CS 131 Lecture 1: Course introduction

Olivier Moindrot
Department of Computer Science
Stanford University
Stanford, CA 94305
olivierm@stanford.edu

1 What is computer vision?

1.1 Definition

Two definitions of computer vision Computer vision can be defined as *a scientific field that extracts information out of digital images*. The type of information gained from an image can vary from identification, space measurements for navigation, or augmented reality applications.

Another way to define computer vision is through its applications. Computer vision is *building algorithms that can understand the content of images and use it for other applications*. We will see in more details in section 4 the different domains where computer vision is applied.

A bit of history The origins of computer vision go back to an MIT undergraduate summer project in 1966 [4]. It was believed at the time that computer vision could be solved in one summer, but we now have a 50-year old scientific field which is still far from being solved.

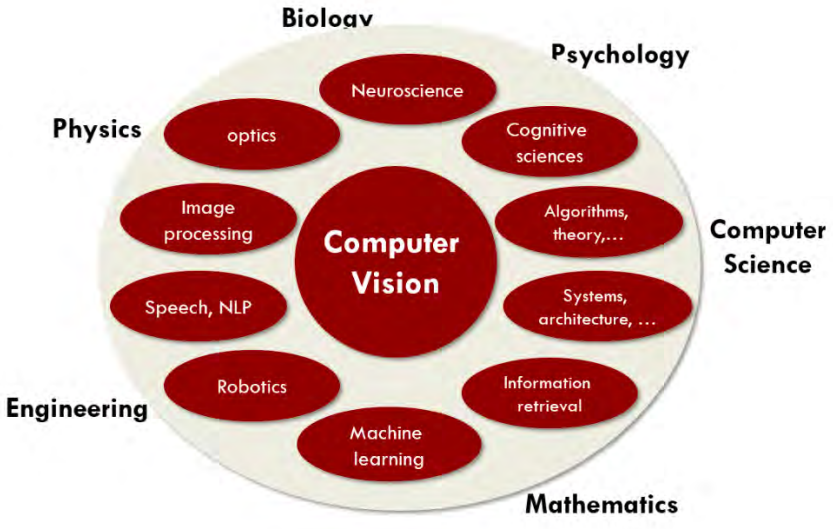


Figure 1: Computer vision at the intersection of multiple scientific fields

1.2 An interdisciplinary field

Computer vision brings together a large set of disciplines. Neuroscience can help computer vision by first understanding human vision, as we will see in section 2. Computer vision can be seen as a part of computer science, and algorithm theory or machine learning are essential for developing computer vision algorithms.

We will show in this class how all the fields in figure 1 are connected, and how computer vision draws inspiration and techniques from them.

1.3 A hard problem

Computer vision has not been solved in 50 years, and is still a very hard problem. It's something that we humans do unconsciously but that is genuinely hard for computers.

Poetry harder than chess The IBM supercomputer Deep Blue defeated for the first time the world chess champion Garry Kasparov in 1997. Today we still struggle to create algorithms that output well formed sentences, let alone poems. The gap between these two domains show that what humans call *intelligence* is often not a good criteria to assess the difficulty of a computer task. Deep Blue won through brute force search among millions of possibilities and was not more *intelligent* than Kasparov.

Vision harder than 3D modeling It is today easier to create a 3D model of an object up to millimeter precision than to build an algorithm that recognizes chairs. Object recognition is still a very difficult problem, although we are approaching human accuracy.

Why is it so hard? Computer vision is hard because there is a huge gap between pixels and meaning. What the computer sees in a 200×200 RGB image is a set of $120,000$ values. The road from these numbers to meaningful information is very difficult. Arguably, the human brain's visual cortex solves a problem as difficult: understanding images that are projected on our retina and converted to neuron signals. The next section will show how studying the brain can help computer vision.

2 Understanding human vision

A first idea to solve computer vision is to understand how human vision works, and transfer this knowledge to computers.

2.1 Definition of vision

Be it a computer or an animal, vision comes down to two components.

First, a **sensing device** captures as much details from an image as possible. The eye will capture light coming through the iris and project it to the retina, where specialized cells will transmit information to the brain through neurons. A camera captures images in a similar way and transmit pixels to the computer. In this part, cameras are better than humans as they can see infrared, see farther away or with more precision.

Second, the **interpreting device** has to process the information and extract meaning from it. The human brain solves this in multiple steps in different regions of the brain. Computer vision still lags behind human performance in this domain.

2.2 The human visual system

In 1962, Hubel & Wiesel [3] tried to understand the visual system of a cat by recording neurons while showing a cat bright lines. They found that some specialized neurons fired only when the line was in a particular spot on the retina or if it had a certain orientation. [1]

[1] More information in this blog post

Their research led to the beginning of a scientific journey to understand the human visual system, which is still active today.

They were awarded the Nobel Prize in Physiology and Medecine in 1981 for their work. After the announcement, Dr. Hubel said:

> There has been a myth that the brain cannot understand itself. It is compared to a man trying to lift himself by his own bootstraps. We feel that is nonsense. The brain can be studied just as the kidney can.

2.3 How good is the human visual system?

Speed The human visual system is very efficient. As recognizing threats and reacting to them quickly was paramount to survival, evolution perfected the visual system of mammals for millions of years.

The speed of the human visual system has been measured [7] to around 150ms to recognize an animal from a normal nature scene. Figure 2 shows how the brain responses to images of animals and non-animals diverge after around 150ms.

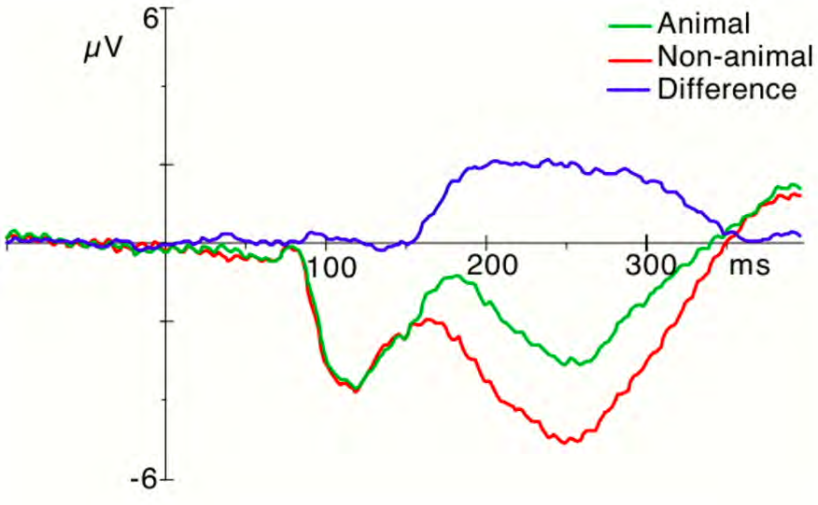


Figure 2: Difference between animal and non-animal response. From [7]

Fooling humans However, this speed is obtained at the price of some drawbacks. Changing small irrelevant parts of an image such as water reflection or background can go unnoticed because the human brain focuses on the important parts of an image [5].

If the signal is very close to the background, it can be difficult to detect and segment the relevant part of the image.

Context Humans use context all the time to infer clues about images. Previous knowledge is one of the most difficult tool to incorporate into computer vision. Humans use context to know where to focus on an image, to know what to expect at certain positions. Context also helps the brain to compensate for colors in shadows.

However, context can be used to fool the human brain.

2.4 Lessons from nature

Imitating birds did not lead humans to planes. Plainly copying nature is not the best way or the most efficient way to learn how to fly. But studying birds made us understand aerodynamics, and understanding concepts like lift allowed us to build planes.

The same might be true with intelligence. Even though it is not possible with today's technology, simulating a full human brain to create intelligence might still not be the best way to get there. However, neuroscientists hope to get insights at what may be the concepts behind vision, language and other forms of intelligence.

3 Extracting information from images

We can divide the information gained from images in computer vision in two categories: measurements and semantic information.

3.1 Vision as a measurement device

Robots navigating in an unknown location need to be able to scan their surroundings to compute the best path. Using computer vision, we can measure the space around a robot and create a map of its environment.

Stereo cameras give depth information, like our two eyes, through triangulation. Stereo vision is a big field of computer vision and there is a lot of research seeking to create a precise depth map given stereo images.

If we increase the number of viewpoints to cover all the sides of an object, we can create a 3D surface representing the object [2]. An even more challenging idea might be to reconstruct the 3D model of a monument through all the results of a google image search for this monument [1].

There is also research in grasping, where computer vision can help understand the 3D geometry of an object to help a robot grasp it. Through the camera of the robot, we could recognize and find the handle of the object and infer its shape, to then enable the robot to find a good grasping position [6].

3.2 A source of semantic information

On top of measurement informations, an image contains a very dense amount of *semantic information*. We can label objects in an image, label the whole scene, recognize people, recognize actions, gestures, faces.

Medical images also contain a lot of semantic information. Computer vision can be helpful for a diagnosis based on images of skin cells for instance, to decide if they are cancerous or not.

4 Applications of computer vision

Cameras are everywhere and the number of images uploaded on internet is growing exponentially. We have images on Instagram, videos on YouTube, feeds of security cameras, medical and scientific images... Computer vision is essential because we need to sort through these images and enable computers to understand their content. Here is a non exhaustive list of applications of computer vision.

Special effects Shape and motion capture are new techniques used in movies like Avatar to animate digital characters by recording the movements played by a human actor. In order to do that, we have to find the exact positions of markers on the actor's face in a 3D space, and then recreate them on the digital avatar.

3D urban modeling Taking pictures with a drone over a city can be used to render a 3D model of the city. Computer vision is used to combine all the photos into a single 3D model.

Scene recognition It is possible to recognize the location where a photo was taken. For instance, a photo of a landmark can be compared to billions of photos on google to find the best matches. We can then identify the best match and deduce the location of the photo.

Face detection Face detection has been used for multiple years in cameras to take better pictures and focus on the faces. Smile detection can allow a camera to take pictures automatically when

the subject is smiling. Face recognition is more difficult than face detection, but with the scale of today's data, companies like Facebook are able to get very good performance. Finally, we can also use computer vision for biometrics, using unique iris pattern recognition or fingerprints.

Optical Character Recognition One of the oldest successful applications of computer vision is to recognize characters and numbers. This can be used to read zipcodes, or license plates.

Mobile visual search With computer vision, we can do a search on Google using an image as the query.

Self-driving cars Autonomous driving is one of the hottest applications of computer vision. Companies like Tesla, Google or General Motors compete to be the first to build a fully autonomous car.

Automatic checkout Amazon Go is a new kind of store that has no checkout. With computer vision, algorithms detect exactly which products you take and they charge you as you walk out of the store [2].

Vision-based interaction Microsoft's Kinect captures movement in real time and allows players to interact directly with a game through moves.

Augmented Reality AR is also a very hot field right now, and multiple companies are competing to provide the best mobile AR platform. Apple released ARKit in June and has already impressive applications [3].

Virtual Reality VR is using similar computer vision techniques as AR. The algorithm needs to know the position of a user, and the positions of all the objects around. As the user moves around, everything needs to be updated in a realistic and smooth way.

References

[1] Michael Goesele, Noah Snavely, Brian Curless, Hugues Hoppe, and Steven M Seitz. Multi-view stereo for community photo collections. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1–8. IEEE, 2007.

[2] Anders Heyden and Marc Pollefeys. Multiple view geometry. *Emerging topics in computer vision*, pages 45–107, 2005.

[3] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154, 1962.

[4] Seymour A Papert. The summer vision project. 1966.

[5] Ronald A Rensink, J Kevin O'Regan, and James J Clark. On the failure to detect changes in scenes across brief interruptions. *Visual cognition*, 7(1-3):127–145, 2000.

[6] Ashutosh Saxena, Justin Driemeyer, and Andrew Y Ng. Robotic grasping of novel objects using vision. *The International Journal of Robotics Research*, 27(2):157–173, 2008.

[7] Simon Thorpe, Denise Fize, and Catherine Marlot. Speed of processing in the human visual system. *nature*, 381(6582):520, 1996.

[2] see their video here

[3] check out the different apps

Lecture #2: Color and Linear Algebra pt.1

John McNelly, Alexander Haigh, Madeline Saviano, Scott Kazmierowicz, Cameron Van de Graaf
Department of Computer Science
Stanford University
Stanford, CA 94305
{jmcnelly, haighal, msaviano, scottkaz, camvdg}@cs.stanford.edu

1 Physics of Color

1.1 What is color?

Color is the result of interaction between physical light in the environment and our visual system. A psychological property of our visual experiences when we look at objects and lights, not a physical property of those objects or lights. [**?**]

1.2 Color and light

White light is composed of almost equal energy in all wavelengths of the visible spectrum

1.3 Electromagnetic Spectrum

Light is made up of waves of different wavelengths. The visual spectrum of light ranges from 400nm to 700nm, and humans are most sensitive to light with wavelengths in the middle of this spectrum. Humans see only visible light because the Sun emits yellow light more than any other color and due to its temperature.

1.4 Visible light

Planck's Law for Blackbody radiation estimates the wavelengths of electromagnetic radiation emitted by a star, based on surface temperature. For instance, since the surface of the sun is around 5800K, the peak of the sun's emitted light lies in the visible region.

1.5 The Physics of Light

Any source of light can be completely described physically by its spectrum (i.e., the amount of energy emitted, per time unit, at each wavelength 400-700nm). Surfaces have reflectance spectra: reflected light is focused on a certain side of the visible light spectrum. For example, bananas reflect mostly yellow light, and tomatoes reflect mostly red light.

1.6 Interaction of light and surfaces

Reflected color is the result of interaction of the light source spectrum with the surface reflectance. As a rule, definitions and units are stated as "per unit wavelengths", and terms are assumed to be "spectral" (i.e., referring to a spectrum of light, not a single wavelength). Illumination is quantified as $Illumination. * Reflectance = ColorSignal$ [**?**]

2 Human Encoding of Color

As mentioned in the previous section, color is not a mere physical property of light - rather, the phenomenon of color arises via the interaction between light and the human visual system.

2.1 Rods and Cones

When we look at a scene, light first enters our eyes through the pupil before reaching the retina. The retina is primarily composed of two types of light-sensitive cells: rods and cones, named for their appearance under a microscope. **Rods** are the more numerous of the two and are highly sensitive, making them ideal for detecting objects in low-light conditions. However, they do not encode any color information. **Cones**, on the other hand, are less numerous and less sensitive, but they are useful for distinguishing between objects in high light conditions. They also allow us to perceive colors by a mechanism discussed below.

2.2 Cones and Color

A crucial difference between rods and cones is that the latter comes in three different types, each characterized by a unique response curve to different wavelengths of light. Each response curve peaks at a unique wavelength, namely 440, 530, and 560nm, aligning with the colors of blue, green, and red. However, both cones and rods act as **filters**, the output of which can be interpreted as the multiplication of each response curve by the spectrum, integrated over all wavelengths. While the information encoded by the resulting three numbers is sufficient for most tasks, some information is lost in the compression from spectrum to electrical impulse in the retina. This implies that some subset(s) of spectra will be erroneously perceived as identical - such spectra are called **metamers**

2.3 Color Matching

Since we are interested in designing systems that provide a consistent visual experience across viewers, it is helpful to understand the minimal colors that can be combined to create the experience of any perceivable color. An experiment from Wandell's Foundations of Vision (Sinauer Assoc., 1995) demonstrates that most people report being able to recreate the color of a given test light by tuning three experimental lights of differing colors. The only condition is that each of three lights must be a primary color. Moreover, the experiment showed that for the same test light and primaries, the majority of people select similar weights, though color blind people are an exception. Finally, this experiment validates the trichromatic theory of color - the proposition that three numbers are sufficient for encoding color - which dates from Thomas Young's writings in the 1700s.

3 Color Spaces

3.1 Definition

Color space, also known as the color model (or color system), is an abstract mathematical model which describes the range of colors as tuples of numbers, typically as 3 or 4 values or color components (e.g. RGB). A color space may be arbitrary or structured mathematically. Most color models map to an absolute and globally understood system of color interpretation.

3.2 Linear Color Spaces

Defined by a choice of three primaries, and the coordinates of the color are given by the weights of the primaries used to match it

- RGB Space
  - Primary colors are monochromatic lights (for monitors, they correspond to the three types of phosphors)
  - Subtractive matching is required for certain wavelengths of light
- CIE XYZ Color Space

Figure 1: Mixing two lights produces colors that lie along a straight line in color space. Mixing three lights produces colors that lie within the triangle they define in color space.

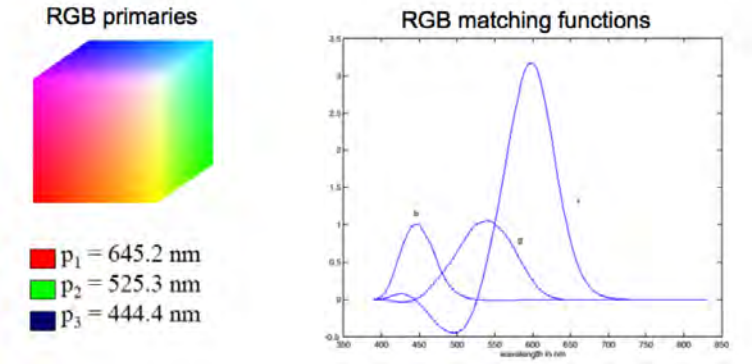


Figure 2: Representation of RBG primaries and corresponding matching functions. The matching functions are the amounts of primaries needed to match the monochromatic test color at the wavelength shown on the horizontal scale. Source: `https://en.wikipedia.org/wiki/CIE_1931_color_space`

- Primaries are imaginary, but matching functions are everywhere positive
- The Y parameter corresponds to brightness or luminance of a color
- Related to RGB space by linear transformation, upholding Grassmann's Law

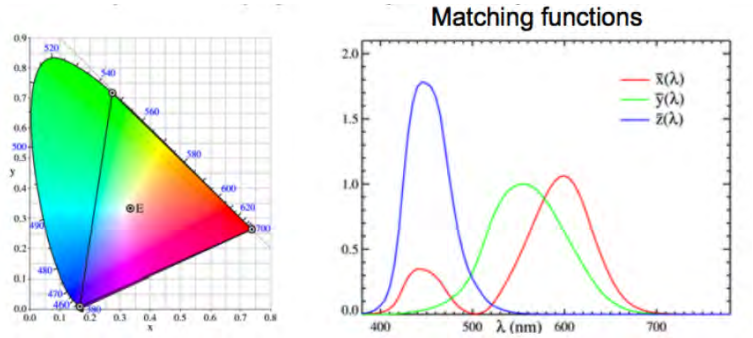


Figure 3: Source: `https://en.wikipedia.org/wiki/CIE_1931_color_space`

3.3 Nonlinear Color Spaces: HSV

- Designed to reflect more traditional and intuitive color mixing models (e.g. paint mixing)
- Based on how colors are organized and conceptualized in human vision
- Dimensions are: Hue, Saturation, and Value (Intensity)

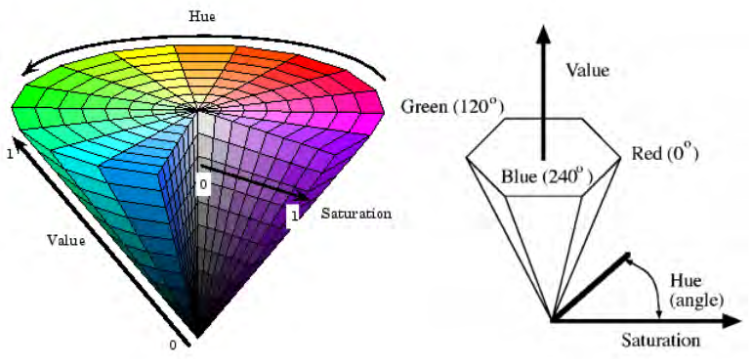


Figure 4: General source: https://en.wikipedia.org/wiki/HSL_and_HSV

4 White Balancing

4.1 Definition

White Balance is the processing of adjusting the image data received by sensors to properly render neutral colors (white, gray, etc). This adjustment is performed automatically by digital cameras (custom settings for different light), and film cameras offer several filters and film types for different shooting conditions.

4.2 Importance of White Balancing

Unadjusted images have an unnatural color "cast" for a few reasons, making white balancing very important:

1. The sensors in cameras or film are different from those in our eyes
2. Different display media render images differently, which must be accounted for
3. The viewing conditions when the image was taken are usually different from the image viewing conditions

Figure 5: Example of two photos, one unbalanced, and one with incorrect white balancing. Source: http://www.cambridgeincolour.com/tutorials/white-balance.htm

**Upload images from slides

4.3 Von Kries Method

Von Kries' method for white balancing was to scale each channel by a "gain factor" to match the appearance of a gray neutral object.

In practice, the best way to achieve this is the **Gray Card Method**: hold up a neutral (gray or white) and determine the values of each channel. If we find that the card has RGB values $r_w,\ g_w,\ b_w$, then we scale each channel of the image by $\frac{1}{r_w},\ \frac{1}{g_w},\ \frac{1}{b_w}$.

4.4 Other Methods

Without Gray Cards, we need to guess which pixels correspond to white objects. Several methods attempt to achieve this, including statistical and Machine Learning models (which are beyond the scope of this class)

Gray World Assumption Under this model, we assume that the average pixel value in the photo $(r_{ave},\ g_{ave},\ b_{ave})$ is gray and scale the image pixels by $\frac{1}{r_{ave}},\ \frac{1}{g_{ave}}$, and $\frac{1}{b_{ave}}$

Brightest Pixel Assumption this works on non-saturated images and assumes that image highlights usually have the color of the light source (which is usually white). So, it corrects white balance by weighting each channel inversely proportional to the values of the brightest pixels

Gamut Mapping The Gamut of an image is the set of all pixel colors displayed in an image (in mathematical terms, this is a "convex hull" and a subset of all possible color combinations). We can then apply a transformation to the image that maps the gamut of the image to the gamut of a "standard" image under white light.

4.5 Other Uses of Color in Computer Vision

Color plays a critical role in skin detection, nude detection, and image segmentation, among other applications.

5 Linear Algebra Primer: Vectors and Matrices

Definition Vectors and matrices are simply collections of ordered numbers that represent something: movements in space, scaling factors, pixel brightness, etc.

5.1 Vectors

A column vector $v \in \mathbb{R}^{nx1}$ where $v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$.

A row vector $v^T \in \mathbb{R}^{1xn}$ where $v^T = \begin{bmatrix} v_1 & v_2 & \dots & v_n \end{bmatrix}$. T denotes the transpose operation which flips a matrix over its diagonal, switching the row and column indices of the matrix.

As a note, CS 131 will use column vectors as the default. You can transpose vectors in python: for vector v, do $v.t$.

Uses of Vectors There are two main uses of vectors. Firstly, vectors can represent an offset in 2 or 3 dimensional space. In this case, a point is a vector from the origin. Secondly, vectors can represent data (such as pixels, gradients at an image keypoint, etc.). In this use case, the vectors do not have a geometric interpretation but calculations like "distance" can still have value.

5.2 Matrices

A matrix $\mathbf{A} \in \mathbb{R}^{mxn}$ is an array of numbers with m rows and n columns:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \dots & a_{mn} \end{bmatrix}$$

If $m = n$, we say **A** is square.

An image is represented in python as a matrix of pixel brightness. The upper left corner of the image is $[y, x]$. **Grayscale images** have one number per pixel and are stored as an mxn matrix. **Color images** have 3 numbers per pixel – red, green, and blue brightness (RGB) and are thus stored as a $mxnx3$ matrix. Multidimensional matrices like these are called **tensors**.

5.3 Basic Matrix Operations

Addition $\begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} a+1 & b+2 \\ c+3 & d+4 \end{bmatrix}$

You can only add a matrix with matching dimensions or a scalar.

$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + 7 = \begin{bmatrix} a+7 & b+7 \\ c+7 & d+7 \end{bmatrix}$

Scaling $\begin{bmatrix} a & b \\ c & d \end{bmatrix} * 3 = \begin{bmatrix} 3a & 3b \\ 3c & 3d \end{bmatrix}$

Norm $\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$

More formally, a norm is any function $f : \mathbf{R}^n \to \mathbf{R}$ that satisfies four properties:

1. **Non-negativity**: for all $x \in \mathbf{R}^n, f(x) \geq 0$
2. **Definiteness**: $f(x) = 0$ if and only if $x = 0$
3. **Homogeneity**: for all $x \in \mathbf{R}^n, t \in \mathbf{R}, f(tx) = |t|f(x)$
4. **Triangle Inequality**: for all $x, y \in \mathbf{R}^n, f(x + y) \leq f(x) + f(y)$

Example Norms:

One Norm $\|x\|_1 = \sum_{i=1}^{n} |x_i|$

Infinity Norm $\|x\|_{\text{inf}} = \max_i |x_i|$

General P Norm $\|x\|_p = \left(\sum_{i=1}^{n} x_i^p \right)^{1/p}$

Matrix Norm $\|A\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij}^2} = \sqrt{tr(A^T A)}$

Inner or Dot Product Multiply corresponding entries of two vectors and add up the result. $x \cdot y$ is $\|x\| \, \|y\| \cos(\text{the angle between x and y})$. Thus if y is a unit vector then $x \cdot y$ gives the length of x which lies in the direction of y.

