ 1D int32 tensor
  C = tf.constant([[123,456,789], [2,8,4]]) ⟶ 2D " "
- ~~Session~~
- **Computational Graph**
- tf.Session() an environment for running computational graphs
- tf.placeholder(), feed_dict

```
x = tf.placeholder(tf.string)
with tf.Session() as sess:
    output = sess.run(x, feed_dict={x: 'hello'})
```

- tf.add(), tf.subtract, tf.multiply()
- tf.cast() ⟶ no implicit casting!

- Classification ⟶ core building block tasks in many ML tasks
  - ↙ reinforcement Learning
  - ↓ detection
  - ↘ ranking
  - → regression
- **Logistic Classifier (Linear Classifier)**

$$WX + b = \hat{y}$$

- Softmax function: $S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$

  y_i, y_j : scores

  Softmax function takes the **scores** (a.k.a. **logits**) and turns them to **probabilities**.

- Tensorflow uses $XW + b$ instead of $WX + b$
- tf.Variable(), x = tf.Variable(5)
- ~~tf.global_variables_initializer()~~
- tf.global_variables_initializer()
- tf.truncated_normal() ⟶ generate random numbers from Normal dist
  (n_features, n_labels)
- tf.zeros()
- tf.matmul()
- tf.nn.softmax( )