One dimensional dot product: $x^T y = x \cdot y = \begin{bmatrix} x_1 & \dots & x_n \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \sum_{i=1}^{n} x_i y_i$ (scalar)

Multiplication The product of two matrices $A \in \mathbf{R}^{mxn}, B \in \mathbf{R}^{nxp}$ results in $C = AB \in \mathbf{R}^{mxp}$ where $C_{ij} = \sum_{k=1}^{n} A_{ik}B_{kj}$.

$$C = AB = \begin{bmatrix} — & a_1^T & — \\ — & a_2^T & — \\ & \vdots & \\ — & a_m^T & — \end{bmatrix} \begin{bmatrix} | & | & \cdots & | \\ b_1 & b_2 & \cdots & b_p \\ | & | & \cdots & | \end{bmatrix} = \begin{bmatrix} a_1^T b_1 & a_1^T b_2 & \cdots & a_1^T b_p \\ a_2^T b_1 & a_2^T b_2 & \cdots & a_2^T b_p \\ \vdots & \vdots & \ddots & \vdots \\ a_m^T b_1 & a_m^T b_2 & \cdots & a_m^T b_p \end{bmatrix}$$

Each entry of the matrix product is made by taking the dot product of the corresponding row in the left matrix, with the corresponding column in the right one.

Matrix multiplication is

1. **Associative**: $(AB)C = A(BC)$
2. **Distributive**: $(A + B)C = AC + BC$
3. **Not Commutative**: Generally $AB \neq BA$. For example, if $A \in \mathbb{R}^{mxn}, B \in \mathbb{R}^{nxq}$, the product BA does not even exist if m and q are not equal!

Powers By convention, we can refer to the matrix product AA as A^2 and AAA as A^3, etc. Obviously only square matrices can be multiplied that way.

Transpose Flip matrix so row 1 becomes column 1:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

A useful identity: $(ABC)^T = C^T B^T A^T$.

Determinant $det(\mathbf{A})$ returns a scalar. It represents the area (or volume) of the parallelogram described by the vectors in the rows of the matrix.

For $\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$, $det(\mathbf{A}) = ad - bc$

Properties:

- $det(\mathbf{AB}) = det(\mathbf{BA})$
- $det(\mathbf{A}^{-1}) = \frac{1}{det(\mathbf{A})}$
- $det(\mathbf{A}^T) = det(\mathbf{A})$
- $det(\mathbf{A}) = 0$ if and only if $\mathbf{A}$ is singular.

Trace $tr(\mathbf{A})$ is the sum of diagonal elements. For $\mathbf{A} = \begin{bmatrix} 1 & 3 \\ 5 & 7 \end{bmatrix}$, $tr(\mathbf{A}) = 1 + 7 = 8$

Properties:

- $tr(\mathbf{AB}) = tr(\mathbf{BA})$
- $tr(\mathbf{A} + \mathbf{B}) = tr(\mathbf{A}) + tr(\mathbf{B})$

Special Matrices

- Identity Matrix: a square matrix with 1's along the diagonal and 0's elsewhere. $\mathbf{I}$ * another matrix = that matrix. Example identity matrix: $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
- Diagonal Matrix: a square matrix with numbers along the diagonal and 0's elsewhere. A diagonal * another matrix scales the rows of that matrix. Example diagonal matrix: $\begin{bmatrix} 3 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 2.5 \end{bmatrix}$

- Symmetric Matrix: $A^T = A$. Example: $\begin{bmatrix} 1 & 2 & 5 \\ 2 & 1 & 7 \\ 5 & 7 & 1 \end{bmatrix}$
- Skew-symmetric matrix:$A^T = -A$. Example: $\begin{bmatrix} 0 & -2 & -5 \\ 2 & 0 & -7 \\ 5 & 7 & 0 \end{bmatrix}$

References

[1] Stephen E Palmer. *Vision science: Photons to phenomenology*. MIT press, 1999.

[2] Brian A Wandell. *Foundations of vision.* Sinauer Associates, 1995.

Linear Algebra Primer Part 2

Liangcheng Tao, Vivian Hoang-Dung Nguyen, Roma Dziembaj, Sona Allahverdiyeva
Department of Computer Science
Stanford University
Stanford, CA 94305
{lctao13, vnguyen2, romad, sonakhan}@cs.stanford.edu

1 Vectors and Matrices Recap

1.1 Vector

A column vector $\mathbf{v} \in \mathbb{R}^{\mathrm{nx1}}$ where

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

A row vector $\mathbf{v}^T \in \mathbb{R}^{\mathrm{1xn}}$ where

$$\mathbf{v}^T = \begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix}$$

T denotes the transpose operation.

The **norm** is

$$||x||_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$$

Formally, the norm can also be defined as any function $f : \mathbb{R}^{\mathrm{n}} \mapsto \mathbb{R}$ that satisfies 4 properties:

- **Non-negativity:** For all $x \in \mathbb{R}^{\mathrm{n}}$, $f(x) \geq 0$
- **Definiteness:** $f(x) = 0$ if and only if $x = 0$
- **Homogeneity:** For all $x \in \mathbb{R}^{\mathrm{n}}$, $t \in \mathbb{R}$, $f(tx) = |t|f(x)$
- **Triangle inequality:** For all $x, y \in \mathbb{R}^{\mathrm{n}}$, $f(x + y) \leq f(x) + f(y)$

1.1.1 Projection

A **projection** is an inner product (dot product) of vectors. If B is a unit vector, then $A \cdot B$ gives the length of A which lies in the direction of B.

1.2 Matrix

A matrix $\mathbf{A} \in \mathbb{R}^{\mathrm{mxn}}$ is an array of numbers with size m by n.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & & & & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix}$$

If $m = n$, we say that $\mathbf{A}$ is square.

1.2.1 An Application of Matrices

Grayscale images have one number per pixel and are stored as an $m \times n$ matrix. Color images have 3 numbers per pixel - red, green, and blue

2 Transformation Matrices

Matrices can be used to transform vectors in useful ways, through multiplication: $x' = Ax$. The simplest application of that is through scaling, or multiplying a scaling matrix with scalars on its diagonal by the vector.

We can also use matrices to rotate vectors. When we multiply a matrix and a vector; the resulting x coordinate is the original vector **dot** the first row.

In order to rotate a vector by an angle θ, counter-clockwise:

$x' = cos\theta x - sin\theta y$ and

$y' = cos\theta y + sin\theta x$

therefore, we multiply it by the matrix

$$M = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix}$$

which gives us that $P' = RP$.

We can also use multiple matrices to transform a point. For example, $p' = R_2R_1Sp$. The transformations are applied one after another, from right to left. In our example, this would be $(R_2(R_1(Sp)))$.

In order to translate vectors, we have to implement a somewhat hacky solution of adding a "1" at the very end of the vector. This way, using these "homogeneous coordinates", we can also translate vectors. The multiplication works out so the rightmost column of our matrix gets added to the respective coordinates. A homogenous matrix will have [0 0 1] in the bottom row to ensure that the elements get added correctly, and the resulting vector has '1' at the bottom.

By convention, in homogeneous coordinates, we divide the result by its last coordinate after doing matrix multiplication.

$$\begin{bmatrix} x \\ y \\ 7 \end{bmatrix}$$

$$\begin{bmatrix} x/7 \\ y/7 \\ 1 \end{bmatrix}$$

So to obtain the result of $P(x, y) -> P' = (s_x x, s_y y)$

we have to first $P = (x, y) -> (x, y, 1)$ and then $P' = (s_x x, s_y y) -> (s_x x, s_y y, 1)$ so we can then do the matrix multiplication S*P. Though, we have to note that scaling and translating is not the same as translating and scaling. In other words, $T * S * P \neq S * T * P$

Any rotation matrix R belongs to the category of normal matrices, and it satisfies interesting properties. For example, $R\dot{R}^T = I$ and $det(R) = 1$

The rows of a rotation matrix are always mutually perpendicular (a.k.a. orthogonal) unit vectors; this is what allows for it to satisfy some of the few unique properties mentioned above.

3 Matrix Inverse

Given a matrix A, its inverse A^{-1} is a matrix such that:

$$AA^{-1} = A^{-1}A = I$$

where I is the identity matrix of the same size.

An example of a matrix inverse is:

$$\begin{bmatrix} 2 & 0 \\ 0 & 3 \end{bmatrix}^{-1} = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{3} \end{bmatrix}$$

A matrix does not necessarily have an inverse. If A^{-1} exists, A is known as *invertible* or *non-singular*.

Some useful identities for matrices that are invertible are:

- $(A^{-1})^{-1} = A$
- $(AB)^{-1} = B^{-1}A^{-1}$
- $A^{-T} \triangleq (A^T)^{-1} = (A^{-1})^T$

3.1 Pseudoinverse

Frequently in linear algebra problems, you want to solve the equation $AX = B$ for X. You would like to compute A^{-1} and multiply both sides to get $X = A^{-1}B$. In python, this command would be: np.linalg.inv(A)*B.

However, for large floating point matrices, calculating inverses can be very expensive and possibly inaccurate. An inverse could also not even exist for A. What should we do?

Luckily, we have what is known as a *pseudoinverse*. This other matrix can be used to solve for $AX = B$. Python will try many methods, including using the pseudo-inverse, if you use the following command: np.linalg.solve(A,B). Additionally, using the pseudo-inverse, Python finds the closest solution if there exists no solution to $AX = B$.

4 Matrix Rank

- The rank of a transformation matrix A tells you how many dimensions it transforms a matrix to.
- col-rank(A) = maximum number of linearly independent column vectors of A
- row-rank(A) = maximum number of linearly independent row vectors of A. Column rank always equals row rank.
- For transformation matrices, the rank tells you the dimensions of the output.
- For instance, if rank of A is 1, then the transformation $p' = Ap$ points onto a line.
- Full rank matrix- if mxm and rank is m
- Singular matrix- if mxm matrix rank is less than m, because at least one dimension is getting collapsed. (No way to tell what input was from result) –> inverse does not exist for non-square matrices.

5 Eigenvalues and Eigenvectors (SVD)

5.1 Definitions

An *eigenvector* **x** of a linear transformation A is a non-zero vector that, when A is applied to it, does not change its direction. Applying A to the eigenvector scales the eigenvector by a scalar value λ, called an *eigenvalue*.

The following equation describes the relationship between eigenvalues and eigenvectors:

$$A\mathbf{x} = \lambda\mathbf{x}, \quad \mathbf{x} \neq \mathbf{0}$$

5.2 Finding eigenvectors and eigenvalues

If we want to find the eigenvalues of A, we can manipulate the above definition as follows:

$$\begin{aligned} A\mathbf{x} &= \lambda\mathbf{x}, \quad \mathbf{x} \neq \mathbf{0} \\ A\mathbf{x} &= (\lambda I\mathbf{x}), \quad \mathbf{x} \neq \mathbf{0} \\ (\lambda I - A)\mathbf{x} &= \mathbf{0}, \quad \mathbf{x} \neq \mathbf{0} \end{aligned}$$

Since we are looking for non-zero **x**, we can equivalently write the above relation as:

$$|\lambda I - A| = \mathbf{0}$$

Solving this equation for λ gives the eigenvalues of A, and these can be substituted back into the original equation to find the corresponding eigenvectors.

5.3 Properties

- The trace of A is equal to the sum of its eigenvalues:
$$tr(A) = \sum_{i=1}^{n} \lambda_i$$
- The determinant of A is equal to the product of its eigenvalues:
$$|A| = \prod_{i=1}^{n} \lambda_i$$
- The rank of A is equal to the number of non-zero eigenvalues of A.
- The eigenvalues of a diagonal matrix $D = diag(d_1, \ldots, d_n)$ are just the diagonal entries $d_1, \ldots d_n$.

5.4 Spectral Theory

5.4.1 Definitions

- An *eigenpair* is the pair of an eigenvalue and its associated eigenvector.
- An *eigenspace* of A associated with λ is the space of vectors where:
$$(A - \lambda I) = 0$$
- The *spectrum* of A is the set of all its eigenvalues:
$$\sigma(A) = \{\lambda \in \mathbb{C} : \lambda I - A \text{ is singular}\}$$
Where $\mathbb{C}$ is the space of all eigenvalues of A
- The *spectral radius* of A is the magnitude of its largest magnitude eigenvalue:
$$\rho(A) = max\{|\lambda_1|, \ldots, |\lambda_n|\}$$

5.4.2 Theorem: Spectral radius bound

Spectral radius is bounded by the infinity norm of a matrix:

$$\rho(A) = \lim_{k\to\infty} ||A^k||^{1/k}$$

Proof:

$$|\lambda|^k||\mathbf{v}|| = ||\lambda|^k\mathbf{v}|| = ||A^k\mathbf{v}||$$

By the Cauchy–Schwarz inequality ($||\mathbf{uv}|| \leq ||\mathbf{u}|| \cdot ||\mathbf{v}||$):

$$|\lambda|^k||\mathbf{v}|| \leq ||A^k|| \cdot ||\mathbf{v}||$$

Since $\mathbf{v} \neq 0$:

$$|\lambda|^k \leq ||A^k||$$

And we thus arrive at:

$$\rho(A) = \lim_{k\to\infty} ||A^k||^{1/k}$$

5.5 Diagonalization

An $n \times n$ matrix A is diagonalizable if it has n linearly independent eigenvectors.

Most square matrices are diagonalizable

- Normal matrices are diagonalizable
  Note: Normal matrices are matrices that satisfy:
  $$A^*A = AA^*$$
  Where A^* is the complex conjugate of A
- Matrices with n distinct eigenvalues are diagonalizable

Lemma: Eigenvectors associated with distinct eigenvalues are linearly independent.

To diagonalize the matrix A, consider its eigenvalues and eigenvectors. We can construct matrices D and V, where D is the diagonal matrix of the eigenvalues of A, and V is the matrix of corresponding eigenvectors:

$$D = \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix}$$

$$V = [v_1 \quad v_2 \quad \dots \quad v_n]$$

Since we know that:

$$AV = VD$$

We can diagonalize A by:

$$A = VDV^{-1}$$

If all eigenvalues are unique, then V is orthogonal. Since the inverse of an orthogonal matrix is its transpose, we can write the diagonalization as:

$$A = VDV^T$$

5.6 Symmetric Matrices

If A is symmetric, then all its eigenvalues are real, and its eigenvectors are orthonormal. Recalling the above diagonalization equation, we can diagonalize A by:

$$A = VDV^T$$

Using the above relation, we can also write the following relationship:

Given $y = V^T x$:

$$x^T Ax = x^T VDV^T x = y^T Dy = \sum_{i=1}^{n} \lambda_i y_i^2$$

Thus, if we want to do the following maximization:

$$max_{x \in \mathbb{R}^n}(x^T Ax) \quad \text{subject to } ||x||_2^2 = 1$$

Then the maximizing x can be found by finding the eigenvector corresponding to the largest eigenvalue of A.

5.7 Applications

Some applications of eigenvalues and eigenvectors include, but are not limited to:

- PageRank
- Schroedinger's equation
- Principle component analysis (PCA)

6 Matrix Calculus

6.1 The Gradient

If a function $f : \mathbb{R}^{\text{mxn}} \mapsto \mathbb{R}$ takes as input a matrix A of size $(m \times n)$ and returns a real value, then the gradient of f is

$$\nabla_A f(A) \in \mathbb{R}^{\text{mxn}} = \begin{bmatrix} \frac{\partial f(A)}{\partial A_{11}} & \frac{\partial f(A)}{\partial A_{12}} & \cdots & \frac{\partial f(A)}{\partial A_{1n}} \\ \frac{\partial f(A)}{\partial A_{21}} & \frac{\partial f(A)}{\partial A_{22}} & \cdots & \frac{\partial f(A)}{\partial A_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f(A)}{\partial A_{m1}} & \frac{\partial f(A)}{\partial A_{m2}} & \cdots & \frac{\partial f(A)}{\partial A_{mn}} \end{bmatrix}$$

Every entry in the matrix is:

$$\nabla_A f(A)_{ij} = \frac{\partial f(A)}{\partial A_{ij}}$$

The size of $\nabla_A f(A)$ is always the same as the size of A. Thus, if A is a vector x:

$$\nabla_x f(x) = \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{bmatrix}$$

6.2 The Gradient: Properties

- $\nabla_x(f(x) + g(x)) = \nabla_x f(x) + \nabla_x g(x)$
- For $t \in \mathbb{R}$, $\nabla_x(tf(x)) = t\nabla_x f(x)$

6.3 The Hessian

The Hessian matrix with respect to x can be written as $\nabla_x^2 f(x)$ or as H. It is an nxn matrix of partial derivatives

$$\nabla_x^2 f(x) \in \mathbb{R}^{\text{nxn}} = \begin{bmatrix} \frac{\partial^2 f(x)}{\partial x_1^2} & \frac{\partial^2 f(x)}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(x)}{\partial x_2 \partial x_1} & \frac{\partial^2 f(x)}{\partial x_2^2} & \cdots & \frac{\partial^2 f(x)}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x)}{\partial x_n \partial x_1} & \frac{\partial^2 f(x)}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_n^2} \end{bmatrix}$$

Every entry in the matrix is:

$$\nabla_x^2 f(x)_{ij} = \frac{\partial^2 f(x)}{\partial x_i \partial x_j}$$

It's important to note that the Hessian is the gradient of **every entry** of the gradient of the vector. For instance, the first column of the Hessian is the gradient of $\frac{\partial f(x)}{\partial x_1}$.

6.4 The Hessian: Properties

Schwarz's theorem: The order of partial derivatives doesn't matter so long as the second derivative exists and is continuous.

Thus, the Hessian is always symmetric:

$$\frac{\partial^2 f(x)}{\partial x_i \partial x_j} = \frac{\partial^2 f(x)}{\partial x_j \partial x_i}$$

6.5 Example Calculations

6.5.1 Example Gradient Calculation

For $x \in \mathbb{R}^n$, let $f(x) = b^T x$ for some known vector $b \in \mathbb{R}^n$

$$f(x) = [b_1 \quad b_2 \quad \cdots \quad b_n]^T \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

Thus,

$$f(x) = \sum_{i=1}^{n} b_i x_i$$

$$\frac{\partial f(x)}{\partial x_k} = \frac{\partial}{\partial x_k} \sum_{i=1}^{n} b_i x_i = b_k$$

Therefore, we can conclude that: $\nabla_x b^T x = b$.

6.5.2 Example Hessian Calculation

Consider the quadratic function $f(x) = x^T A x$

$$f(x) = \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} x_i x_j$$

$$\frac{\partial f(x)}{\partial x_k} = \frac{\partial}{\partial x_k} \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} x_i x_j$$

$$= \frac{\partial}{\partial x_k} [\sum_{i \neq k} \sum_{j \neq k} A_{ij} x_i x_j + \sum_{i \neq k} A_{ik} x_i x_k + \sum_{j \neq k} A_{kj} x_k x_j + A_{kk} x_k^2]$$

$$= \sum_{i \neq k} A_{ik} x_i + \sum_{j \neq k} A_{kj} x_j + 2A_{kk} x_k$$

$$= \sum_{i=1}^{n} A_{ik} x_i + \sum_{j=1}^{n} A_{kj} x_j = 2 \sum_{i=1}^{n} A_{ki} x_i$$

$$\frac{\partial^2 f(x)}{\partial x_k \partial x_l} = \frac{\partial}{\partial x_k} [\frac{\partial f(x)}{\partial x_l}] = \frac{\partial}{\partial x_k} [\sum_{i=1}^{n} 2A_{li} x_i]$$

$$= 2A_{lk} = 2A_{kl}$$

Thus,

$$\nabla_x^2 f(x) = 2A$$

Lecture #4: Pixels and Filters

Brian Hicks, Alec Arshavsky, Sam Trautwein, Christine Phan, James Ortiz
Department of Computer Science
Stanford University
Stanford, CA 94305
{bhicks2, arshava, rodion1, cxphan, jameso2}@cs.stanford.edu

Contents:

1 Image Sampling and Quantization

1.1 Image Types

Binary Images contain pixels that are either black (0) or white (1).

Grayscale Images have a wider range of intensity than black and white. Each pixel is a shade of gray with pixel values ranging between 0 (black) and 255 (white).

Color Images have multiple color channels; each color image can be represented in different color models (e.g., RGB, LAB, HSV). For example, an image in the RGB model consists of red, green, and blue channel. Each pixel in a channel has intensity values ranging from 0-255. Please note that this range depends on the choice of color model. A 3D *tensor* usually represents color images (Width x Length x 3), where the 3 channels can represent the color model such as RGB (Red-Green-Blue), LAB (Lightness-A-B), and HSV (Hue-Saturation-Value).

1.2 Sampling and Resolution

Images are **samples**: they are not continuous; they consist of discrete pixels of a certain size and density. This can lead to errors (or graininess) because pixel intensities can only be measured with a certain resolution and must be approximated.

Resolution is a sampling parameter, defined in dots per inch (DPI). The standard DPI value for screens is 72 DPI.

Pixels are quantized (i.e, all pixels (or channels of a pixel) have one of a set numbers of values (usually [0, 255]). Quantization and sampling loses information due to a finite precision.

2 Image Histograms

Histograms measure the frequency of brightness within the image: how many times does a particular pixel value appear in an image.

Figure 1: Illustrations of different pixel densities. Taken from the accompanying lecture slides. (Slide 14, slide credit Ulas Bagci)

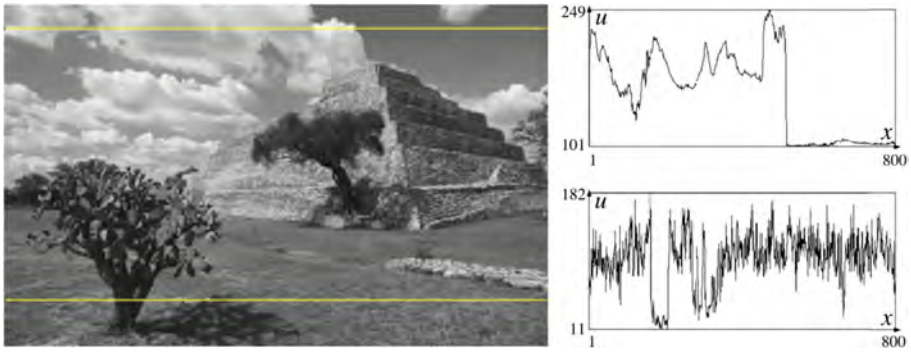


Figure 2: The image is sampled at two vertical positions, sampling a patch of sky and sampling a patch of grass. The corresponding histograms are shown to the right. Adapted from the accompanying lecture slide (Slide 23, slide credit Dr. Mubarak Shah

Histograms help us detect particular features in images, for example:

- Sky: smooth coloration denotes consistency in image, consistent with the image of a sky.
- Grass: a jagged histogram shows wide ranging variety in coloration, consistent with the shadows of a grass field.
- Faces: the color composition of a face will be displayed in a histogram.

An image histogram measures the frequency of certain grayscale intensities in an image. Histograms can be used to provide a quantifiable description of what things look like; this can be used as input to classifiers.

3 Images as Functions

Most images that we deal with in computer vision are digital, which means that they are discrete representations of the photographed scenes. This discretization is achieved through the sampling of 2-dimensional space onto a regular grid, eventually producing a representation of the image as a matrix of integer values.

When dealing with images, we can imagine the image matrix as infinitely tall and wide. However, the displayed image is only a finite subset of this infinite matrix. Having employed such definition of images, we can write them as coordinates in a matrix

$$\begin{bmatrix} \ddots & & \vdots & & \iddots \\ & f[-1,1] & f[0,1] & f[1,1] & \\ \dots & f[-1,0] & f[0,0] & f[0,1] & \dots \\ & f[-1,-1] & f[0,-1] & f[1,-1] & \dots \\ \iddots & & \vdots & & \ddots \end{bmatrix}$$

Figure 3: Graphical representation of a system's mapping of f to g

An Image can also be treated as a function $f : \mathbb{R}^2 \to \mathbb{R}^N$. When we do so, $f[n, m]$ where $f[n, m]$ is the intensity of a pixel at position (m, n). Note that we use square brackets, rather than the typical parentheses, to denote discrete functions.

When we treat an image as a function, it is defined over a rectangle with finite range. For example, the following function f returns the (grayscale) intensity of a single pixel in an image located between a and b horizontally and c and d vertically.

$$f : [a, b] \times [c, d] \to [0, 255] \qquad \text{(Grayscale Pixel Intensity)}$$

The set of values $[a, b] \times [c, d]$ is known as the *domain support*, and contains all values that are valid inputs to the function f, while $[0, 255]$ (in this case) is the range defining the set of possible outputs.

An Image can also be treated as a function mapping $\mathbb{R}^2 \to \mathbb{R}^3$. For example the RGB intensities of a given pixel can be written as the function g below

$$g[x, y] = \begin{bmatrix} r[x, y] \\ g[x, y] \\ b[x, y] \end{bmatrix} \qquad \text{(Color Pixel Intensity)}$$

where $r, g, b : [a, b] \times [c, d] \to [0, 255]$.

4 Linear Systems (Filters)

The term *filtering* refers to a process that forms a new image, the pixel values of which are transformations of the original pixel values. In general, the purpose in applying filters is the extraction of useful information (e.g., edge detection) or the adjustment of an image's visual properties (e.g., de-noising).

Filters are examples of *systems*, which are units that convert an input function $f[m, n]$ to an output (or response) function $g[m, n]$ where m, n are the independent variables. When dealing with images, m, n are the representation of a spatial position in the image.

Notationally, $\mathcal{S}$ is referred to as the *system operator*, which maps a member of the set of possible outputs $g[m, n]$ to a member of the set of possible inputs $f[m, n]$. When using notation involving $\mathcal{S}$, we can write that

$$\mathcal{S}[\mathrm{f}] = \mathsf{g}$$
$$\mathcal{S}\{f[m, n]\} = g[m, n]$$
$$f[m, n] \xrightarrow{\mathcal{S}} g[m, n]$$

4.1 Examples of Filters

Moving Average

One intuitive example of a filter is the *moving average*. This filter sets the value of a pixel to be the average of its neighboring pixels (e.g., the nine pixels in a 3×3 radius, when applying a 3×3 filter). Mathematically, we can represent this as

$$g[m, n] = \frac{1}{9} \sum_{i=-1}^{1} \sum_{j=-1}^{1} f[m - i, n - j] \qquad \text{(Weighted Average)}$$

This weighted average filter serves to smooth out the sharper edges of the image, creating a blurred or smoothed effect.

Image Segmentation

We can also use filters to perform rudimentary *image segmentation* based on a simple threshold system. In this case, the filter sets the value of a pixel either to an extremely high or an extremely low value, depending on whether or not it meets the threshold t. Mathematically, we write this as

$$g[m,n] = \begin{cases} 255 & f[m,n] \geq t \\ 0 & \text{otherwise} \end{cases} \qquad \text{(Threshold)}$$

This basic image segmentation filter divides an image's pixels into binary classifications of bright regions and dark regions, depending on whether or not the $f[m,n] \geq t$.

4.2 Properties of Systems

When discussing specific systems, it is useful to describe their properties. The following includes a list of properties that a system *may* possess. However, not all systems will have all (or any) of these properties. In other words, these are potential characteristics of individual systems, not traits of systems in general.

Amplitude Properties

- *Additivity*: A system is additive if it satisfies the equation

$$\mathcal{S}[f_i[m,n] + f_j[m,n]] = \mathcal{S}[f_i[m,n]] + \mathcal{S}[f_j[m,n]]$$

- *Homogeneity*: A system is homogeneous if it satisfies the equation

$$\mathcal{S}[\alpha f_i[n,m]] = \alpha \mathcal{S}[f_i[n,m]$$

- *Superposition*: A system has the property of superposition if it satisfies the equation

$$\mathcal{S}[\alpha f_i[n,m]] + \beta f_j[n,m]] = \alpha \mathcal{S}[f_i[n,m]] + \beta \mathcal{S} f_j[n,m]]$$

- *Stability*: A system is stable if it satisfies the inequality

$$|f[n,m]| \leq k \Longrightarrow |g[n,m]| \leq ck$$

  for some c.

- *Invertibility*: A system is invertible if it satisfies the equation

$$\mathcal{S}^{-1}[\mathcal{S}[f[n,m]]] = f[n,m]$$

Spatial Properties

- *Causality*: A system is causal if for $m < m_0$ and $n < n_0$

$$f[m,n] = 0 \Longrightarrow g[m,n] = 0$$

- *Shift Invariance*: A system is shift invariant if

$$f[m - m_0, n - n_0] \xrightarrow{\mathcal{S}} g[m - m_0, n - n_0]$$

4.3 Linear Systems

A *linear system* is a system that satisfies the property of superposition. When we employ a linear system for filtering; we create a new image whose pixels are weighted sums of the original pixel values, using the same set of weights for each pixel. A *linear shift-invariant system* is a linear system that is also shift invariant.

Linear systems also have what is known as an *impulse response*. To determine the impulse response of a system $\mathcal{S}$, consider first $\delta_2[m,n]$. This is a function defined as follows

$$\delta_2[m,n] = \begin{cases} 1 & m = 0 \text{and} n = 0 \\ 0 & \text{otherwise} \end{cases}$$

The impulse response r is then simply

$$r = \mathcal{S}[\delta_2]$$

A simple linear shift-invariant system is a system that shifts the pixels of an image, based on the shifting property of the delta function.

$$f[m,n] = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f[i,j]\delta_2[m-i,n-j]$$

We can then use the superposition property to write *any* linear shift-invariant system as a weighted sum of such shifting system

$$\alpha_1 \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f[i,j]\delta_{2,1}[m-i,n-j] + \alpha_{2,2} \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f[i,j]\delta_{2,3}[m-i,n-j] + \dots$$

We then define the filter h of a linear shift-invariant system as

$$h[m,n] = \alpha_1\delta_{2,1}[m-i,n-j] + \alpha_2\delta_{2,2}[m-i,n-j] + \dots$$

Impulse response (for all linear systems) Delta[n,m] function: has value that is 1 specifically at one pixel, gives back a response, h[n,m] A shifted delta function gives back a shifted response

Linear shift invariant systems (LSI) Example: a moving average filter is a summation of impulse responses

- Systems that satisfy the superposition property
- Have an *impulse response*: $\mathcal{S}[\delta_2[n,m]] = \delta_2[n,m]$
- Discrete convolution: $f[n,m] * h[n,m]$ (multiplication of shifted-version of impulse response by original function)

5 Convolution and Correlation

5.1 Convolution

The easiest way to think of convolution is as a system that uses information from neighboring pixels to filter the target pixel. A good example of this is a moving average, or a box filter discussed earlier.

Convolution is represented by *, for example:
$f[n,m]$*$h[n,m]$ represents a function being multiplied by a shifted impulse response

Convolution allows us to compute the output of passing any input signal through a system simply by considering the impulse response of the system. To understand what this means, we must first understand how to break up a signal into a set of impulse functions.

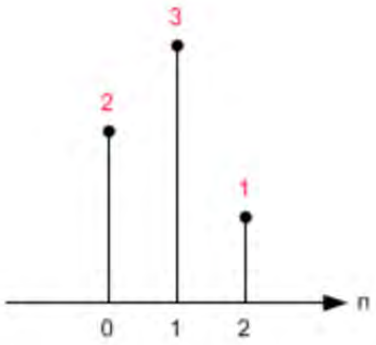
The impulse (delta) function, $\delta[n]$, is defined to be 1 at $n = 0$ and 0 elsewhere. As seen in the image below, any signal can be decomposed as the weighted sum of impulse functions.

More generally, an arbitrary signal x can be written as $x[n] = \sum_{k=-\infty}^{\infty} x[k]\delta[n-k]$.

The impulse response of a system, $h[n]$, is defined as the output resulting from passing an impulse function into a system. When a system is linear, scaling the impulse function results in the scaling of the impulse response by the same amount. Moreover, when the system is shift-invariant, shifting the impulse function shifts the impulse response by the same amount. The image below illustrates these properties for a signal consisting of 3 components.

More generally, for an arbitrary input signal $x[n] = \sum_{k=-\infty}^{\infty} x[k]\delta[n-k]$ passed into a linear, shift-invariant system, the output is $y[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k]$, i.e. the convolution of the signal x with the impulse response h.

Convolution can also be done in 2 dimensions. For example, when an arbitrary, 2D signal $x[n,m] = \sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty} x[i,j]\delta[n-i,m-j]$ is passed into a linear, shift-invariant system, the output is $y[n,m] = \sum_{i=-\infty}^{\infty}\sum_{j=-\infty}^{\infty} x[i,j]h[n-i,m-j]$, i.e. the convolution of the signal x with the impulse response h in 2 dimensions.



$$x[n] = x[0]\delta[n] + x[1]\delta[n-1] + x[2]\delta[n-2]$$

Figure 4: An example decomposition of a signal into an impulse function. (Adapted from http://www.songho.ca/dsp/convolution/convolution.html)

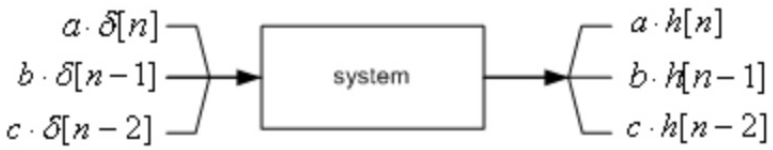


Figure 5: An impulse function is sent through a system to create an impulse response function. Adapted from http://www.songho.ca/dsp/convolution/convolution.html

5.2 Correlation

Cross correlation is the same as convolution, except that the filter kernel is not flipped. Two-dimensional cross correlation is represented as:
$r[k,l] = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f[m+k, n+l]g[m,n]$
It can be used to find known features in images by using a kernel that contains target features.

Acknowledgments

We would like to acknowledge Professor Niebles, Ranjay Krishna, and the rest of the teach staff for compiling the invaluable accompanying lecture slides and delivering such an informative lecture.

Lecture #5: Edge detection

Stephen Konz, Shivaal Roy, Charlotte Munger, Christina Ramsey, Alex Iyabor Jr
Department of Computer Science
Stanford University
Stanford, CA 94305
{swkonz, shivaal, cmunger, cmramsey, aiyabor}@cs.stanford.edu

1 Continuing from Last Lecture

1.1 Linear Systems

Linear Systems (Filters) form new images, the pixels of which are a weighted sum of select groupings of the original pixels. The use of different patterns and weights amplifies different features within the original image. A system S is a linear system If and Only If it satisfies the Superposition Property of systems:

$$S[\alpha f_i[n,m] + \beta f_j[h,m]] = \alpha S[f_i[n,m]] + \beta S[f_j[h,m]]$$

As previously introduced in the last lecture, the process of applying a filter to an input image is referred to as *convolution*.

1.2 LSI (Linear Shift Invariant Systems) and the impulse response

A shift invariant system is one in which shifting the input also shifts the output an equal amount. A linear shift-invariant (LSI) system's is characterized by its response to an impulse; this response is known as the *impulse response*. The impulse response helps us to understand the output of an LSI system for any given input signal.

1.3 Why are Convolutions Flipped?

Proof of $2D$ cross correlation's commutativity:

$$f[n,m] * *h[n,m] = \sum_k \sum_l f[k,l] \cdot h[n-k, m-l]$$

let $N = n - k$, $M = m - l$ so $k = n - N$ and $l = m - M$

$$\begin{aligned} &= \sum_k \sum_l f[n-N, m-M] \cdot h[N,M] \\ &= \sum_N \sum_M h[N,M] \cdot f[n-N, m-M] \\ &= h[n,m] * *f[n,m] \end{aligned}$$

1.3.1 Example

Apply kernel k to matrix M.

$$M = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, k = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,1} & x_{3,2} & x_{3,3} \end{bmatrix}$$

$$M * k = \begin{bmatrix} x_{3,3} & x_{3,2} & x_{3,1} \\ x_{2,3} & x_{2,2} & x_{2,1} \\ x_{1,3} & x_{1,2} & x_{1,1} \end{bmatrix}$$

Here, the kernel is expected to match the convolution. Instead, the output is equal to the kernel flipped in the x and y direction. To rectify this, the kernel is flipped at the initial step to ensure correct output form.

1.4 Convolution vs Cross Correlation

A convolution is an integral that expresses the amount of overlap of one function as it is shifted over another function. We can think of the convolution as a filtering operation.

The *Correlation* calculates a similarity measure for two input signals (e.g., two image patches). The output of correlation reaches a maximum when the two signals match best. Correlation can be though of as a measure of relatedness of two signals.

2 Edge Detection in Mammals

2.1 Hubel & Wiesel

In a series of experiments conducted by Hubel and Wiesel [1], the neuron responses in a cat's brain were recorded to observe how each part of the brain responds to different stimuli. They revealed that the cat's neurons were most excited by the edges at different orientations; namely, certain neurons correlated to specific orientations of edges or edges that moved in specific directions. Having one of these edges moving in its field of vision would cause that particular neuron to fire excitedly.

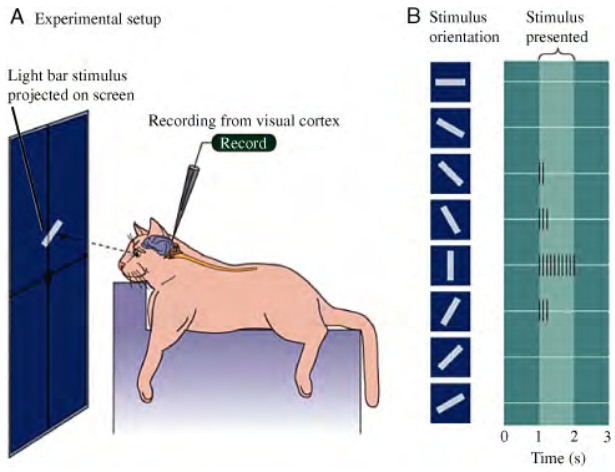


Figure 1: The Hubel and Wiesel's experiment. Source: [1]; Lecture 5, slide 34

2.2 Biederman

Biederman investigated the rate at which humans can recognize the object they're looking at. To test this, he drew outlines of common and recognizable objects and split them into two halves, with each line segment divided in only one of the halves. These outlines were then shown to partipants to test whether they could recognize the original objects while only seeing half of the original outline.

Surprisingly, he observed no difference in terms of the speed with which people recognized the objects. It was easy for them to recognize an object via only parts of its edges. This study benefits computer vision by providing an insight: even if only part of the original image is shown, a system theoretically should still be able to recognize the whole object or scene.

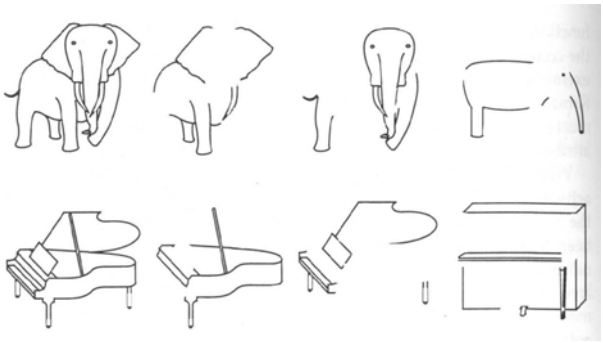


Figure 2: The examples of the Biederman's outlines. Source: Lecture 5, slide 36

2.3 Walther, Chai, Caddigan, Beck & Fei-Fei

In similar experiments, another group of researchers took two variations of the same image - the original color image and the outline of that image - to test color image vs line image recognition in humans. They traced recognition through different layers, each one a different level of processing in the visual pcortex of the brain. They found that lower layers could recognize the scenes faster via the line image, but as it moved up the layers of the brain (which encode increasingly higher level concepts), the color drawings were much more helpful in allowing people to recognize the scene as compared to the line drawing. This is believed to have happened because lower layers are better at recognizing pieces, like edges, while higher layers are better at recognizing concepts (like "human," "chair," or "tiger").

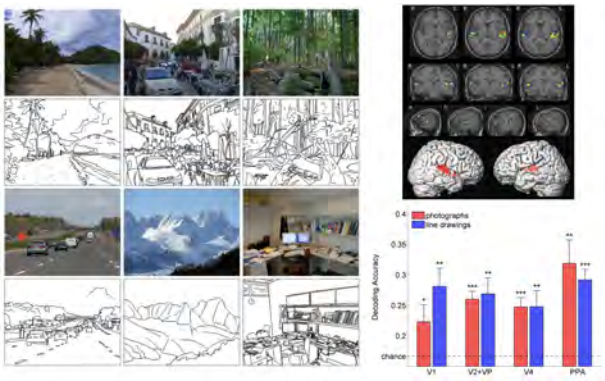


Figure 3: The visualization of the images and outcomes. Source: [2];Lecture 5, Slide 37

3 Edge Detection for Computer Vision

The goal of edge detection is to identify sudden changes (*discontinuities*) in an image. Intuitively, most semantic and shape information from the image can be encoded in its edges.

The edges help us extract information, recognize objects, and recover geometry and viewpoint. They arise due to discontinuities in surface normal, depth, surface color, and illumination .

3.1 Types of Discrete Derivative in 1D

There are three main types of derivatives that can be applied on pixels. Their formulas and corresponding filters are:
Backward

$$\frac{df}{dx} = f(x) - f(x-1) = f'(x)$$

$$[0, 1, -1]$$

Forward:

$$\frac{df}{dx} = f(x) - f(x+1) = f'(x)$$

$$[-1, 1, 0]$$

Central:

$$\frac{df}{dx} = f(x+1) - f(x-1) = f'(x)$$

$$[1, 0, -1]$$

3.2 Discrete Derivative in 2D

Gradient vector

$$\nabla f(x, y) = \begin{bmatrix} f_x \\ f_y \end{bmatrix}$$

Gradient magnitude

$$|\nabla f(x, y)| = \sqrt{f_x^2 + f_y^2}$$

Gradient direction

$$\theta = tan^{-1}\Big(\frac{\frac{df}{dy}}{\frac{df}{dx}}\Big)$$

3.3 Example

The gradient at a matrix index can be approximated using neighboring pixels based on the central discrete derivative equation expanded to $2D$. A filter like

$$\frac{1}{3}\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

When overlapped on top of a pixel $x[m, n]$, such that the center of the filter is located at $x[m, n]$ shown with its neighbors below

$$\begin{bmatrix} ... & ... & ... & ... & ... \\ ... & x_{m-1,n-1} & x_{m-1,n} & x_{m-1,n+1} & ... \\ ... & x_{m,n-1} & x_{m,n} & x_{m,n+1} & ... \\ ... & x_{m+1,n-1} & x_{m+1,n} & x_{m+1,n+1} & ... \\ ... & ... & ... & ... & ... \end{bmatrix}$$

Produces an output of

$$\frac{1}{3}\Big((x_{m-1,n+1} - x_{m-1,n-1}) + (x_{m,n+1} - x_{m,n-1}) + (x_{m+1,n+1} - x_{m+1,n-1})\Big) =$$

Which is equivalent to an approximation of the gradient in the horizontal (n) direction at pixel (m, n). This filter detects the horizontal edges, and a separate kernel is required to detect vertical ones.

4 Simple Edge Detectors

4.1 Characterizing Edges

Characterizing edges (i.e., characterizing them properly so they can be recognized) is an important first step in detecting edges. For our purposes, we will define an edge as a rapid place of change in the image intensity function. If we plot the intensity function along the horizontal scanline, we see that the edges correspond to the extra of the derivative. Therefore, noticing sharp changes along this plot will likely give us edges.

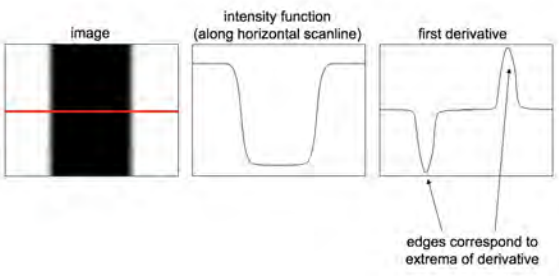


Figure 4: An image with intensity function and first derivative. Source: Lecture 5, slide 66

4.2 Image Gradient

The gradient of an image has been defined as the following:

$$\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right],$$

while its direction has been defined as:

$$\theta = \tan^{-1}\left(\frac{\partial f}{\partial y} / \frac{\partial f}{\partial x}\right).$$

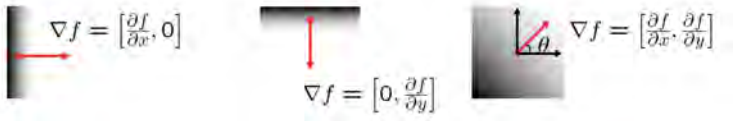


Figure 5: The gradient vector directions. Source: Lecture 5, slide 67

The gradient vectors point toward the direction of the most rapid increase in intensity. In vertical edge, for example, the most rapid change of intensity occurs in the x-direction.

Figure 6: The gradients as applied to the image of a tiger. Source: Lecture 5, slide 68

4.3 Effects of Noise

If there is excessive noise in an image, the partial derivatives will not be effective for identifying the edges.

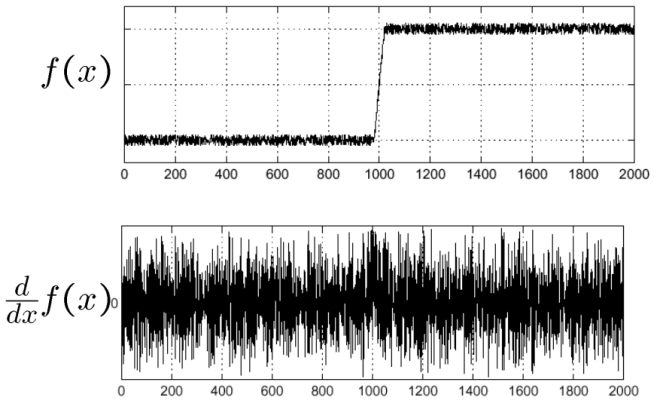


Figure 7: The derivative of an edge in a noisy image. Source: Steve Seitz; Lecture 5, slide 70

In order to account for the noise, the images must first be smoothed. This is a process in which pixel values are recalculated so that they more closely resemble their neighbors. The smoothing is achieved by means of convolving the image with a filter (e.g., gaussian kernel).

There are, of course, some concerns to keep in mind when smoothing an image. The image smoothing does remove noise, but it also blurs the edges; the use of large filters can result in the loss of edges and the finer details of the image.

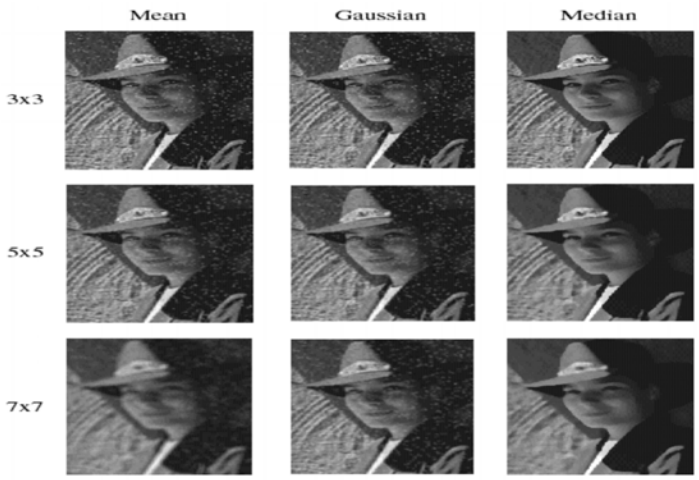


Figure 8: Smoothing with different filters and filter sizes. Source: Steve Seitz; Lecture 5, slide 75

The image smoothing facilitates the process of edge detection. After smoothing an image f, the search for peaks is initiated by calculating $f * \frac{d}{dx}g$ with kernel g.

4.4 Gaussian Blur

The Gaussian blur is the result of blurring an image by a Gaussian function to reduce image noise. It is a low-pass filter, meaning it attenuates high frequency signals. You will generally use Gaussian blurring as a preliminary step.
One dimension:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{x^2}{2\sigma^2}}$$

Two dimension:

$$G(x, y) = \frac{1}{2\pi\sigma} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

5 Designing a Good Edge Detector

An optimal edge detector must have certain qualities:

1. Good detection
   - It must minimize the probability of detecting false positives (which are spurious edges, generally caused by noise) and false negatives (missing real edges, which can be caused by smoothing, among other things). If it detects something as an edge, it should be an edge.
2. Good localization
   - The detected edges must be as close as possible to the actual edges in the original image. The detector must identify where the edges occur and pinpoint the exact location of the edge; it must also be consistent in determining which pixels are involved in each edge.
3. Silent response
   - It must minimize the number of local maxima around the true edge (returning one point only for each true edge point). It should tell you that there is one very specific edge instead of splitting one edge into multiple detected edges. In other words, only the real border of the edge is captured; other possibilities are suppressed.

Figure 9: Sample problems of bad edge detectors. Source: Lecture 5, slide 84

References

[1] DH Hubel and TN Wiesel. Receptive fields of optic nerve fibres in the spider monkey. *The Journal of physiology*, 154(3):572–580, 1960.

[2] Dirk B Walther, Barry Chai, Eamon Caddigan, Diane M Beck, and Li Fei-Fei. Simple line drawings suffice for functional mri decoding of natural scene categories. *Proceedings of the National Academy of Sciences*, 108(23):9661–9666, 2011.

Lecture #06: Edge Detection

Winston Wang, Antonio Tan-Torres, Hesam Hamledari
Department of Computer Science
Stanford University
Stanford, CA 94305
{wwang13, tantonio}@cs.stanford.edu; hesamh@stanford.edu

1 Introduction

This lecture covers edge detection, Hough transformations, and RANSAC. The detection of edges provides meaningful semantic information that facilitate the understanding of an image. This can help analyzing the shape of elements, extracting image features, and understanding changes in the properties of depicted scenes such as discontinuity in depth, type of material, and illumination, to name a few. We will explore the application of Sobel and Canny edge detection techniques. The next section introduces the Hough transform, used for the detection of parametric models in images;for example, the detection of linear lines, defined by two parameters, is made possible by the Hough transform. Furthermore, this technique can be generalized to detect other shapes (e.g., circles). However, as we will see, the use of Hough transform is not effective in fitting models with a high number of parameters. To address this model fitting problem, the random sampling consensus (RANSAC) is introduced in the last section; this non-deterministic approach repeatedly samples subsets of data, uses them to fit the model, and classifies the remaining data points as "inliers" or "outliers" depending on how well they can be explained by the fitted model (i.e., their proximity to the model). The result is used for a final selection of data points used in achieving the final model fit. A general comparison of RANSAC and Hough transform is also provided in the last section.

2 Edge Detection

2.1 Motivation for Edge Detection

Edge detection is extremely relevant for mammalian eyes. Certain neurons within the brain are adept at recognizing straight lines. The information from these neurons is put together in the brain for recognition of objects. In fact, edges are so useful for recognition in humans, line drawings are almost as recognizable as the original image (Fig. 1).We would like to be able to extract information, recognize objects, and recover the geometry and viewpoint of an image.

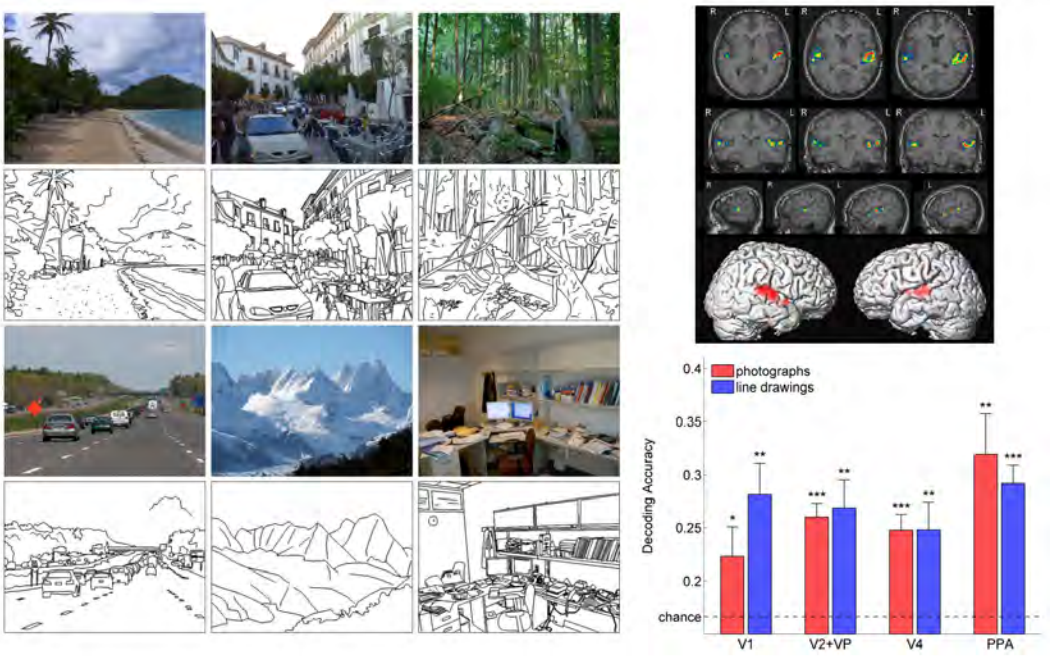


Fig. 1. Certain areas of the brain react to edges; the line drawings are as recognizable as the original image; image source: [4]

2.2 Edge Basics

There are four possible sources of edges in an image: surface normal discontinuity (surface changes direction sharply), depth discontinuity (one surface behind another), surface color discontinuity (single surface changes color), illumination discontinuity (shadows/lighting). These discontinuities are demonstrated in the Fig.2a; different types of edges can be seen in Fig.2b:

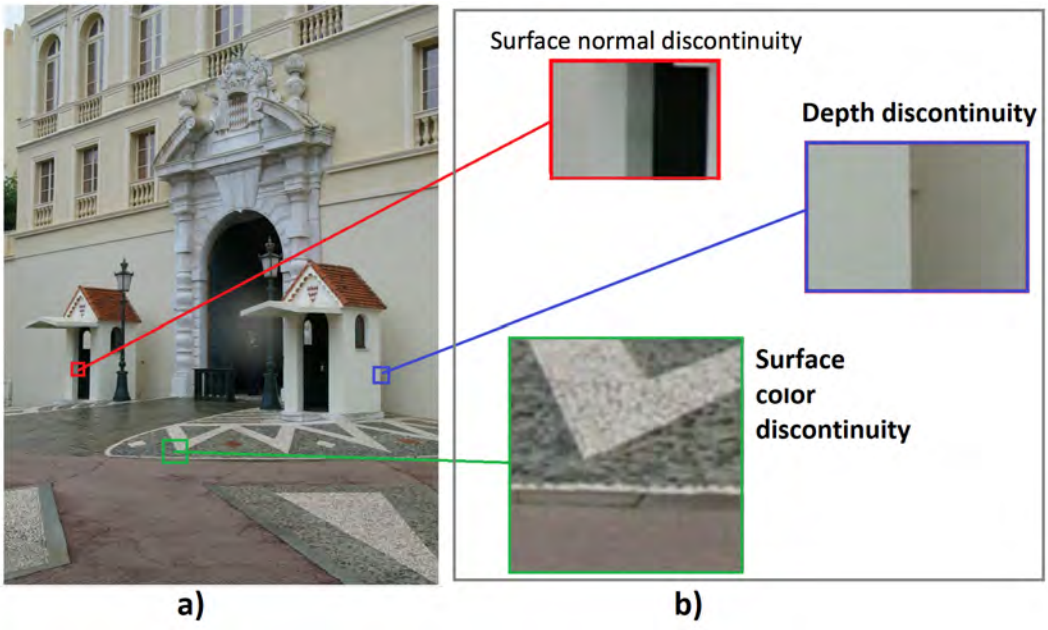


Fig. 2. Different types of edges due to discontinuities in surface color, surface depth, and surface normal (source: lecture notes)

Edges occur in images when the magnitude of the gradient is high

2.3 Finding the Gradient

In order to find the gradient, we must first find the derivatives in both the x and y directions

2.3.1 Discrete Derivatives

$$\frac{df(x)}{dx} = \lim_{\delta x \to 0} \frac{f(x) - f(x - \delta x)}{\delta x} = f'(x)$$
$$\frac{df(x)}{dx} = \frac{f(x) - f(x-1}{1} = f'(x)$$
$$\frac{df(x)}{dx} = f(x) - f(x-1) = f'(x)$$

It is also possible to take the derivative three different ways

- Backward: $f'(x) = f(x) - f(x-1)$
- Forward: $f'(x) = f(x+1) - f(x)$
- Central: $f'(x) = \frac{(x+1) - f(x-1)}{2}$

Each of these can also be represented as a filter (convoluting the filter with the image gives the derivative)

- Backward: $f'(x) = f(x) - f(x-1) \rightarrow [0, 1, -1]$
- Forward: $f'(x) = f(x) - f(x+1) \rightarrow [-1, 1, 0]$
- Central: $f'(x) = f(x+1) - f(x-1) \rightarrow [1, 0, -1]$

The gradient (∇f) can be calculated as follows:

$$\nabla f(x, y) = \begin{bmatrix} \frac{\partial f(x,y)}{\partial x} \\ \frac{\partial f(x,y)}{\partial y} \end{bmatrix}$$
$$= \begin{bmatrix} f_x \\ f_y \end{bmatrix}$$

We can also calculate the magnitude and the angle of the gradient:

$$|\nabla f(x, y)| = \sqrt{f_x^2 + f_y^2}$$
$$\theta = \tan^{-1}(f_y / f_x)$$

2.4 Reducing noise

Noise will interfere with the gradient, making it impossible to find edges using the simple method, even though the edges are still detectable to the eye. The solution is to first smooth the image.
Let f be the image and g be the smoothing kernel. Thus, in order to find the smoothed gradient, we must calculate (1D example):

$$\frac{d}{dx}(f * g)$$

By the derivative theorem of convolution:

$$\frac{d}{dx}(f * g) = f * \frac{d}{dx} g$$

This simplification saves us one operation. Smoothing removes noise but blurs edges. Smoothing with different kernel sizes can detect edges at different scales

2.5 Sobel Noise Detector

This algorithm utilizes 2 3×3 kernels which, once convolved with the image, approximate the x and y derivatives of the original image.

$$\mathbf{G}_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \qquad \mathbf{G}_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

These matrices represent the result of smoothing and differentiation

$$\mathbf{G}_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$$

The Sobel Filter has many problems, including poor localization. The Sobel Filter also favors horizontal and vertical edges over oblique edges

2.6 Canny Edge Detector

The Canny Edge Detector has five algorithmic steps:

- Suppress noise
- Compute gradient magnitude and direction
- Apply non-maximum suppression
- Hysteresis thresholding
- Connectivity analysis to detect edges

2.6.1 Suppress noise

We can both suppress noise and compute the derivatives in the x and y directions using a method similar to the Sobel filter.

2.6.2 Compute gradient magnitude and direction

From above,

$$|\nabla f(x, y)| = \sqrt{f_x^2 + f_y^2}$$
$$\theta = \tan^{-1}\left(f_y / f_x\right)$$

2.6.3 Apply non-maximum suppression

The purpose of this portion of the algorithm is to make sure the edges are specific. Thus, we assume that the edge occurs when the gradient reaches a maximum. We suppress any pixels that have a non-maximum gradient.
Basically, if the pixel is not the largest of the three pixels in the direction and opposite the direction of its gradient, it is set to 0. Furthermore, all gradients are rounded to the nearest 45°

2.6.4 Hysteresis thresholding

All remaining pixels are subjected to hysteresis thresholding. This part uses two values, for the high and low thresholds. Every pixel with a value above the high threshold is marked as a strong edge. Every pixel below the low threshold is set to 0. Every pixel between the two thresholds is marked as a weak edge

2.6.5 Connectivity analysis to detect edges

The final step is connecting the edges. All strong edge pixels are edges. For weak edge pixels, only the weak edge pixels that are linked to strong edge pixels are edges. The part uses BFS or DFS to find all the edges.

3 Hough Transforms

3.1 Intro to Hough Transform

Hough Transform is a way to detect particular structures in images, namely lines. However, Hough transform can be used to detect any structure whose paramteric equation is known. It gives a robust detector under noise and partial occlusion.

3.2 Goal of Hough Transform for detecting lines

Hough transform can be used to detect lines in images. To do this, we want to locate sets of pixels that make up straight lines in the image. This works to detect lines in an image after an edge detector is applied to get the pixels of just the edges (and thus we find which sets of those pixels make up straight lines).

3.3 Detecting lines using Hough Transform in a,b space

Say we have a x_i, y_i. There are infinite lines that could pass through this point. We can define a line that passes through this pixel x_i, y_i as

$$y_i = a * x_i + b$$

. Using this, we can transform each pixel into a, b space by re-writing this equation as:

$$b = -a * x_i + y_i$$

This equation represents a line in a, b space, and each a, b point on the line represents a possible line passing through our point x_i, y_i.

Thus, for each pixel x_i, y_i in our set of edge pixels, we transform it into a, b space to get a line.

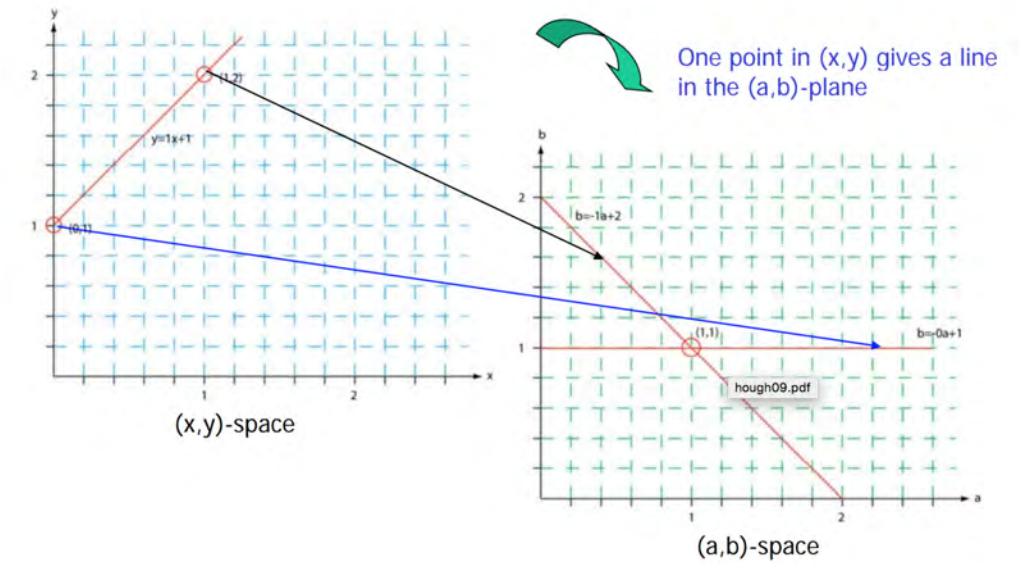


Fig. 3. The transformation from the original space to the Hough space; source: lecture slides

The intersection of lines in a, b space represent the a, b values that compromise a line $y_i = a * x_i + b$ passing through those points. Example: Say we have two points $x_1, y_1 = (1, 1)$, and $x_2, y_2 = (2, 3)$. We transform these points into a, b space with the lines $b = -a * 1 + 1$ and $b = -a * 2 + 3$. Solving for the intersection of these two lines gives us $a = 2$ and $b = -1$. This intersection point in (a, b) space gives us the values for the line that goes through both points in x, y space.

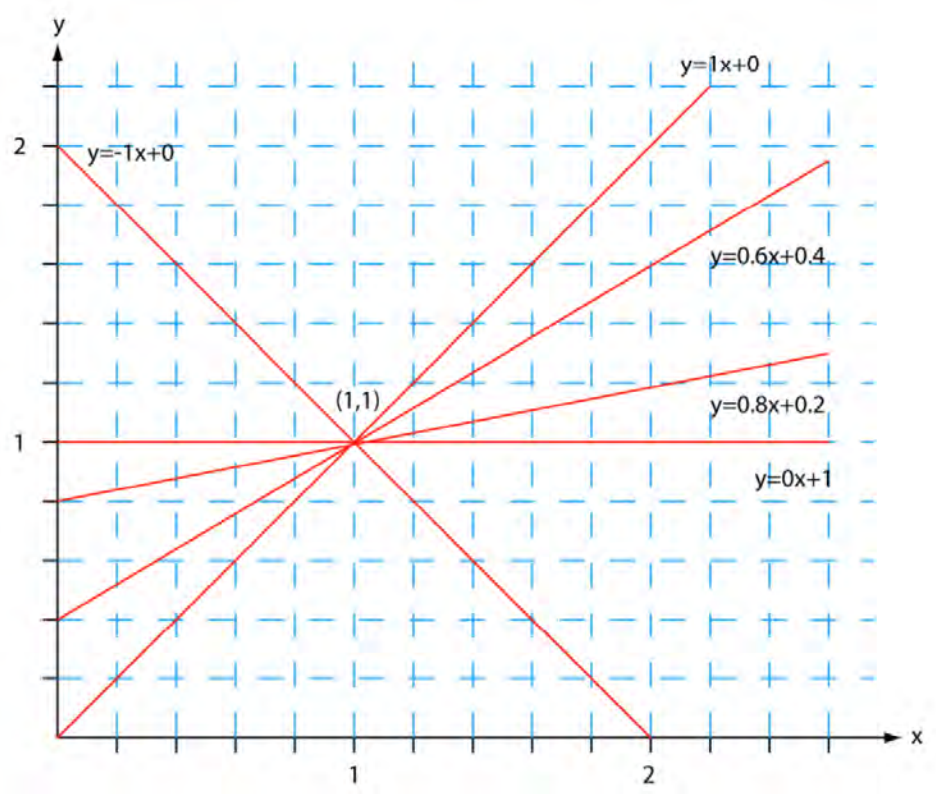


Fig. 4. The lines passing through a point in the original space; source: lecture slides

3.4 Accumulator Cells

In order to get the "best" lines, we quantize the a, b space into cells. For each line in our a, b space, we add a "vote" or a count to each cell that it passes through. We do this for each line, so at the end, the cells with the most "votes" have the most intersections and therefore should correspond to real lines in our image.

The algorithm for Hough transform in a, b space is as follows:

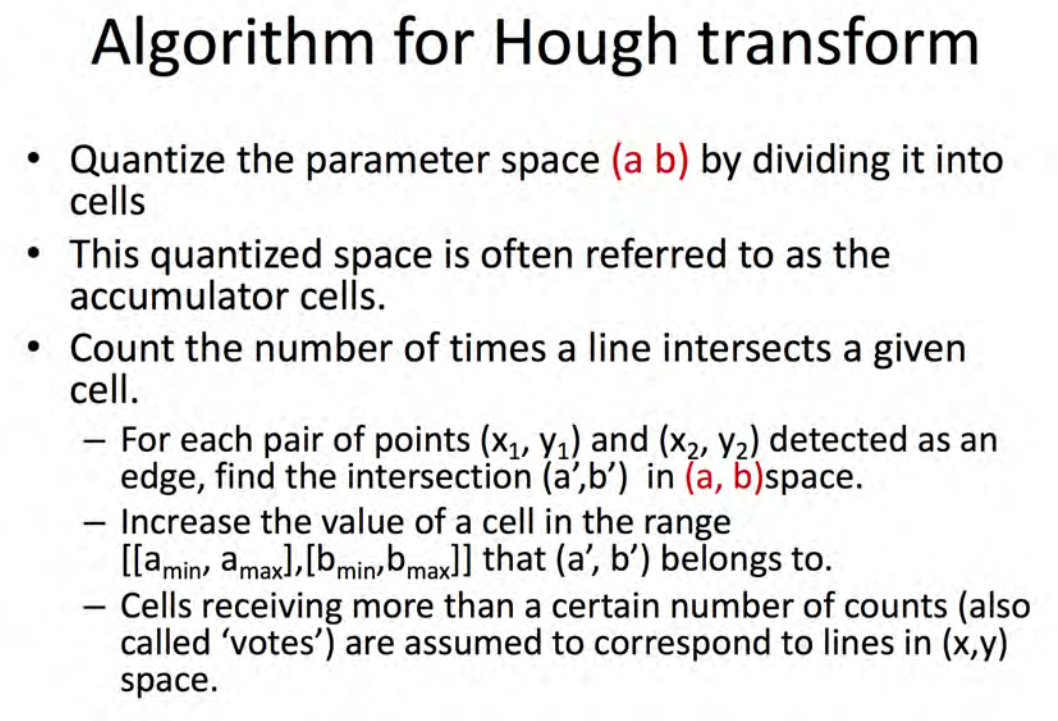


Fig. 5. The Hough transform algorithm; source: lecture slides

3.5 Hough transform in ρ, θ space

A problem with using a, b space to represent lines is that they are limited and cannot represent vertical lines. To solve this, we use polar coordinates to represent lines. For a pixel x_i, y_i, we transform it using the equation:

$$x * \cos\theta + y * \sin\theta = \rho$$

In ρ, θ space, points are not represented as lines but instead as sine wave-like functions.

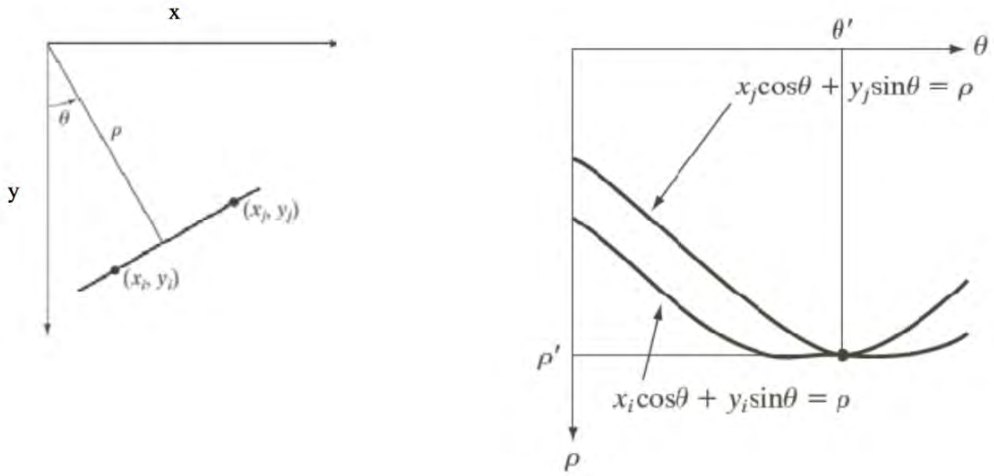


Fig. 6. The Hough transform in ρ, θ space; source: lecture slides

The intersection of these functions in ρ, θ space still correspond to the ρ, θ that comprise a line passing through those points.

So, for each pixel x_i, y_i we transform it into a function in ρ, θ space. We apply the same accumulator cell algorithm to count the most intersections of functions.

In this case, we quantize our ρ, θ space into cells, and add a "vote" to each cell that our function passes through. The cells with the most votes are the most likely real lines in our image.

3.6 Concluding Remarks

Advantages of the Hough Transform is that it is conceptually simple (just transforming and finding intersection in Hough space). It is also fairly easy to implement, and it can handle missing and occluded data well. Another advantage is that it can find other structures other than lines, as long as the structure has a parametric equation.

Some disadvantages include that it gets more computationally complex the more parameters you have. It can also only look for one kind of structure (so not lines and circles together). The length and the position of a line segment can also not be detected by this. It can be fooled by "apparent" lines, and co-linear line segments cannot be separated.

4 RANSAC

With the increase in model complexity (i.e., the number of parameters), the Hough transform loses its effectiveness; this section elaborates on the design of the RAndom Sample Consensus (RANSAC) technique [1] which provides a computationally efficient means of fitting models in images. We begin with an introduction of the RANSAC's basic idea and then introduce its algorithm.

4.1 Introduction to RANSAC Basics

The RANSAC algorithm is used for estimating the parameters of models in images (i.e., model fitting). The basic idea behind RANSAC is to solve the fitting problem many times using randomly selected minimal subsets of the data and choosing the best performing fit. To achieve this, RANSAC tries to iteratively identify the data points that correspond to model we are trying to fit.

Fig. 7a illustrates an example where a linear model (i.e., 2 parameters) is to be fitted to the data; while the majority of data points fit a linear model, the two points in the top right corner can significantly affect the accuracy of overall fit (if they are included in the fit). The RANSAC algorithm aims to address this challenge by identifying the "inliers" and "outliers" in the data.

RANSAC randomly selects samples of the data, with the assumption that if enough samples are chosen, there will be a low probability that at all samples provides a bad fit.

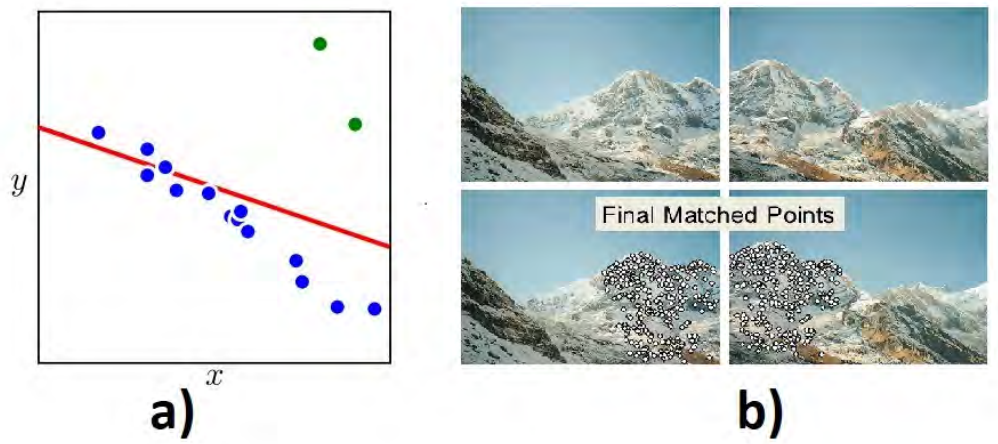


Fig. 7. a) the outliers are detected by RANSAC to improve parameter estimation; b) RANSAC application for image stitching; sources([3]and Derek Hoiem)

4.2 Applications

The RANSAC algorithm can be used to estimate parameters of different models; this is proven beneficial in image stitching (Fig. 6b), outlier detection, lane detection (linear model estimation), and stereo camera calculations.

4.3 The Algorithm

The RANSAC algorithm iteratively samples nominal subsets of the original data (e.g., 2 points for line estimation); the model is fitted to each sample, and the number of "inliers" corresponding to this fit is calculated; this includes the data points that are close to the fitted model. The points closer than a threshold (e.g., 2 standard deviation, or a pre-determined number of pixels) are considered "inliers". The fitted model is considered good if a big fraction of the data is considered as "inliers" for that fit. In the case of a good fit, the model is re-fitted using all the inliers, and the outliers are discarded. This process is repeated, and model estimates with a big enough fraction of inliers (e.g., bigger than a pre-specified threshold) are compared to choose the best-performing fit. Fig. 8 illustrates this process for a linear model and its three samples. The third sample (Fig. 8c) provides the best fit as it includes the most number of inliers.

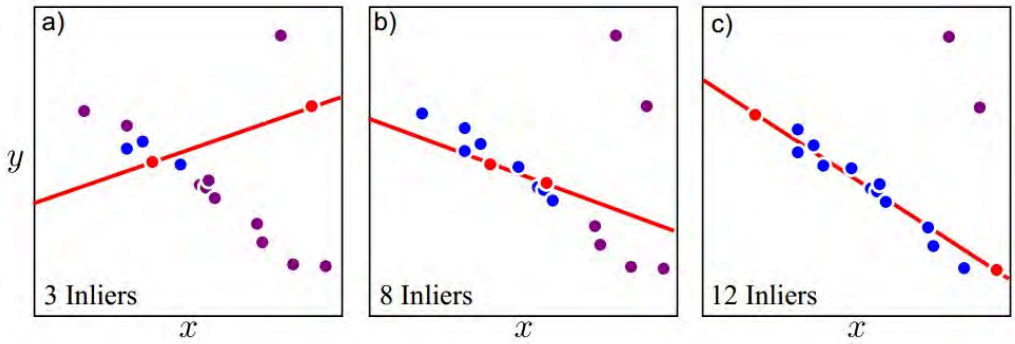


Fig. 8. The demonstration of the RANSAC algorithm for a linear model estimation and three random samples; source([3])

```
Determine n, t, d and k as above
Until there is a good fit or k iterations have occurred
    draw a sample of n points from the data
    uniformly and at random

    fit to that set of n points

    for each data point outside the sample

      test the distance from the point to the line
      against t; if the distance from the point to the line
      is less than t, the point is close

    end
    if there are d or more points close to the line
    then there is a good fit.  Refit the line using all
    these points, and terminate
end
```

Fig. 9. The pseudocode for RANSAC; source([2])

The RANSAC is detailed in Fig. 9. The major steps included in the RANSAC loop:

1. Randomly select a seed group from data.
2. Perform parameter estimation using the selected seed group.
3. Identify the inliers (points close to the estimated model).
4. (If there exists a sufficiently large number of inliers,)re-estimate the model using all inliers.
5. repeat steps 1-4 and finally keep the estimate with most inliers and best fit.

4.4 How Many Samples are Needed?

The RANSAC is a non-deterministic model fitting approach; this means that the number of samples need to be large enough to provide a high confidence estimate of parameters. The number of required samples depends on 1) the number of fitted parameters and 2) the amount of noise. Fig. 10 lists the minimum number of samples needed based on p=0.99 and variations of sample size (i.e., the number of parameters) and the fraction of outliers (i.e., noise). More samples are needed for estimating bigger models and noisier data. A large enough number of samples (k) need to be chosen such that a low probability of only seeing bad samples (P_f) is guaranteed:

$$P_f = (1 - W^n)^k = 1 - p$$

where W and n are respectively the fraction of inliers and the number of points needed for model fitting. The minimum number of samples:

$$k = \frac{log(1-p)}{log(1-W^n)}$$

RANSAC: Computed k (p=0.99)

Sample size	Proportion of outliers						
n	5%	10%	20%	25%	30%	40%	50%
2	2	3	5	6	7	11	17
3	3	4	7	9	11	19	35
4	3	5	9	13	17	34	72
5	4	6	12	17	26	57	146
6	4	7	16	24	37	97	293
7	4	8	20	33	54	163	588
8	5	9	26	44	78	272	1177

Fig. 10. The number of samples for different choices of noise population and model size; source: David Lowe)

4.5 Advantages, Limitations, and Considerations

The advantages of the RANSAC lie in its simple implementation and wide application range in the model fitting domain. Other advantages include its computational efficiency; the sampling approach provides a better alternative to solving the problem for all possible combinations of features.

In some cases, it'd be more efficient to use the Hough transform instead of the RANSAC:

1. The number of parameters are small; for example, linear model estimation (2 parameters) can be achieved efficiently using Hough transform, while image stitching requires a more computationally frugal approach such as RANSAC.
2. If the noise population is high; as we saw earlier, increase in noise requires a more extensive sampling approach (higher number of samples), increasing computation cost. Increased noise reduces the chances of correct parameter estimation and the accuracy of inlier classification.

The poor performance in highly noisy data is a primary limitation of the RANSAC; this is espcially crucial as real-world problems have high rate of outliers.

References

[1] Martin A Fischler and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[2] David Forsyth and Jean Ponce. *Computer vision: a modern approach*. Upper Saddle River, NJ; London: Prentice Hall, 2011.

[3] Simon JD Prince. *Computer vision: models, learning, and inference*. Cambridge University Press, 2012.

[4] Dirk B. Walther, Barry Chai, Eamon Caddigan, Diane M. Beck, and Li Fei-Fei. Simple line drawings suffice for functional mri decoding of natural scene categories. *Proceedings of the National Academy of Sciences*, 108(23):9661—-9666, 2011.

Lecture #6-7: Features and Fitting/Feature Descriptors

Trevor Danielson, Wesley Olmsted, Kelsey Wang, Ben Barnett
Department of Computer Science
Stanford University
Stanford, CA 94305
{trevord, wolmsted, kyw, ben.barnett}@cs.stanford.edu

1 RANSAC

1.1 Goal

RANdom SAmple Consensus (RANSAC) is used for model fitting in images (e.g., line detection); it can be extremely useful for object identification, among other applications. It is often more effective than pure edge detection which is prone to several limitations: edges containing extra points due to the noise/clutter, certain parts of the edges being left out, and the existence of noise in measured edges' orientation.

1.2 Motivation

One of the primary advantages of RANSAC is that it is relatively efficient and accurate even when the number of parameters is high. However, it should be noted that RANSAC is likely to fail or produce inaccurate results in images with a relatively large amount of noise.

1.3 General Approach

The intuition for RANSAC is that by randomly sampling a group of points in an edge and applying a line of best fit to those points *many times*, we have a high probability of finding a line that fits the points very well. Below is the general process "RANSAC loop":

1. Randomly select a seed group of points from the overall group of points you are trying to fit with a line.
2. Compute a line of best fit among the seed group. For example, if the seed group is only 2 distinct points, then it is clear to see that there is only one line that passes through both points, which can be determined with relative ease from the points' coordinates.
3. Find the number of **inliers** to this line by iterating over each point in the data set and calculating its distance from the line; if is less than a (predetermined) threshold value, it counts as an inlier. Otherwise, it is counted as an outlier.
4. If the number of inliers is sufficiently large, conclude that this line is "good", and that at least one line exists that includes the inliers tallied in the previous step. To further improve the line, re-compute the line using a least-squares estimate using all of the inliers that were within the threshold distance. Keep this transformation as the line that best approximates the data.

1.4 Drawbacks

The biggest drawback to RANSAC is its inefficient handling of highly noisy data; an increase in the fraction of outliers in a given data set results in an increase in the number of samples required for model fitting (e.g., line of best fit). More importantly, the noisier an image is, the less likely it is for a line to *ever* be considered sufficiently good at fitting the data. This is a significant problem because most real world problems have a relatively large proportion of noise/outliers.

2 Local Invariant Features

2.1 Motivation

The local invariant image features and their descriptors are used in a wide range of computer vision applications; they include, but are not limited to, object detection, classification, tracking, motion estimation, panorama stitching, and image registration. The previously discussed methods, such as cross-correlation, are not effective and robust in many of such appl. This method works by finding local, distinctive structures within an image (i.e., features), and it describes each feature using the surrounding region (e.g., a small patch centered on the detected feature). This "local" representation of image features (as opposed to a "global" one, such as cross-correlation) provides a more robust means of addressing the above-mentioned computer vision problems; such strategy is invariant to object rotations, point of view translations, and scale changes.

2.2 General Approach

The general approach for employing local invariant features is detailed below:

1. Find and define a set of distinctive keypoints.
2. Define a local region around the keypoint.
3. Extract and normalize the regional content from the area.
4. Compute a local descriptor from the normalized region (i.e., a function of pixel intensity).
5. Match local descriptors.

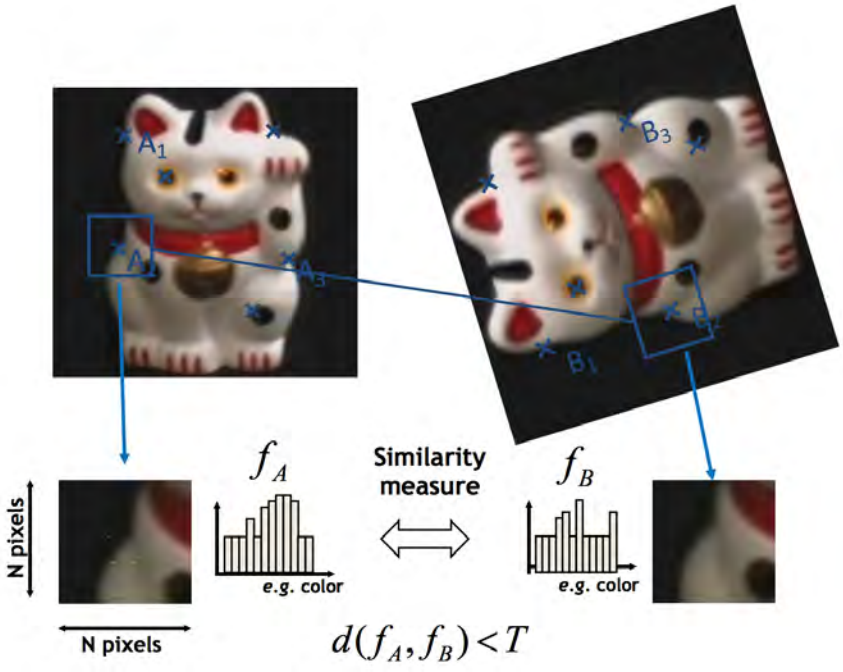


Figure 1: The local features are identified within similar pictures; descriptors are calculated for normalized patches and compared to each other for quantifying the similarity. Source: Lecture 6 slides.

2.3 Requirements

Good local features should have the following properties:

1. *Repeatability*: given the same object or scene under different image conditions such as lighting or change in viewpoint, a high amount of features should be detectable in both images being compared. In other words, the feature detection should be invariant to rotations and viewpoint changes and robustly handle lighting changes, noise, and blur.
2. *Locality*: features should be local to avoid issues caused by occlusion and clutter.
3. *Quantity*: enough features should be chosen to ensure the accuracy of techniques relying on descriptors such as object detection.
4. *Distinctiveness*: results need to contain "salient" image features that show a large amount of variation; this ensures that the detected features can be distinguished from each other and properly matched between different images.
5. *Efficiency*: feature matching in new images should be conducive to real-time applications.

3 Keypoint Localization

3.1 Motivation

The goal of keypoint localization is to detect features consistently and repeatedly, to allow for more precise localization, and to find interesting content within the image.

3.2 General Approach

We will look for corners since they are repeatable and distinctive in the majority of images. To find corners, we look for large changes in intensity in all directions. To provide context, a "flat" region would not have change in any direction, and an edge would have no change along the direction of the edge. We will find these corners using the Harris technique.

3.3 Harris Detector

The intuition behind Harris detector is that if a window (w) slides over an image, the change in the intensity of pixel values caused by the shift is highest at corners. This is because change in pixel intensity is observed in both directions (x and y) at corners, while it is limited to only one direction at the edges, and it is negligible at flat image regions. To calculate the change of intensity due to the shift $[u, v]$:

$$E(u,v) = \sum_{x,y} w(x,y)[I(x+u, y+v) - I(x,y)]^2$$

To find corners, we must maximize this function. Taylor Expansion is then used in the process to get the following equation:

$$E(u,v) = \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix}$$

where we have M defined as:

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x I_x & I_x I_y \\ I_x I_y & I_y I_y \end{bmatrix}$$

This matrix reveals:

$$M = \begin{bmatrix} \sum I_x I_x & \sum I_x I_y \\ \sum I_x I_y & \sum I_y I_y \end{bmatrix} = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$$

Corners have both large and similar eigenvalues, whereas edges have one significantly greater eigenvalue, and flat regions have two small eigenvalues. Because the calculation of eigenvalues is computationally intensive, an alternative approach is employed where the Corner Response Function (θ) is used to compute a score for each window:

$$\theta = det(M) - \alpha \text{trace}(M)^2$$

where α is a constant around $.04 - .06$.

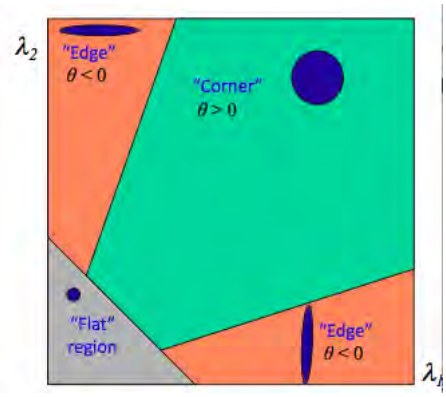


Figure 2: The visualization of theta thresholds from Corner Response Function (θ) indicating corner, edge, or flat regions. Source: Lecture 6 slides

This is not rotation invariant. To allow for rotation invariance, we will smooth with Gaussian, where the Gaussian already performs the weighted sum:

$$M = g(\sigma) \begin{bmatrix} I_x I_x & I_x I_y \\ I_x I_y & I_y I_y \end{bmatrix}$$

Illustrated below is an of keypoints identified by Harris detector.

Figure 3: An example of Harris keypoint detection on an image. Source: Lecture 6 slides

4 Scale Invariant Keypoint Detection

4.1 Motivation

Earlier we used the Harris detector to find keypoints or corners. This detector used small windows in order to maintain good locality. Since the Harris detector uses this small window, if an image is scaled up, the window now no longer sees the same gradients it had with a smaller image.

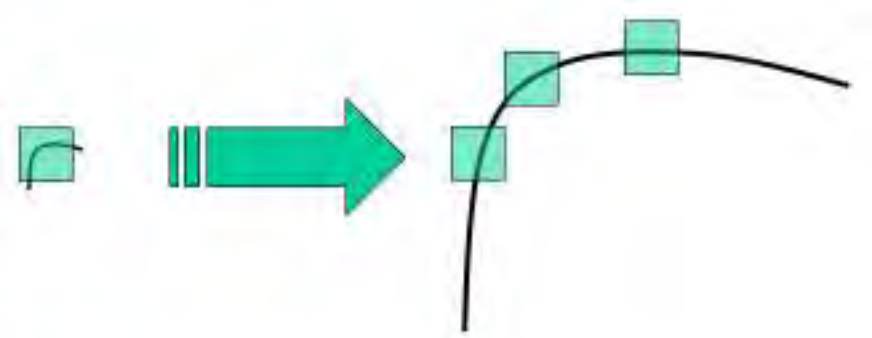


Figure 4: Harris detector windows on an increased scale. Source: [1]

Figure 4: Looking at the above image, we can see how the three windows on the right no longer see the sharp gradient in the x and y directions. All three windows are, therefore, classified edges. In order to address this problem, we need to normalize the scale of the detector.

4.2 Solution

We design a function to be scale-invariant, meaning that the function values for the corresponding regions are the same regardless of the scale. We can use a circle to represent this scalable function. A point on the circle is a function of the region size of the circle's radius.

4.3 General Approach

We can find the local maximum of a function. Relative to the local maximum, the region size should be the same regardless of the scale. This also means that the region size is co-variant with the image scale. A "good" function results in a single and distinct local maximum. In general, we should use a function that responds well to stark contrasts in intensity.

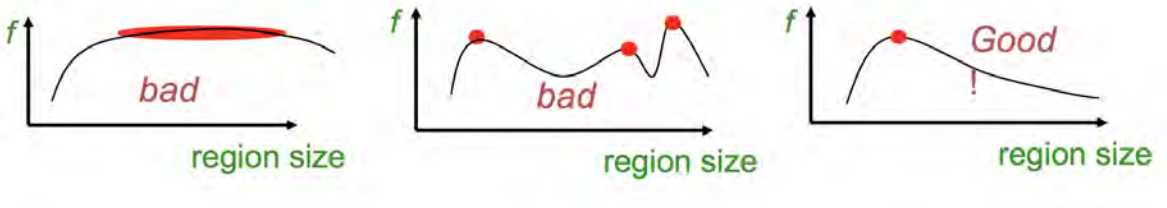


Figure 5: The examples of different functions for finding local maximums. Source: Lecture 7 slides.

The function is defined as: $f = kernel * image$. Two such kernels include the Laplacian and the Difference of Gaussians (DoG).

$$L = \sigma^2(G_{xx}(x, y, \sigma) + G_{yy}(x, y, \sigma))$$

$$DoG = G(x, y, k\sigma) - G(x, y, \sigma)$$

$$G(x, y, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Both these kernels are scale and rotation invariant.

References

[1] *OpenCV 3.1.0 Documentation:* $https://docs.opencv.org/3.1.0/$.

Lecture 8: Feature Descriptors and Resizing

Harrison Caruthers, Diego Celis, Claire Huang, Curtis Ogren, Junwon Park, Krithika Iyer
Department of Computer Science
Stanford University
Stanford, CA 94305
{hdcaruth, dcelis, chuang20, ceogren, junwonpk, ksiyer}@cs.stanford.edu

1 Scale invariant keypoint detection

1.1 Motivation

Thus far, we have covered the detection of keypoints in single images, but broader applications require such detections across similar images at vastly different scales. For example, we might want to search for pedestrians from the video feed of an autonomous vehicle without the prior knowledge of the pedestrians' sizes. Similarly, we might want to stitch a panorama using photos taken at different scales. In both cases, we need to independently detect the same keypoints at those different scales.

1.2 General methodology

Currently, we use windows (e.g., in Harris Corner Detection) to detect keypoints. Using identically sized windows will not enable the detection of the same keypoints across different-sized images (Figure 1). However, if the windows are appropriately scaled, the same content can be captured.

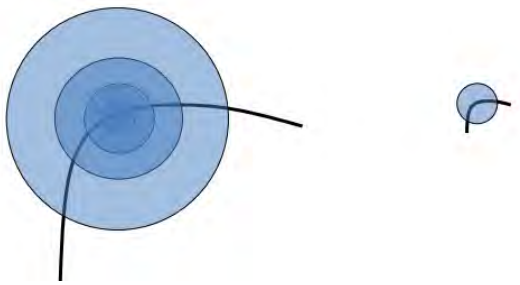


Figure 1: The corner of a curve appears at two scales. Note that the circular window on the right curve captures the entire corner, while the same-sized window on the left curve does not. Instead, we must choose a much larger circular window on the left curve to get the same information. Source: Lecture 7, slide 12.

How do we independently find the correctly scaled windows for each image? We need to describe what it means to "capture the same content" in a scale-invariant way. More specifically, consider a function, $f(\text{window})$, that takes in a region of content and outputs the same value for all scales of that region.

Now consider two similar images at different scales. We can independently vary window size for each of the images and plot the response of $f(\text{window})$ as a function of window size:

Within each of the two plots, we can independently identify local extrema as keypoints. The window sizes that correspond to those extrema (in the case of Figure 2, s_1 and s_2) provide us with the scale difference between the two images.

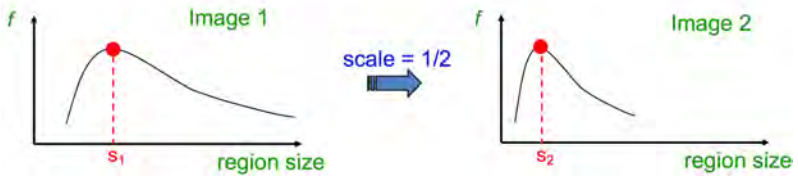


Figure 2: Two plots of the response of $f(\text{window})$ as a function of window size for Images 1 and 2, where Image 2 is similar to Image 1 but scaled by $\frac{1}{2}$. Source: Lecture 7, slide 15.

1.2.1 Average intensity

One candidate for such a function $f(\text{window})$ is the average intensity of the pixels within the window; this is because average intensity does not change as we scale the window up or down.

However, the average intensity is not great at capturing contrast or sharp changes within a window; this makes it harder to find clear extrema when comparing f across two images. To capture contrast, we need to bring in derivatives into the mix.

1.2.2 Difference of Gaussians

Another candidate would be to use the Difference of Gaussians method.

Consider an image I. First, the I is repeatedly convolved with Gaussian filters of different σ's. These convolutions are repeated with scaled down (i.e., down-sampled) versions of I. This results in the pyramid of Gaussians of different σ's and different image sizes (Figure 3). The adjacent Gaussian-convolved images are then subtracted to calculate their difference of Gaussians (DOG):

$$DOG(\sigma) = (G(k\sigma) - G(\sigma)) * I$$

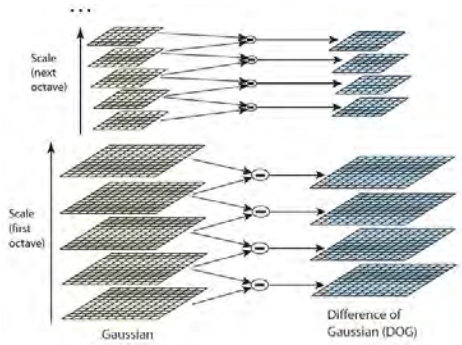


Figure 3: On the left: pyramid of Gaussians of different σ's and different image sizes. On the right: difference of adjacent Gaussians. Source: http://aishack.in/tutorials/sift-scale-invariant-feature-transform-log-approximation/

Intuitively, these differences of Gaussians capture details about I at different scales. More specifically, the difference of two Gaussians σ_1 and σ_2 remove all details that appear at both σ_1 and σ_2 and keep only those details that appear between σ_1 and σ_2. The differences of Gaussians for small and large σ's respectively capture fine and coarse details.

Given the differences of Gaussians pyramid in x-y-scale space, we can now identify local extrema within that $3D$ space to identify both keypoints and their associated scales. We compare a given coordinate against its 26 neighbors (in $3D$ space) and deem it an extrema if it is smaller or larger than all neighbors (see Figure 4).

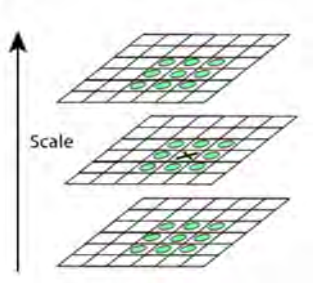


Figure 4: Given a coordinate in x-y-scale space (denoted by the black X), examine its 26 neighbors (denoted by the green circles) to determine if the original coordinate is a local extrema. Source: Lecture 7, slide 22

1.2.3 Harris-Laplacian

A third candidate is to use the Harris-Laplacian method [2], shown to be more effective at scale-invariant keypoint detection than the DoG but also potentially more computationally expensive.

Consider again an image I. First, we create multiple scales of I and run the Harris detector on each to localize keypoints per scale level. We then select the keypoints that maximize the Laplacian across all the scales.

Transition to scale-invariant descriptors: Now that we have several methods to *detect* consistent keypoints across multiple scales, we can move on to developing methods to *describe* those keypoints in a scale-invariant manner, so that they can be matched.

2 SIFT: an image region descriptor

2.1 Invariant Local Features

Point descriptor should be invariant and distinctive. To achieve robustness of point descriptors, we transform image content into local feature coordinates that are invariant to translation, rotation, scale, and other imaging parameters .

The advantages of invariant local features include:

- **Locality**: features describe parts and are robust to occlusion and clutter (no prior segmentation).
- **Distinctiveness**: features are identifiable from a large database of objects.
- **Quantity**: many features can be generated for even small objects.
- **Efficiency**: close to real-time performance.
- **Extensibility**: can easily be extended to wide range of differing feature types, each adding robustness to changes.

2.1.1 Scale invariance

- The only reasonable scale-space kernel is a Gaussian. (Koenderink, 1984; Lindeberg, 1994)
- An efficient choice is to detect peaks in the difference of Gaussian pyramid (Burt & Adelson, 1983; Crowley & Parker, 1984 - but examining more scales)
- Difference-of-Gaussian with constant ratio of scales is a close approximation to Lindeberg's scale-normalized Laplacian (can be shown from the heat diffusion equation)

2.1.2 Rotation invariance

Given a keypoint and its scale from DoG,

1. Smooth (blur) the image associated with the keypoint's scale.

2. Calculate the image gradients over the keypoint neighborhood.
3. Rotate the gradient directions and locations by negative keypoint orientation. In other words, describe all features relative to the orientation.

2.1.3 SIFT descriptor formation

Using precise gradient locations is fragile, so we want to produce a similar descriptor while allowing for generalization. We create an array of orientation histograms and place gradients into local orientation histograms of 8 orientation bins. Dividing gradients into 8 bins is recommended, and this number of bins was found to exhibit best performance through experimentation.

More concretely,

1. Create an array of orientation histograms
2. Put rotated gradients into local orientation histograms, where each gradient contributes to the nearby histograms based on distance; gradients far from center are scaled down. The SIFT authors [3] found that the best results were achieved using 8 orientation bins per histogram and a 4x4 histogram array (Figure 2).
3. Compare each vector between two images to find the matching keypoints.
4. To add robustness to illumination changes in high contrast photos, normalize the vectors before the comparison. This mitigates the unreliable 3D illumination effects such as glare that are caused by the very large image gradients; this is achieved by clamping the values in vector to under 0.2 (an experimentally tuned value) before normalizing again.

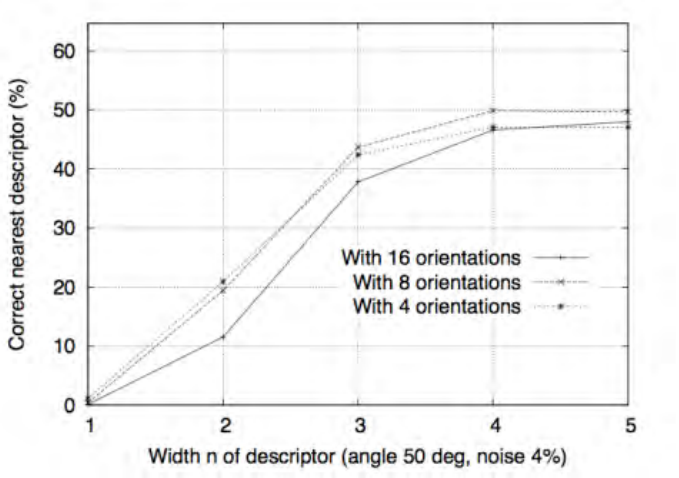


Figure 5: This figure shows the percentage of correctly matched keypoints as a function of the width of the descriptor and of the number of histogram bins. [1]

3 HoG: Another image region descriptor

3.1 Histogram of Oriented Gradients

The histogram of oriented gradients (HOG)[4] Descriptor finds an object within an image that pops-out, an object that can be discriminated. The general algorithm for HOG proceeds as follows:

1. Divide the image window into small spatial regions or cells.
2. For each cell, accumulate a local histogram; group gradient directions into evenly-spaced bins, and allocate the magnitude of a pixel's gradients into the appropriate bin corresponding to the gradient direction (Figure 6).
3. Normalize the local histograms over a larger region, called a "block", comprised of a number of cells.

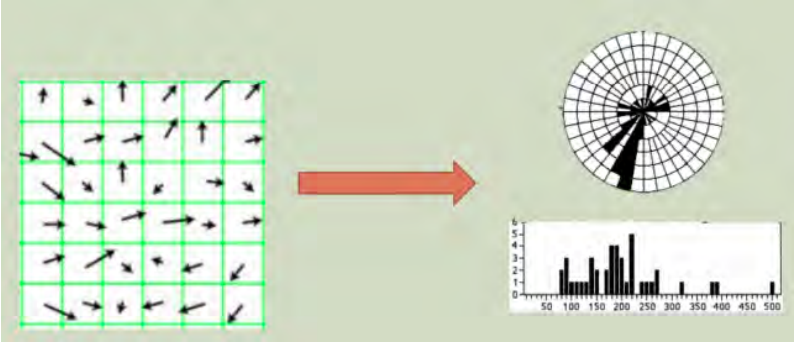


Figure 6: Here we see a visual example of keeping track of the magnitudes of the gradients for each gradient direction. Source: Lecture 7, Slide 60.

There are a few downsides to using HOG:

1. Large variations and ranges when detecting
2. Very slow
3. Not very organized when the backgrounds have different illuminations

Despite these downsides, HOG can be quite effective. Note the results of applying HOG in the image below.

Figure 7: HoG applied to a bicycle. Source: Lecture 7, Slide 65.

3.2 Difference between HoG and SIFT

There are some minor differences between the two. HOG is used over an entire image to find gradients. SIFT is used for key point matching. The SIFT histograms orient towards the natural positive gradient while HOG does not. HOG uses neighborhood bins, while SIFT uses weights to compute varying descriptors.

4 Image Stitching and Panorama Creation

4.1 Homogeneous Coordinates

Stitching together multiple images to create a panoramic view requires an understanding of projective geometry and homography. In addition to our familiar 3D Euclidean space, an additional dimension called W needs to be considered. W may be thought of as the distance between a projector and an image. The four dimensional space is called “projective space” and the coordinates in projective space are called “homogeneous coordinates”. Conversion between image coordinates and homogeneous coordinates is carried out as follows: (i) to convert to homogeneous coordinates,

$$(x, y) \Rightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

and (ii) from homogeneous coordinates to image coordinates

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \Rightarrow (\frac{x}{w}, \frac{y}{w})$$

When taking pictures with a camera, W may be thought as the distance from the camera to the objects in the picture. Objects near the camera appear larger (in the image) that objects that are farther from the camera. This phenomenon is known as "perspective". Images from different "perspective"s need to be transformed to a common perspective before they can be stitched together.

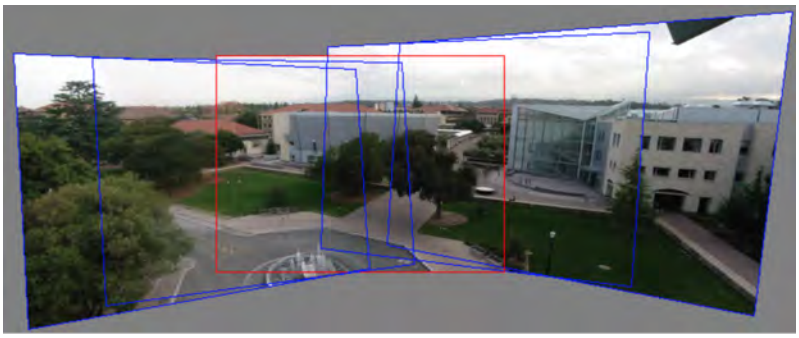


Figure 8: The different rectangles show images taken from different perspectives, but transformed to a common perspective plane through a keypoint matching and image transformation process. Source: $http://graphics.cs.cmu.edu/courses/15-463/2010fall/$

4.2 Transformations

The transformation of an image from one projective plane to another may involve translation, scaling (up or down), rotation, shear, and changes in aspect ratio. Such operations are illustrated below.

The matrices (in homogeneous coordinates) used to calculate such transformations from one perspective to another are shown below.

4.3 Homography

Homography matrix describes the transformation between points in one picture and a related neighboring picture (different perspective). In Figure 11, the identical points are denoted by their respective perspective based coordinates. Homography matrix H describes the transformation from one set of coordinates to the other.

The coordinates of the keypoint denoted by the red dot in both the images are related by the Homography matrix H. The point $p_1 = [x_1, y_1]$ on the left image (in Figure 11) matches with the point $p_2 = [x_1', y_1']$ on the right image, the transformation from one to the other is given by:

$$\begin{bmatrix} wx' \\ wy' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

It may be impossible to to find the transformation matrix H that transforms every point in image 2 to the corresponding point in another image. It is possible to estimate the transformation matrix H with least squares. Given N matched keypoint pairs, X_1 and X_2 (both $NX3$ matrices whose rows are homogeneous coordinates), H can be estimated by solving the least squares problem:

$$X_2 H = X_1$$

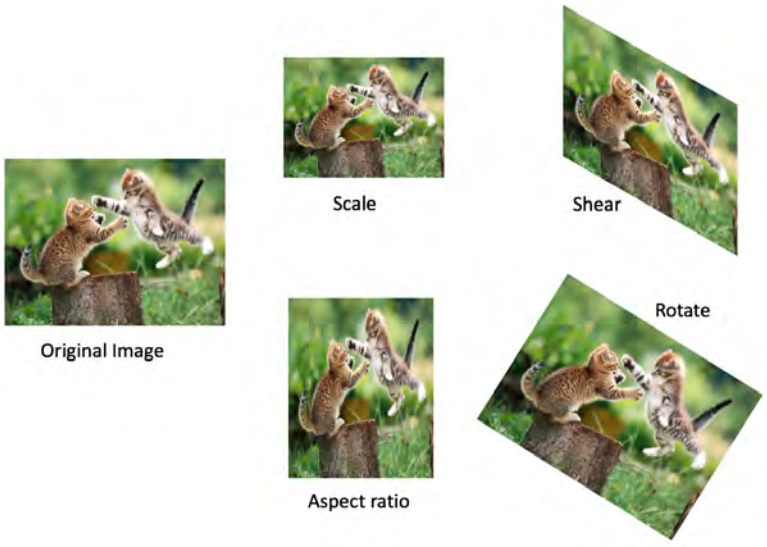


Figure 9: Basic transformation operations. Effect of 2D translation (which moves the image in x, y directions) is not shown here. Image Credit: S. Seitz and R. Collins. Penn State.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Translate

$$\begin{bmatrix} \boldsymbol{x}' \\ \boldsymbol{y}' \\ 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{s}_x & 0 & 0 \\ 0 & \boldsymbol{s}_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \\ 1 \end{bmatrix}$$

Scale

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\Theta & -\sin\Theta & 0 \\ \sin\Theta & \cos\Theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Rotate

$$\begin{bmatrix} \boldsymbol{x}' \\ \boldsymbol{y}' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & \boldsymbol{sh}_x & 0 \\ \boldsymbol{sh}_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{y} \\ 1 \end{bmatrix}$$

Shear

Figure 10: Matrices used to compute the new coordinates after basic transformations. Image Credit: Svetlana Lazebnik. UIUC.

4.4 RANSAC

During the least squares based estimation of the H matrix, it is a good practice to utilize RANSAC ("RANdom SAmple Consensus") to reduce the effect of outliers in the data. We avoid the impact of outliers by looking for inliers. The reasoning is that when an outlier is chosen to estimate the current fit, there will not be much support from the inliers for the current fit. Inliers are defined as points that are less than a threshold distance t, from the current fit.

The RANSAC steps are: (Source: Homework # 3)

1. Select a random set of matches.
2. Compute affine transform matrix.
3. Find the inliers using given threshold.
4. Repeat and find the largest set of inliers.
5. Re-compute least square estimates on all inliers.

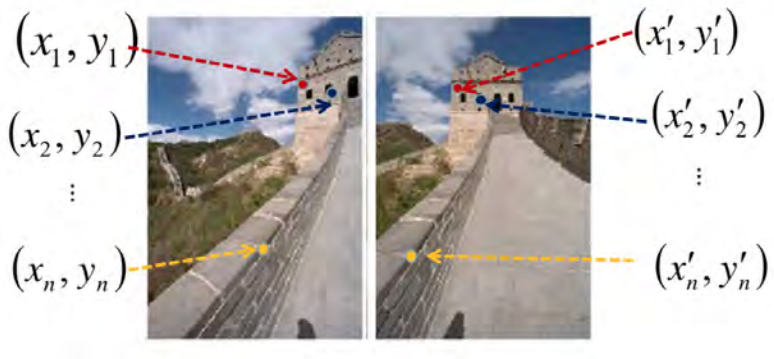


Figure 11: Identical points in different images and perspectives. Image Credit: Svetlana Lazebnik. UIUC.

5 Image resizing with seam carving

Because there are different screen sizes, we need to resize content according to display capabilities. Normally, we try to force content to fill up any kind of display by stretching or shrinking an image. However, this produces less than desirable results. So what is the solution? Content-aware re-targeting operators.

Retargeting means that we take an input and "re-target" to a different shape or size. Imagine input as being an image of size $n \times m$ and the desired output as an image of size $n' \times m'$. The idea behind retargeting is to

1. adhere to geometric constraints (e.g., aspect ratio),
2. preserve important content and structures, and
3. limit artifacts.

However, what is considered "important" is very subjective, for what may be important to one observer may not be important to another.

5.1 Pixel energy

A way to decide what is considered "important" is using **saliency measures**. There are many different types of saliency measures, but the concept is the same: each pixel p has a certain amount of "energy" that can be represented by the function $E(p)$.

The concept is that pixels with higher energy values are more salient, or more important, than pixels with lower energy values. What actually goes into the heart of E is up to the beholder.

A good example is to use the gradient magnitude of pixel p to heavily influence $E(p)$, for this usually indicates an edge. Since humans are particularly receptive to edges, this is a part of the image that is potentially valuable and interesting, compared to something that has a low gradient magnitude. As a result, this preserves strong contours and is overall simple enough to produce nice results. This example of E for image I could be represented as

$$E(\mathbf{I}) = \left|\frac{\partial \mathbf{I}}{\partial x}\right| + \left|\frac{\partial \mathbf{I}}{\partial y}\right|.$$

5.2 Seam carving

Let's assume that we have an input image with resolution $m \times n$ and we are looking for an output image $m \times n'$, where $n' < n$. How do we know what pixels to delete? We can use this concept of pixel energy to identify paths of adjacent pixels, or **seams**, that have the lowest combined pixel energy to remove from the image.

Note that the seams need not be strictly rows and columns. In fact, most of the time seams are curves that go through an image horizontally or vertically. A seam is horizontal if it reaches from the bottom edge to the top edge of an image. Similarly, a seam is vertical if it reaches from the left edge to the right edge of an image. However, seams are always laid out in a way such that there is only one pixel per row if the seam is vertical, or only one pixel per column if the seam is horizontal.

In essence, a seam avoids all important parts of an image when choosing what to remove from the image so as to cause the least disruption to the image when removed. There are more things to consider regarding seam carving and use cases that can be improved with similar techniques, but this is the core idea of how seams operate.

References

[1] David G. Lowe, "Distinctive image features from scale-invariant keypoints," International Journal of Computer Vision, 60, 2 (2004), pp. 91-110

[2] Mikolajczyk, Krystian. "Detection of Local Features Invariant to Affine Transformations." Perception Group, Institut National Polytechnique de Grenoble, INRIA Grenoble Rhône-Alpes, 2002.

[3] Lowe, David G. "Object recognition from local scale-invariant features." Computer vision, 1999. The proceedings of the seventh IEEE international conference on. Vol. 2. Ieee, 1999.

[4] Dalal, Navneet, and Bill Triggs. "Histograms of oriented gradients for human detection." Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on. Vol. 1. IEEE, 2005.

Lecture #9: Image Resizing and Segmentation

Mason Swofford, Rachel Gardner, Yue Zhang, Shawn Fenerin
Department of Computer Science
Stanford University
Stanford, CA 94305
{mswoff, rachel0, yzhang16, sfenerin}@cs.stanford.edu

1 Introduction

The devices used for displaying images and videos are of different sizes and shapes. The optimal image/video viewing configuration, therefore, varies across devices and screen sizes. For this reason, image resizing is of great importance in computer vision. The intuitive idea is to rescale or crop the original image to fit the new device, but that often causes artifacts or even loss of important content in images. This lecture discussed the techniques used for resizing images while preserving important content and limiting the artifacts.

1.1 Problem Statement

Input an image of size $n \times m$ and return an image of desired size $n' \times m'$ which will be a good representative of original image. The expectations are:

1. The new image should adhere to device geometric constraints.
2. The new image should preserve the important content and structures.
3. The new image should have limited artifacts.

1.2 Importance Measures

1. A function, $S : p \rightarrow [0, 1]$, is used to determine which parts in an image are important; then, different operators can be used to resize the image. One solution is to use an optimal cropping window to extract the most important contents, but this may result in the loss of important contents.

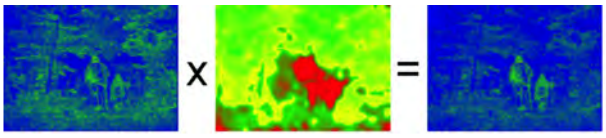


Figure 1: Importance Measurement by function method. Source: Lecture 7-11.

2. There are also more sophisticated techniques for measuring the image regions with higher degree of important content; they include, but are not limited to, attention models, eye tracking (gaze studies), and face detectors.

Figure 2: Importance Measurement by more sophisticated methods such as eye tracking. Source: Lecture 7-11, [5]

2 Seam Carving

2.1 Basic Idea

The human vision is more sensitive to the edges in an images. Thus, a simple but effective solution is to remove contents from smoother areas and preserve the more informative image regions that contain edges; this is achieved using a gradient-based energy function, defined as:

$$E(I) = |\frac{\partial}{\partial x}I| + |\frac{\partial}{\partial y}I|$$

Unimportant contents are, therefore, pixels with smaller values of energy function.

2.2 Pixel Removal

There exist different approaches for removing the unimportant pixels, and each can lead to different visual results. The figure below demonstrates three examples of such approaches; the first two (i.e., optimal least-energy pixel and row removal) are observed to negatively affect the image quality. The last one (i.e., least-energy column removal), on the other hand, works significantly better, but it still causes plenty of artifacts in the new image. An alternative solution is presented in the next section.

1. Removing all pixels with less energy.
2. Removing the rows of pixels with the least energy.
3. Removing the columns of pixels with the least energy.

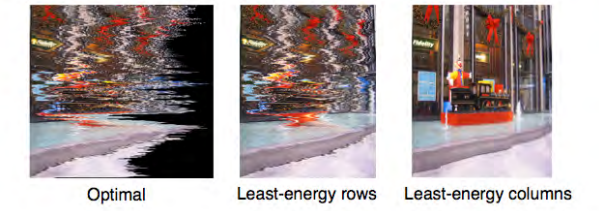


Figure 3: The effects of different pixel removal methods on the image quality. Source: Lecture 7-18

2.3 A Seam

1. A seam is defined as a connected path of pixels from top to bottom (or left to right). For top-to-bottom pixel, we shall pick exactly one pixel from each row. The mathematical definition is

$$s^x = \{s_i^x\}_{i=1}^n = \{x(i), i\}_{i=1}^n, s.t. \forall i, |x(i) - x(i-1)| \leq 1$$
$$s^y = \{s_j^y\}_{j=1}^m = \{j, y(j)\}_{j=1}^m, s.t. \forall j, |y(j) - y(j-1)| \leq 1$$

2. The optimal seam is the seam which minimizes the energy function, based on pixel gradients.

$$s^* = argmin_s E(s), \quad \text{where} \quad E(I) = |\frac{\partial}{\partial x}I| + |\frac{\partial}{\partial y}I|$$

Figure 4: The red line indicates the location of the optimal seam with the least energy. Source: Lecture 7-22

3. The recursion relation can be used to find the optimal seam. If $M(i,j)$ is defined as the minimal energy cost of a seam going through pixel (i,j), the recursion relation is

$$M(i,j) = E(i,j) + min(M(i-1,j-1), M(i-1,j), M(i-1,j+1))$$

This problem can be solved efficiently by using dynamic programming in $O(snm)$, $s = 3$ in the original algorithm. Given the energy function value as

5	8	12	3
4	2	3	9
7	3	4	2
5	5	7	8

Figure 5: An example of energy function used in seam carving algorithm. Source: Lecture 7-24

The recursion relation gives

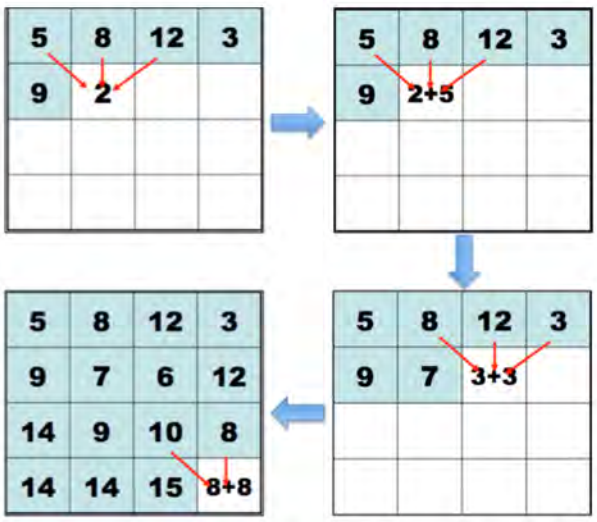


Figure 6: The process of using relation recursion for computing the seam cost. Source: Lecture 7-(24-27)

4. To search for the optimal seam, backtracking method is introduced. It starts from the pixel at the bottom with the lowest energy function, and it then goes up toward the top of the image (i.e., first image row).

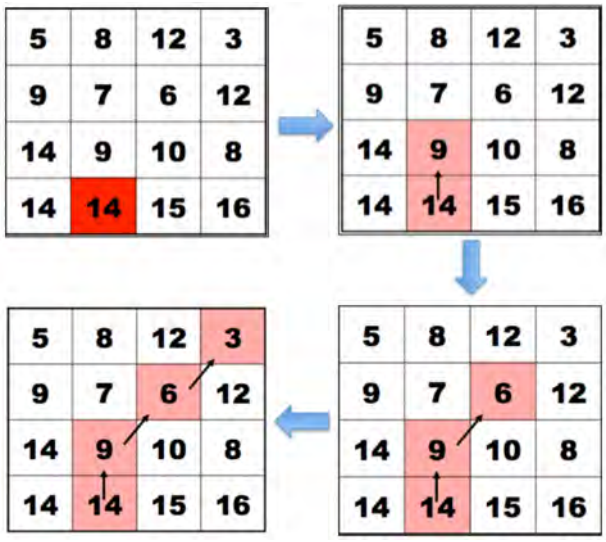


Figure 7: Using backtrack to find the optimal seam. Source: Lecture 7-(28-31)

2.4 Seam Carving Algorithms

This algorithm runs $O((n - n')mn)$. In each loop, each update of E, s and im takes $O(mn)$. For vertical resizing, the image could be transposed so that the same algorithm can be used.

Algorithm 1 Seam-Carving

$im \leftarrow$ *original image of size m* $\times$ *n*
$n' \leftarrow$ *desired image size n'*

Do (n-n') times:
    $E \leftarrow$ *Compute energy map on im*
    $s \leftarrow$ *Find optimal seam in E*
    $im \leftarrow$ *Remove s from im*

return im

The average energy of images increase given that the seam carving algorithm removes low energy pixels. The described seam carving algorithm can be used to modify aspect ratio, to achieve object removal, and to perform image resizing. The process is the same if an image is flipped. When resizing, both horizontal and vertical seams need to be removed. One can solve the order of adding and removing seams in both directions by dynamic programming. Specifically, the recurrence relation is: $T(r, c) = min(T(r - 1, c) + E(s^x(I_{n-r-1\times m-c})), T(r, c - 1) + E(s^y(I_{n-r\times m-c-1})))$ for more information refer to the SIGGRAPH paper on seam carving [8].

Figure 8: The Sea off Satta, Hiroshige woodblock print (Public Domain)

3 Advanced Seam Carving

3.1 Image Expansion

A similar approach can be employed to increase the size of images. By expanding the least important areas of the image (as indicated by our seams), the image dimensions can be increased without impacting the main content. A naive approach is to iteratively find and duplicate the lowest energy seam. However, this provides us results as depicted below:

Figure 9: Duplicating the least-energy seam is not a good strategy for image expansion [1]

On the right side of the image (Figure 9), one seam has been duplicated repeatedly. This is the program retrieves the same (least-energy) seam repeatedly. A more effective implementation is to find the first k seams at once and duplicate each:

Figure 10: A more effective image expansion strategy using the first k low-energy seams [2]

As observed above, the second image expansion strategy significantly improves the quality of resized image. Note, however, that this method can only enlarge the image by a factor of 2 (since there simply aren't enough seams to duplicate). For very dramatic enlargements, you can instead iteratively enlarge by 1.4-1.5x.

3.2 Multi-Size Image Representation

While we've seen that image resizing can be very effective, it is still very computationally intensive. In practice, images are stored alongside a representation of their seams to forgo seam re-calculation of the seams and streamline image resizing on devices. These seam representations are the same dimensions as the image, but instead of pixel intensities, they have numbered paths ordering seams based on their energy values (ranging from least-energy to highest-energy). In order to calculate these seams in the pre-processing step, the image energy must be recalculated after each seam removal (this is because of the changes in the cost function). See below for an example of such a representation:

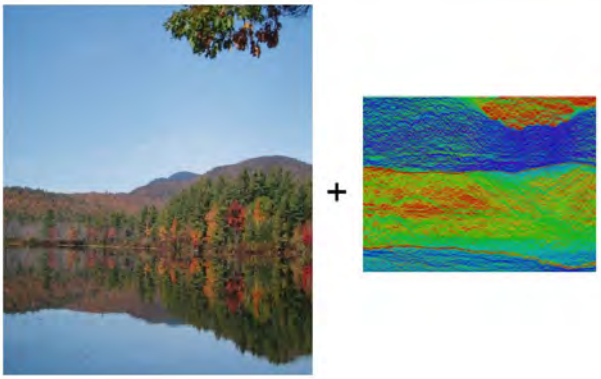


Figure 11: Examples of pre-computed seams used for fast image resizing on devices [8].

Given this representation, a program seeking to remove k seams from the original image can remove the pixels corresponding to those labeled 1 through k in the seam image (at right).

3.3 Object Removal

By allowing users to specify which areas of the image to give high or low energy, we can use seam carving to specifically preserve or remove certain objects. The algorithm chooses seams specifically so that they pass through the given object (in green below).

Figure 12: The seam carving algorithm can be used for object removal by assigning low energy values to part of the image [8].

3.4 Limitations

Seam carving is an effective method for image resizing. However, there are limitations: 1) the primary limitation is the lower and upper limit to effectiveness as the size of the image is drastically changed; 2) the failure in recognizing important features in the context of an object that can be low energy. Let us consider the image below.

Figure 13: The limitations of seam carving; source: CS131 Lecture 7, slide 71

While the flat and smooth image regions (i.e., with low gradients) are important to the image, they are removed; for example, this includes the woman's cheeks and forehead. While these regions are low in energy, they are important features to human perception and should be preserved. To address such limitations, the energy function can be modified to consider additional information. For example, a face detector can be used to identify important contents (i.e., human faces) or other constraints can be applied by the users.

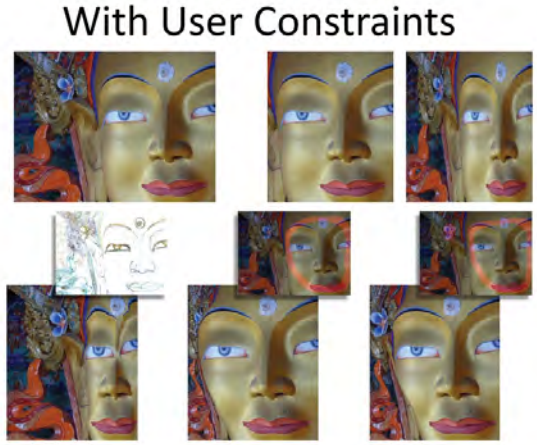


Figure 14: Seam Carving for Content-Aware Image Resizing [4].

3.5 Forward Energy

When we carve seams, we are removing the lowest energy pixels and preserving the highest energy pixels. Hence, the average image energy is increased and can lead to artifacts and jagged edges. One way to avoid this issue is to instead focus on removing seams that insert the least energy into the image. This approach is known as the forward energy; our original accumulated cost matrix is modified by adding the forward energy from corresponding new neighbors as seen in the image below. The originally introduced method is referred to as "backward" energy.

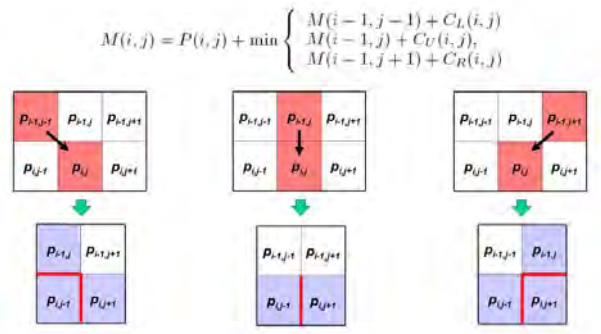


Figure 15: The forward energy calculations; source: Lecture 7-71

Figure 15 gives us 3 cases of removing seam containing pixel $p(i, j)$. Calculating the three possible vertical seam step costs for pixel $p(i, j)$ using forward energy. After removing the seam, new neighbors (in gray) and new pixel edges (in red) are created. In each case the cost is defined by the forward difference in the newly created pixel edges. This is because at each step, we search for the seam whose removal inserts the minimal amount of energy into the image. These are seams that are not necessarily minimal in their energy, but will leave less artifacts in the resulting image, after removal. Note that the new edges created in row $i - 1$ were accounted for in the cost of the previous row pixel.

Thus we have

$$C_L(i, j) = \|I(i, j + 1) - I(i, j - 1)\| + \|I(i - 1, j) + I(i, j - 1)\|$$

$$C_R(i, j) = \|I(i, j + 1) - I(i, j - 1)\| + \|I(i - 1, j) + I(i, j + 1)\|$$

$$C_U(i, j) = \|I(i, j + 1) - I(i, j - 1)\|$$

.

The figure below compares the performance of backward and forward computations for image resizing. The traditional "backward" energy approach resulted in jagged edges appearing along the handle of the wrench and the stem of the plant. The "forward" energy approach, on the other hand, minimizes the added energy, and the smoothness of the edges is maintained.

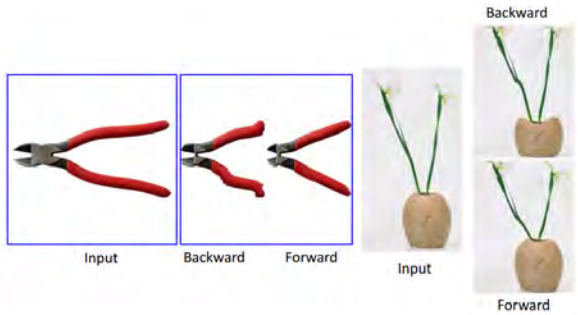


Figure 16: Forward energy approach for content-aware image resizing [4].

3.6 Seam-Carving in Videos

While the powerful capabilities of seam carving is explored, the question remains as to how it can be applied to videos. This algorithm faces challenges with respect to video resizing. Videos are significantly more difficult to resize. We face two primary issues.

First, let us consider a one-minute window of a film recorded at 30fps. In that duration, we have 1800 frames. If our seam carving algorithm takes a full minute to process an image, it would take us 30 hours to completely process the video.

The second issue is with temporal coherency. One can consider the intuitive and naive approach of seam carving to process each frame independently. However, this does not necessarily preserve the important content with respect to the relation of consecutive frames. Since the human eye is particularly sensitive to movement, the failure to consider the context across images can create a poor looking video with noticeable distortion from frame to frame. There is no coherency to changes across frames. A more effective approach is to consider the video as a three dimensional space where each vertical plane is an image from the video. The lowest energy 2D seam can be calculated throughout the entire video's length. This produces much better results, but it still faces limitations.

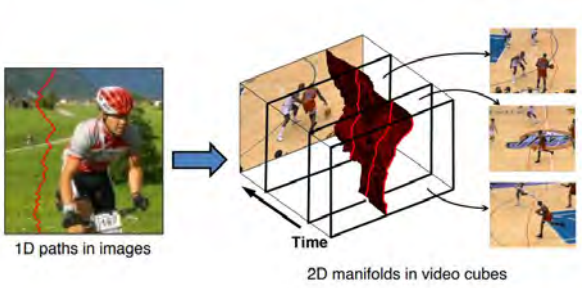


Figure 17: Improved Seam Carving for Video Retargeting [7].

This 3D representation gives us the same capabilities as 2D Seam-Carving such as object removal and frame resizing.

4 Segmentation

In computer vision, we are often interested to identify groups of pixels that go together. We call this problem image segmentation. Humans perform image segmentation by intuition. For instance, two people looking at the same optical illusion might see different things, all depending on how their brain segments the images. In the image below, you might see zebras, or you might see a lion.

Figure 18: Optical illusions regarding the problem of image segmentation [3].

One of the motivations behind image segmentation is the separation of an image into coherent objects. Here are two examples of this type of segmentation:

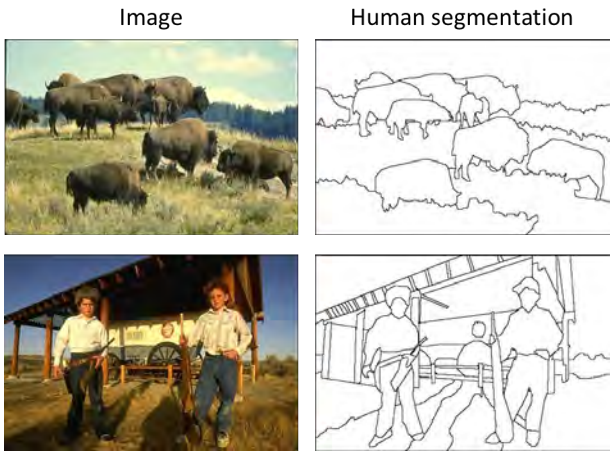


Figure 19: Human segmentation of example images; source: Svetlana Lazebnik

We might also want to segment an image into many groups based off nearby pixels being similar. We call these groups "superpixels." Superpixels allow us to treat many individual pixels as one cluster, and therefore enable faster computations. Here is an example of an image segmented by superpixels:

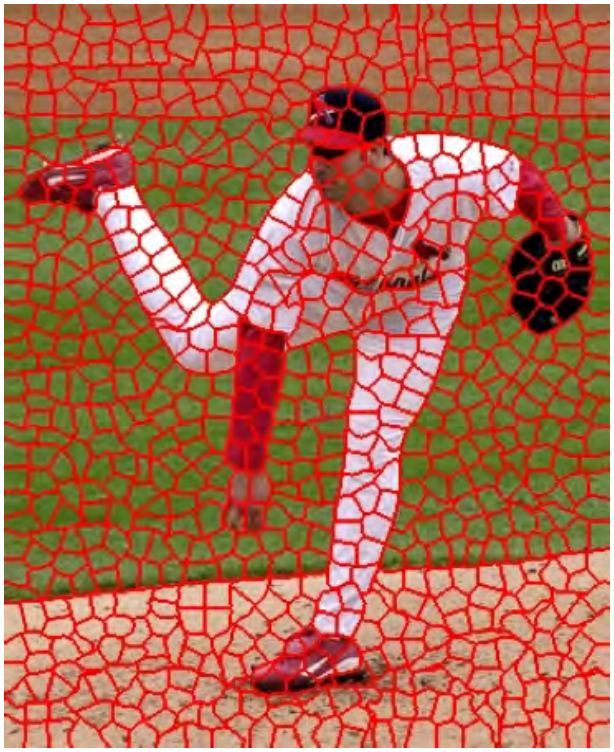


Figure 20: The superpixels allow faster computations by clustering pixels [6].

Superpixel segmentation and other forms of segmentation can help in feature support. We can treat the groups of pixels as one feature and garner information about the image from them.Image segmentation is also beneficial for some common photo effects such as background removal. If we can properly segment an image, we will be able to only keep the groups we want and remove the others.

While segmentation is clearly useful and has many practical applications, there is no one way to segment an image, and we must compare different segmentation algorithms to find our optimal solution. The images are prone to under-segmentation and over-segmentation if they respectively have very few or an excessive number of groups. However, even a properly segmented photo can have multiple different possible groupings.

To tackle the problem of how to segment an image, we will think of segmentation as clustering. By clustering, we are able to group similar data points together and represent them with one singular value. This again aids in our ability to manipulate the image or extract features from it. However, we must decide a few important issues:

1. How do we determine if two pixels, patches, or images are similar?

2. How do we compute an overall grouping from pairwise similarities?

Clustering algorithms have different answers for these questions; they will be discussed in depth in the next notes.

In general, the two broad categories of clustering algorithms are top down and bottom up. The top-down clustering approach groups pixels and patches together if they lie on the same visual entity. The bottom-up approach groups pixels together if they are locally coherent.

We may also use certain human-recognizable visual patterns for our clustering algorithm. Some example patterns include grouping similar objects together or using symmetry to aid in segmentation. In some instances, we can also look at "common fate." Common fate means that a group of objects appear to be moving together, so they share the same "fate." Here is an example of camels, which we can group by their common fate.

Figure 21: Common fate provides visual cues for the segmentation problem; source: Arthus-Bertrand (via F. Durand)

We can also illustrate common fate with this optical illusion. This illusion, called the Müller-Lyer illusion, tricks us into thinking the bottom line segment is longer than the top line segment, even though they are actually the same length (disregarding the four mini-tails).

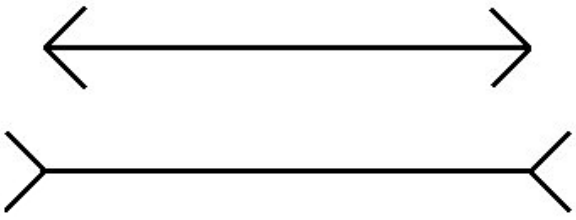


Figure 22: Common fate results in an optical illusion; source: Simon Barthelme

Another way we can group objects is by proximity. With proximity, we group objects with what they appear to be close to. For instance, in this image, we might group the three people in the foreground together.

Figure 23: Proximity can aid the image segmentation; source: Kristen Grauman

References

[1] https://commons.wikimedia.org/wiki/file:broadway_tower_edit.jpg.

[2] https://commons.wikimedia.org/wiki/file:broadway_tower_edit.jpg.

[3] https://www.weirdoptics.com/hidden-lion-visual-optical-illusion.

[4] Shai Avidan and Ariel Shamir. Seam carving for content-aware image resizing. In *ACM Transactions on graphics (TOG)*, volume 26, page 10. ACM, 2007.

[5] Tilke Judd, Krista Ehinger, Frédo Durand, and Antonio Torralba. Learning to predict where humans look. In *Computer Vision, 2009 IEEE 12th international conference on*, pages 2106–2113. IEEE, 2009.

[6] Xiaofeng Ren and Jitendra Malik. Learning a classification model for segmentation. In *null*, page 10. IEEE, 2003.

[7] Michael Rubinstein, Ariel Shamir, and Shai Avidan. Improved seam carving for video retargeting. In *ACM transactions on graphics (TOG)*, volume 27, page 16. ACM, 2008.

[8] Ariel Shamir Shai Avidan. Seam carving for content-aware image resizing. *ACM Trans. Graph*, 26(3):10, 2007.

Lecture 10: Semantic Segmentation and Clustering

Vineet Kosaraju, Davy Ragland, Adrien Truong, Effie Nehoran, Maneekwan Toyungyernsub
Department of Computer Science
Stanford University
Stanford, CA 94305
{vineetk, dragland, aqtruong, effie, maneekwt}@cs.stanford.edu

1 Clustering and Segmentation

Image segmentation is a task in computer vision; it aims to identify groups of pixels and image regions that are similar and belong together. Different similarity measures can be used for grouping pixels; this includes texture and color features. An instance of image segmentation is illustrated below. In Figure 1, the objective is to group all the pixels that make up the tiger, the grass, the sky, and the sand. The resulting segmentation can be observed in Figure 2.

Figure 1: Input image. Source: lecture 12, slide 4

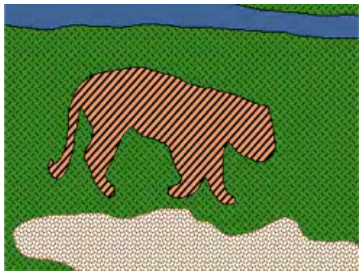


Figure 2: Output segmentation. Source: lecture 12, slide 4

Other examples of image segmentation include grouping video frames into shots and separating an image's foreground and background. By identifying the groups of pixels that belong together, an image can be broken down into distinct objects. In addition to its immediate use for object recognition, this can also allow for greater efficiency in any further processing.

2 Gestalt School and Factors

Many computer vision algorithms draw from the areas outside of computer science, and image segmentation is no different. Computer vision researchers drew inspiration from the field of psychology, specifically the Gestalt Theory of Visual Perception. At a very high level, this theory states that "the whole is greater than the sum of its parts." The relationships between parts can yield new properties and features. This theory defines Gestalt factors, properties that can define groups in images. Below are examples of such factors.

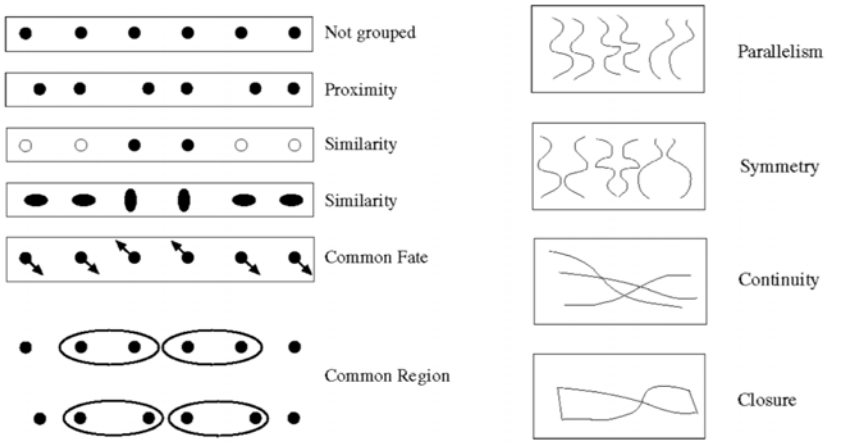


Figure 3: Examples of gestalt factors. Source: Forsyth & Ponce [1]

Here is an example of an other factor; while Figure 4 does not appear to have meaningful visual content, the addition of the overlaying gray lines (Figure 5) on the same image provides visual cues as to the pixel groupings and image content.

Figure 4: Source: Forsyth & Ponce [1]

Figure 5: Source: Forsyth & Ponce [1]

We can now more clearly see that the image is a few 9's occluded by the gray lines. This is an example of a continuity through occlusion cue. The gray lines give us a cue that the black pixels are not separate and should in fact be grouped together. By grouping the black pixels together and not perceiving them as separate objects, our brain is able to recognize that this picture contains a few digits.

Below is another example:

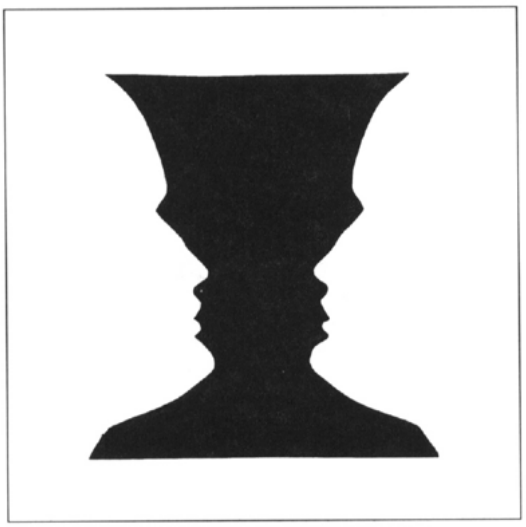


Figure 6: Source: Forsyth & Ponce [1]

What do you see? Do you see a cup or do you see the side of 2 heads facing each other? Either option is correct, depending on your perspective. This variation in perception is due to what we identify as the foreground and the background. If we identify the black pixels as the foreground, we see a cup. But, if we identify the white pixels as the foreground, we see 2 faces.

This is an overview of some of the factors that affect human's perception of pixel/image region grouping within visual data. One thing to note, however, is that while these factors may be intuitive, it's hard to translate these factors into working algorithms.

How do we perform image segmentation? What algorithms do we use? One way to think about segmentation is clustering. We have a few pixels and we want to assign each to a cluster. In the following sections, different methods of clustering will be detailed.

3 Agglomerative Clustering

Clustering is an unsupervised learning technique where several data points, $x_1, ..., x_n$, each of which are in R^D, are grouped together into clusters without knowing the correct label/cluster. **Agglomerative clustering** is one of the commonly used techniques for clustering.

The general idea behind agglomerative clustering is to look at similarities between points to decide how these points should be grouped in a sensible manner. Before we discuss the details of the algorithm, we must first decide how we determine similarity.

3.1 Distance Measures

We measure the similarity between objects by determining the distance between them: the smaller the distance, the higher the degree of similarity. There are several possible distance functions, but it is hard to determine what makes a good distance metric, so usually the focus is placed on standard, well-researched distance metrics such as the two detailed below.

3.1.1 Euclidean Distance

One common measure of similarity is the Euclidean distance; it measures the distances between two data points, x and x' by taking into account the angle and the magnitude. We can write the Euclidean distance as the following:

$$sim(x, x') = x^T x' \tag{1}$$

This distance measure does not normalize the vectors, so their magnitude is factored into the similarity calculations.

3.1.2 Cosine Similarity Measure

Another common solution is the Cosine similarity measure; it only accounts for the angle between two data points, x and x'. Note that **unlike the Euclidean distance**, the cosine measure only represents *similarity*, not *distance*. This means that the similarity between a data point x, and itself, equals 1. As its name implies, this measure relies on the cosine between the two points, as found by:

$$sim(x, x') = cos(\theta) \tag{2}$$

$$= \frac{x^T x'}{||x|| \cdot ||x'||} \tag{3}$$

$$= \frac{x^T x'}{\sqrt{x^T x}\sqrt{x'^T x'}} \tag{4}$$

The division by the magnitudes of the vectors results in the normalization of the distance metric, and it ensure that the measure is only dependent on the angle between the objects.

3.2 Desirable Clustering Properties

Now that the potential distance metrics are defined, the next step is to choose a clustering technique. There are various properties of clustering methods that we might want to consider when choosing specific techniques:

1. **Scalable** - in terms of compute power & memory
2. **Different data types** - algorithm should support arbitrary data being in R^d for all d
3. **Input parameters** - The parameter tuning for the algorithm should not be difficult. The algorithm is more useful if it does not heavily rely on our accurate understanding of the data.
4. **Interpretable** - we should be able to interpret the results.
5. **Constraints** - The algorithm should effectively use the predefined constraints (e.g., we know two points should be in the same cluster, or they shouldn't belong together).

The following sections cover the implementation of the agglomerative clustering and its benefits and drawbacks.

3.3 Agglomerative Clustering Implementation

The agglomerative clustering calculates the similarities among data points by grouping closer points together. The newly created groups can further be merged to other groups that are close to them; this iterative process results in the generation of bigger groups until there only remains one group. This creates a hierarchy, best viewed as a **dendrogram**.

We can visualize this in the following diagram, which shows data points and the results of an agglomerative clustering algorithm.

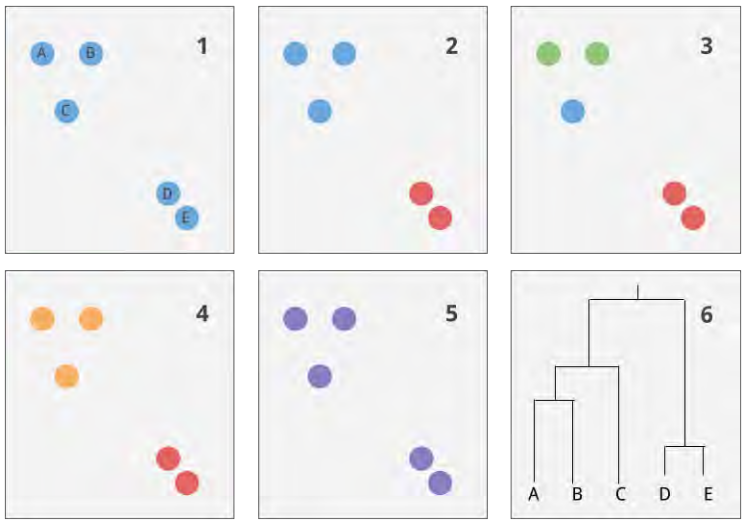


Figure 7: Agglomerative clustering on sample input, and resulting dendrogram

The first picture shows all the data points, and pictures 2 to 5 show various steps in the clustering algorithm. Step 2 groups the two red points together; step 3 groups the two green points together; step 4 groups the previous green group and the nearby blue point into a new orange group; and step 5 groups all the points together into one large group. This creates the dendrogram in picture 6.

3.3.1 Algorithm

Our general algorithm is based on our intuition and has four main steps:

1. Initialize each point as its own cluster
2. Find the most similar pair of clusters
3. Merge the similar pair of clusters into a *parent* cluster
4. Repeat steps 2 & 3 until we have only 1 cluster.

3.3.2 Questions

Although agglomerative clustering is a powerful technique, various factors should be taken into consideration when implementing it. For instance:

1. *How do we define similarity between clusters? How do we measure the distance between two clusters?*
   We can measure the distance between two clusters in a variety of different ways including the average distance between points, the minimum distance between points in the clusters, the distance between the means of each cluster, and the maximum distance between points in the clusters. The method used for measuring the distance between clusters can highly affect the results.
2. *How many clusters do we chose?*
   When we create the dendrogram, we can decide how many clusters we want based on a distance threshold. Alternatively, we can cut the dendrogram horizontally at its different levels to create as many clusters as we want.

3.4 Different measures of nearest clusters

There are three main models we can use to determine the distance between cluster points as we segment our dataset: single, complete, and average.

1. Single link:
   Distance is calculated with the formula:
   $$d(C_i, C_j) = min_{x \in C_i, x' \in C_j} d(x, x') \tag{5}$$

With single linkage, we cluster by utilizing the minimum distance between points in the two clusters.
This is also known as a minimum spanning tree.
We can stop clustering once we pass a threshold for the acceptable distance between clusters.
This algorithm tends to produces long, skinny clusters (since it is very easy to link far away points to be in the same cluster as we only care about the point in that cluster with the minimum distance).

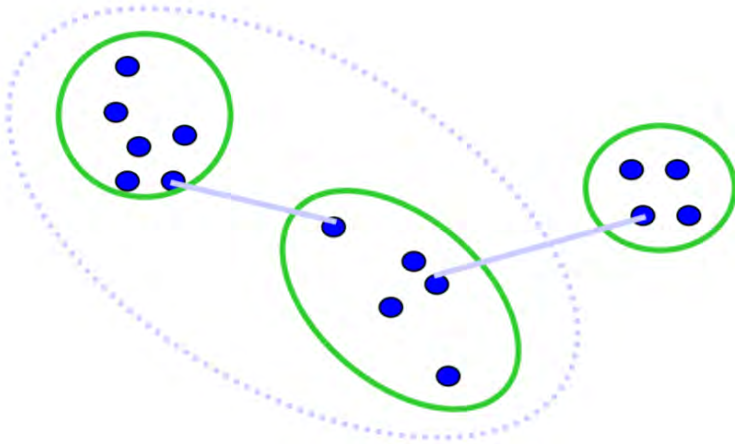


Figure 8: Image segmentation example using single link measurement of nearest clusters.
Source: lecture 12, slide 46

2. Complete link:
   Distance is calculated with the formula:

$$d(C_i, C_j) = max_{x \in C_i, x' \in C_j} d(x, x') \tag{6}$$

   With complete linkage, we cluster by utilizing the maximum distance between points in the two clusters.
   This algorithm tends to produces compact and tight clusters of roughly equal diameter (since it favors having all the points close together).

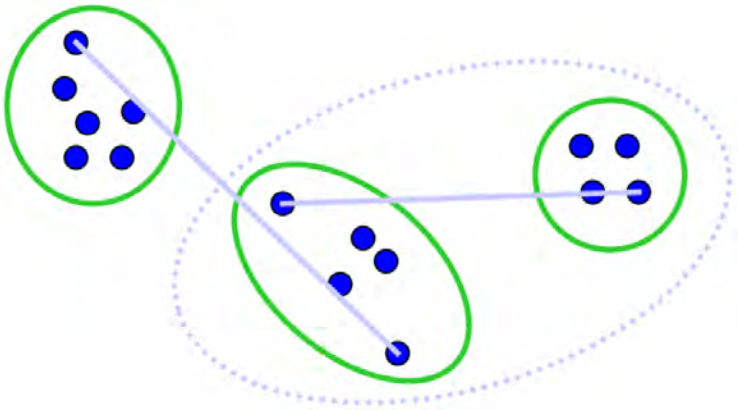


Figure 9: Image segmentation example using complete link measurement of nearest clusters.
Source: lecture 12, slide 47

3. Average link:
   Distance is calculated with the formula:

$$d(C_i, C_j) = \frac{\sum x \in C_i, x' \in C_j d(x, x')}{|C_i| \cdot |C_j|} \tag{7}$$

   With average linkage, we cluster by utilizing the average distance between points in the two clusters.

This model is robust against noise because the distance does not depend on single pair of points unlike single links and complete links, where points can be affected by artifacts in the data.

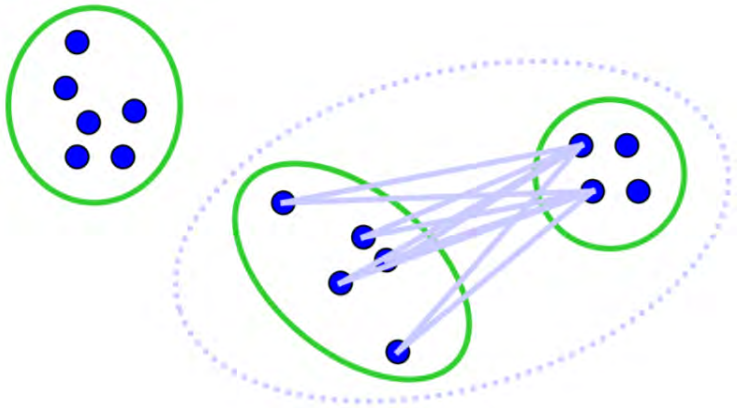


Figure 10: Image segmentation example using average link measurement of nearest clusters.
Source: lecture 12, slide 48

3.5 Agglomerative clustering conclusions

The positive characteristics of agglomerative clustering:

1. Simple to implement and apply
2. Cluster shape adapts to dataset
3. Results in a hierarchy of clusters
4. No need to specify number of clusters at initialization

In terms of its negative characteristics:

1. Can return imbalanced clusters
2. Threshold value for number of clusters must be specified
3. Does not scale well with a runtime of $O(n^3)$
4. Greedy merging can get stuck at local minima

4 K-Means Clustering

Another algorithm is k-means clustering. It identifies a fixed number of cluster "centers" as representatives of their clusters, and labels each point according to the center it is closest to. A major difference between k-means and agglomerative clustering is that k-means requires the input of a target number of clusters to run the algorithm.

4.1 Image Segmentation Example

At the top left of figure 11, we have an image with three distinct color regions, so segmenting the image using color intensity can be achieved by assigning each color intensity, shown on the top right, to a different cluster. In the bottom left image, however, the image is cluttered with noise. To segment the image, we can use k-means.

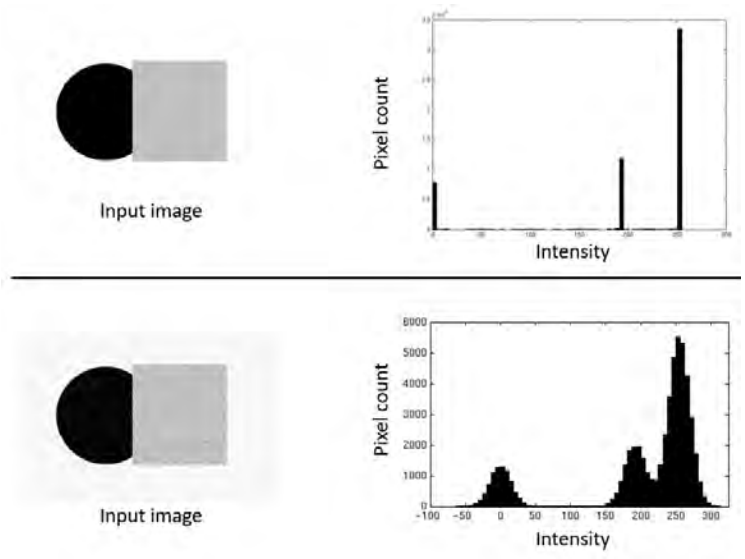


Figure 11: Image segmentation example using k-means. The picture on the top left has three distinct colors, but the bottom picture has Gaussian noise. Source: lecture 11, slide 8, slide credit: Kristen Grauman

Using k-means, the objective here is to identify three cluster centers as the representative intensities, and label each pixel according to its closest center. The best cluster centers are those that minimize Sum of Square Distance between all points and their nearest cluster center c_i:

$$SSD = \sum_{i \in \text{clusters}} \sum_{x \in \text{cluster}_i} (x - c_i)^2 \tag{8}$$

When we are using k-means to summarize a dataset, the goal is to minimize the variance in the data points that are assigned to each cluster. We would like to preserve as much information as possible given a certain number of clusters. This is demonstrated by the equation below.

$$c^*, \ \delta^* = \arg\min_{c,\delta} \frac{1}{N} \sum_{j}^{N} \sum_{i}^{K} \delta_{ij} \left(c_i - x_j \right)^2$$

Cluster center, Data, Whether x_j is assigned to c_i

Figure 12: Clustering for summarization. Source: lecture 11, slide 11

4.2 Algorithm

Finding the cluster centers and group memberships of points can be thought of as a "chicken and egg" problem. If we knew the cluster centers, we could allocate points to groups by assigning each point to the closest center. On the other hand, if we knew the group memberships, we could find the centers by computing the mean of each group. Therefore, we alternate between the tasks.

In order to find the centers and group memberships, we start by initializing k cluster centers, usually by assigning them randomly. We then run through an iterative process that computes group memberships and cluster centers for a certain number of iterations or until the values of the cluster centers converge. The process is outlined below:

1. Initialize cluster centers $c_1, ..., c_K$. Usually, these centers are randomly chosen data points.
2. Assign each point in the dataset to the closest center. As in agglomerative clustering, we can use the Euclidean distance or the cosine distance measure to compute the distance to each center.

3. Update the cluster centers to be the mean of the points in the cluster's group.

4. Repeat Steps 2-3 until the value of the cluster centers stops changing or the algorithm has reached the maximum number of iterations.

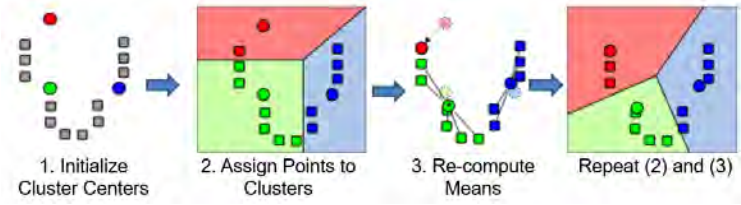


Figure 13: Visualization of k-means clustering. Source: lecture 11, slide 15

4.3 Output

Each time it is run, k-means converges to a local minimum solution. Additionally, since the centers are initialized randomly, each run of the algorithm may return a different result. Therefore, initializing multiple runs of k-means and then choosing the most representative clustering yields the best results. The best clustering can be measured by minimizing the sum of the square distances to the centers or the variance of each cluster. K-means works best with spherical data.

4.4 Segmentation as Clustering

As with the example in section 4.1, clustering can be used to segment an image. While color intensity alone can be effective in situations like Figure 11, others such as Figure 14 may require us to define a feature space, in which we choose which features of pixels should be used as input to the clustering. In other words, the choice of feature space directly affects the calculation of the similarity measure between data points; the creative choice of feature space enables us to "represent" the data points in a way that clusters are more easily distinguishable from each other.

Figure 14: Image of a panda. Source: lecture 11, slide 19

In addition to pixel intensity, examples of pixel groupings using an image feature space include RGB color similarities, texture similarities, and pixel positions. Clustering based on color similarities can be modeled using separate features for red, green, and blue. Texture can be measured by the similarities of pixels after applying specific filters. Position features include the coordinates of pixels within an image. Both intensity and position can be used together to group pixels based on similarity and proximity.

4.5 K-Means++

K-means method is appealing due to its speed and simplicity but not its accuracy. By augmentation with a variant on choosing the initial seeds for the k-means clustering problems, arbitrarily bad clusterings that are sometimes a result of k-means clustering may be avoided. The algorithm for choosing the initial seeds for k-means++ is outlined as following:

1. Randomly choose a starting center from the data points
2. Compute a distance $D(X)$, which is a distance between each data point x to the center that has been chosen. By using a weighted probability distribution, a new data point is chosen as a new center based on a probability that is proportional to $D(x)^2$
3. Repeat the previous step until k centers have been chosen and then proceed with the usual k-means clustering process as the initial seeds have been selected

It has been shown that k-means++ is Olog(K) competitive.

4.6 Evaluation of clusters

The clustering results can be evaluated in various ways. For example, there is an $internal$ evaluation measure, which involves giving a single quality score the results. $External$ evaluation, on the other hand, compares the clustering results to an existing true classification. More qualitatively, we can evaluate the results of clustering based on its $generative$ measure: how well is the reconstruction of points from the clusters or is the center of the cluster a good representation of the data. Another evaluation method is a $discriminative$ method where we evaluate how well clusters correspond to the labels. We check if the clusters are able to separate things that should be separated. This measure can only be worked with supervised learning as there are no labels associated with unsupervised learning.

4.7 Pros & Cons

There are advantages and disadvantages associated with k-means clustering technique:

Pros

1. Simple and easy to implement
2. Fast for low-dimensional data
3. Good representation of the data (cluster centers minimize the conditional variance)

Cons

1. Doesn't identify outliers
2. Need to specify k value, which is unknown
3. Cannot handle non-globular data of different sizes and densities
4. Restricted to data which has the notion of a center
5. Converge to a local minimum of the objective function instead and is not guaranteed to converge to the global minimum of the objective function

In order to choose the number of clusters or the k value, the objective function can be plotted against different k values. The abrupt change in the objective function at a certain k value is suggestive of that specific number of clusters in the data. This technique is called "knee-finding" or "elbow-finding"

5 Mean-shift Clustering

Mean-shift clustering is yet another clustering algorithm. At its essence, mean-shift clustering is about finding the densest areas in our feature space. This algorithm has four main steps:

1. Initialize random seed, and window W

2. Calculate center of gravity (the "mean) of W: $\sum_{x \in W} xH(x)$
3. Shift the search window to the mean
4. Repeat step 2 until convergence

One way to mentally visualize this algorithm is to picture each data point as a marble. If each marble is attracted to areas of high density, all the marbles will eventually converge onto 1 or more centers.

We can also try to visualize the algorithm via this picture:

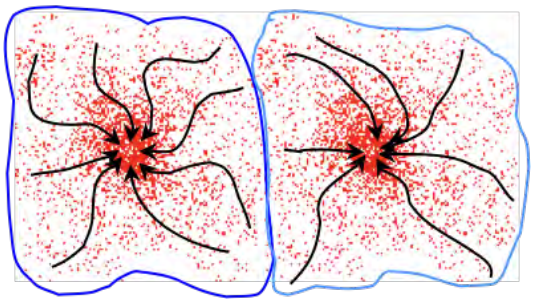


Figure 15: Results of mean-shift clustering.
Source: Y. Ukrainitz & B. Sarel

In this picture, we see the algorithm will generate 2 clusters. All the data points on the left converge onto one center and all the data points on the right converge onto a different center.

To learn more about mean-shift clustering please refer to the next set of notes.

References

[1] David Forsyth and Jean Ponce. *Computer vision: a modern approach*. Upper Saddle River, NJ; London: Prentice Hall, 2011.

Lecture #11: OBJECT RECOGNITION

Darrith Bin Phan, Zahra Abdullah, Kevin Culberg, Caitlin Go
Department of Computer Science
Stanford University
Stanford, CA 94305
{darrithp, zahraab, kculberg, cgo2}@cs.stanford.edu

1 Mean-Shift

Previously, we discussed the applications of agglomerative clustering and k-means algorithm for image segmentation. This lecture focuses on a mean shift, a mode-seeking technique that analyzes a density function to identify its maxima; one of the primary application of mean-shift algorithm is image segmentation. In the following sections, the different steps involved in the mean-shift algorithm are detailed.

1.1 Optimizations

To correctly assign points to clusters, a window must be initialized at each point and shifted to the most dense area. This procedure can result in a large number of redundant or very similar computations. It is possible to improve the speed of the algorithm by computing window shifts in parallel or by reducing the number of windows that must be shifted over the data; this is achieved using a method called "basin of attraction".

Parallelization The computations required to shift different windows are independent of each other and can be split across multiple processors and computed simultaneously. By parallelizing mean-shift over multiple processors or machines, it is possible to achieve a large speed increase without any loss to accuracy.

Basin of Attraction It is very likely that points close to the path and stopping points of the window will be assigned to the same cluster. Thus, these points can be assigned early without the need for calculating mean-shift operations and initializing windows at their location; this reduces the computation time.
after each update to the window location, nearby points are assigned using the following methods:
Assign all points within a radius r of the shift end point to the same cluster. This is because the window has just been shifted to an area of higher density, thus it is likely that points close to this area all belong to the same cluster

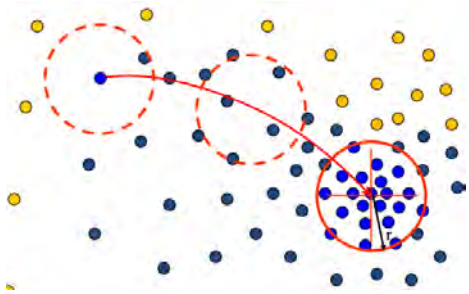


Figure 1: Adding points within radius r of shift end point

Assign all points within radius c of the path of the shift to the same cluster. Imagine initializing a window at one of the points near the path of the shift. Because all windows are shifted in the direction of the most dense area, it is likely that this window will be shifted in the same direction and the point will eventually be assigned to the same cluster. It is common to select the radius c such that $c \leq r$.

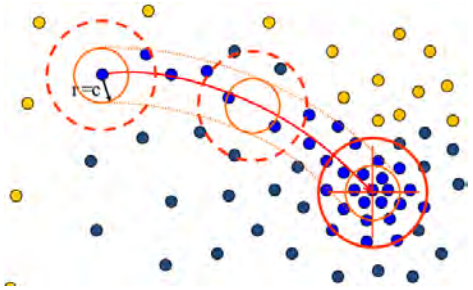


Figure 2: Adding points within radius c of window path

There is a trade off in selecting the values for r and c. The smaller the values, the fewer nearby points are assigned early and the higher the computation cost will be. However, the resulting cluster assignments will have less error if mean-shift was calculated without this method. The larger the values, the more nearby points are assigned resulting in faster speed increases, but also the possibility that the final cluster assignments will be less accurate to standard mean-shift.

1.2 Technical Details

In order to correctly shift the window you must first identify a nearby area with highest density to calculate the shift vector. This can be accomplished using the multivariate kernel density estimate, a means of estimating the probability density function of a random variable.

Given n data points $\mathbf{x} \in \mathbb{R}^d$, the multivariate kernel density estimate, using a radially symmetric (Comaniciu & Meer, 2002) kernel $K(x)$, is given by

$$\hat{f}_K = \frac{1}{nh^d} \sum_{i=1}^{n} K\left(\frac{\mathbf{x} - \mathbf{x}_i}{h}\right), \tag{1}$$

where h is known as the bandwidth parameter which defines the radius of the kernel. The radially symmetric kernel, $K(x)$ is given by

$$K(x) = c_k k(||x||^2), \tag{2}$$

where c_k represents a normalization constant.

Selecting the appropriate h is crucial to achieving accurate density estimation. Selecting a value that is too small results in a small radius and can be subject to noise in the data. Selecting a value that is too large includes many far away points and results in fewer clusters.

The resulting derivative of the multivariate kernel density estimate is given by

$$\nabla \hat{f}(x) = \frac{2c_{k,d}}{nh^{d+2}} \left[\sum_{i=1}^{n} g\left(||\frac{\mathbf{x} - \mathbf{x}_i}{h}||^2\right)\right] \left[\frac{\sum_{i=1}^{n} \mathbf{x}_i g\left(||\frac{\mathbf{x}-\mathbf{x}_i}{h}||^2\right)}{\sum_{i=1}^{n} g\left(||\frac{\mathbf{x}-\mathbf{x}_i}{h}||^2\right)} - \mathbf{x}\right], \tag{3}$$

where $g(x) = -K'(x)$ denotes the derivative of the selected kernel profile.

The first term in equation (3), $\frac{2c_{k,d}}{nh^{d+2}} \left[\sum_{i=1}^{n} g\left(||\frac{\mathbf{x}-\mathbf{x}_i}{h}||^2\right)\right]$, is proportional to the density estimate at $\mathbf{x}$. The second term, $\left[\frac{\sum_{i=1}^{n} \mathbf{x}_i g\left(||\frac{\mathbf{x}-\mathbf{x}_i}{h}||^2\right)}{\sum_{i=1}^{n} g\left(||\frac{\mathbf{x}-\mathbf{x}_i}{h}||^2\right)} - \mathbf{x}\right]$, is the mean-shift vector that points towards the direction of maximum density.

1.3 Mean-Shift Procedure

From a given point $\mathbf{x}_t$, calculate the following steps in order to reach the center of the cluster.

1. Compute the mean-shift vector $\mathbf{m}$ [term 2 from equation (3) above]:

$$\mathbf{m} = \left[\frac{\sum_{i=1}^{n} \mathbf{x}_i g\left(||\frac{\mathbf{x}-\mathbf{x}_i}{h}||^2\right)}{\sum_{i=1}^{n} g\left(||\frac{\mathbf{x}-\mathbf{x}_i}{h}||^2\right)} - \mathbf{x} \right] \tag{4}$$

2. Translate the density window using the mean-shift vector:

$$\mathbf{x}_i^{t+1} = \mathbf{x}_i^t + \mathbf{m}(\mathbf{x}_i^t) \tag{5}$$

3. Repeat steps 1 and 2 until convergence

$$\nabla f(\mathbf{x}_i) = 0 \tag{6}$$

1.4 Kernel Functions

A kernel, $K(x)$ is a non-negative function that integrates to 1 over all values of x. These requirements ensure that kernel density estimation will result in a probability density function.

Examples of popular kernel functions include:

- Uniform (rectangular)
  - $K(x) = \frac{1}{2}$ where $|x| \leq 1$ and 0 otherwise
- Gaussian
  - $K(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}u^2}$
- Epanechnikov (parabolic)
  - $K(x) = \frac{3}{4}(1 - x^2)$ where $|x| \leq 1$ and 0 otherwise

1.5 Mean-Shift Conclusions

The mean-shift algorithm has many pros and cons to consider.

Positives of mean-shift:

- Very general and application independent.
- Model-free. Mean-shift does not assume any prior shape of data clusters.
- Only depends on a single parameter defining the window size. Additional parameters will be required (r and c) if basin of attraction is applied.
- Finds a variable number of modes. The modes of a distribution function are the local maximums and the locations of these maximums are where the clusters will be centered.
- Robust to outliers in the data. Mean-shift won't try to force outliers to an existing cluster if they are further than the window size away from all other points.

Negatives of mean-shift:

- Output depends on window size and determining the appropriate window size is not trivial.
- Computationally relatively expensive to compute the mean-shift from all data points.
- Does not scale well with dimension of feature space.

2 Object recognition

2.1 Object recognition tasks

The field of object recognition can be broken down into a number of different visual recognition tasks.

Classification Classification involves teaching computers to properly label images based on the categories of object. It focuses on answering the questions, "is this image of a particular object?" and "does this image contain a particular object?" Typically, classification questions have yes or no answers. For example, typical classification questions could ask if the image contains a building or if the image is of a lake.

Image search This recognition task involves searching a collection of photos for photos of a particular object; an examples is Google Photos.

Organizing photo collections Object recognition can help organize photo collections based upon location of the photos, similarity of activities, the people who appear in the photos, and other recognizable objects.

Detection Detection focuses on the question, "where is a particular object in the image?" Traditional detection methods search for a bounding box that contains the object in question. Using segmentation techniques, the object can also be more specifically selected from the pixels in the image, for what is called accurate localization.
Detection can also be targeted at finding geometric and semantic attributes. For example, detection tasks include asking questions such as "is the traffic light red?", "what angle do two buildings make with one another?". Other common attributes found by detection include the distance objects are from one another, and what view of the object the image has.

Single Instance Recognition Instead of searching to generally categorize objects, single instance recognition seeks to recognize a particular object or landmark in images. For example, we may want to determine if the picture contains the Golden Gate Bridge or just a bridge. We may want to find a box of a specific brand of cereal.

Activity or event recognition Activity or event recognition is used to detect what is occurring in a photo. For example, we can ask what people are doing or if the event is a wedding.

2.2 Challenges

The challenges to building good object recognition methods include both image and category complications. Since computers can only see the pixel values of images, object recognition methods must account for a lot of variance.

Category numbers Studies have shown that humans can recognize roughly 10,000 to 30,000 categories of objects. Currently, the best visual recognition methods can work with about 200 categories for detection, and 1,000 for classification. Many researchers are working to build image datasets to improve object recognition; performing recognition on higher number of categories is the subject of many competitions!

Viewpoint variation There are many potential ways to view an object, and the change in viewpoint can lead an object to look very different.

Illumination Different levels of light, particularly low light or a different light direction, will cause shadows to shift and the details of an object to become obscured.

Scale Objects belonging to one category can come in a variety of sizes. If a classification is only trained on a particular size of object, then it will fail to recognize the same object in a different size. One way to solve this bias would be to figure out a way to make better datasets that account for scale variation.

Deformation Objects can change form and look very different, while still being considered the same object. For example, a person can be photographed in a number of poses, but is still considered a person if they're bending over or if their arms are crossed.

Occlusion Objects may be occluded, which could hide aspects of their characteristic geometry. While occlusion may be an excellent tool for artists (see Magritte), it is a challenge for computers.

Background clutter Similarities between the texture, color, and shapes in the background and the foreground can make it difficult to detect an object by blending in the object and/or causing it to appear differently.

Intra-class variation There can be significant shape variations event within one class of objects. For example, everything from a barstool to a lounge chair, and a beanbag to an abstract artistic bench can be considered a chair.

3 K-nearest neighbors

3.1 Supervised learning

We can use machine learning to learn how to label an image based upon its features. The goal of supervised learning is to use an existing data set to find the following equation:

$$y = f(\mathbf{x}) \tag{7}$$

In the above equation, y is the output, f is the prediction function, and $\mathbf{x}$ is a set of features from an image.

There are two phases of supervised learning: the train and the test phase. In the train phase, we fit an equation, f, to the training data set $\{(x_1, y_1)...\}$, which matches sets of image features to known identifier labels. f can be estimated by minimizing the prediction error, the difference between y and $f(\mathbf{x})$.

In the second phase, the test phase, we evaluate our equation, $y = f(x)$, using a new test example that has never been seen by f before. We can compare $f(\mathbf{x})$, and its expected output, y, to evaluate if the prediction function works on a new image. If the prediction function does not work on new images, it's not very useful!

One way to use supervised machine learning is to learn the decision rules for a classification task. A decision rule divides input space into decision regions that are separated by decision boundaries.

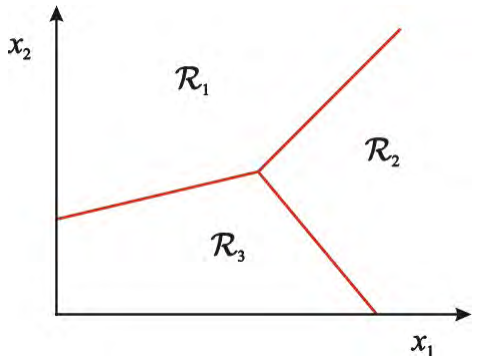


Figure 3: Decision regions and decision boundaries (in red) for three categories, R_1, R_2, and R_3 over a feature space with two dimension. **?**

3.2 Nearest neighbor classifier

The nearest neighbor classifier is an algorithm for assigning categories to objects based upon their nearest neighbor. We assign a new test data point the same label as its nearest training data point.

Nearest neighbors are found using the Euclidean distance between features. For a set of training data, where X^n and X^m are the n-th and the m-th data points, the distance equation is written as:

$$Dist(X^n, X^m) = ||X^n - X^m||^2 = \sqrt{\sum_i (X_i^n - X_i^m)^2} \tag{8}$$

The definition of the nearest neighbors classifier allows for complex decision boundaries that are shaped around each data point in the training set.

3.3 K-nearest neighbors classifier

We can enhance the nearest neighbors classifier by looking at the k nearest neighbors as opposed to just the nearest neighbor. The k-nearest neighbor classifier calculates the k nearest neighbors, and labels a new object by a score calculated over this set of nearest neighbors. A commonly used metric is to assign a new object to the category that a majority of their nearest neighbors belong to. Heuristics are used to break ties, and are evaluated based upon what works the best.

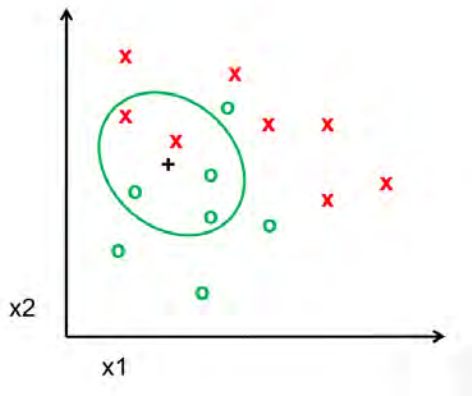


Figure 4: For the **+** data point, the green circle represents it's k-nearest neighbors, for $k = 5$. Since three out of five of its nearest neighbors are green circles, our test data point will be classified as a green circle. **?**

Like the nearest neighbor classifier, the k-nearest neighbor algorithm allows for complex decision boundaries using a relatively simple equation!

3.4 Pros of using k-nearest neighbors

K-NN is a very simple algorithm which makes it a good one to try out at first. IN addition, K-NN has very flexible decision boundaries. Another reason to use K-NN is that with infinite examples, 1-NN provably has error that is at most twice Bayes optimal error (proof is out of scope for this class).

3.5 Problems with k-nearest neighbors

3.5.1 Choosing the value of k

It's important to choose the proper value of K when using the K-nearest neighbor algorithm. If the value of K is too small, then the algorithm will be too sensitive to noise points. If the value of K is too high, then the neighborhood may include points from other classes. Similarly, as K increases, the decision boundaries will become more smooth, and for a smaller value of K, there will be more smaller regions to consider.

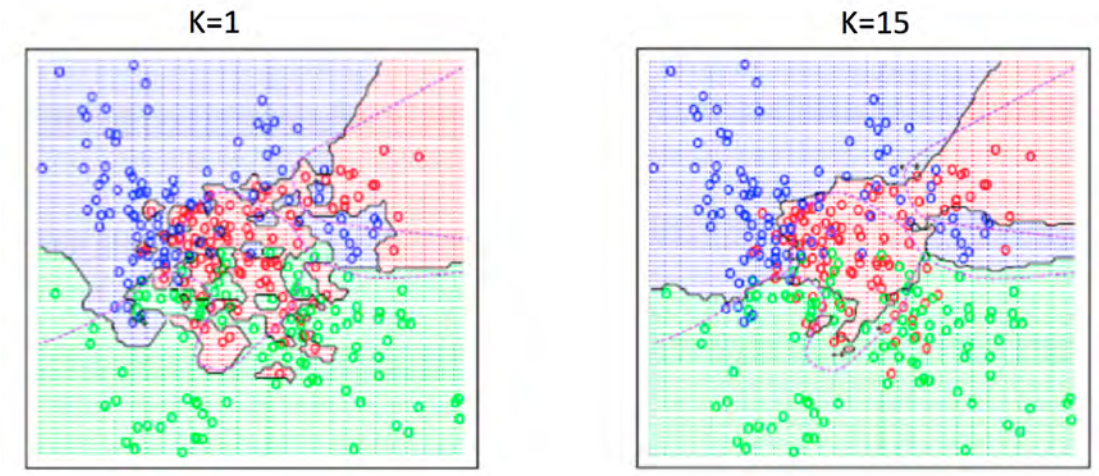


Figure 1: Comparing the result of choosing K=1 vs. K=15

Solution: Cross validate!
One way to find the proper value of K is to create a holdout cross validation/development set from the training set. From there, we would adjust the hyper parameters (e.g, K) on the development set to maximize training, and then 'test' the development set's accuracy. We would then change the validation set from the training set and repeat until we find the best value of K.

3.5.2 Euclidean measurement

Another issue that can arise with the K-nearest neighbor algorithm is that using the Euclidean measure might provide counter-intuitive results. See example below:

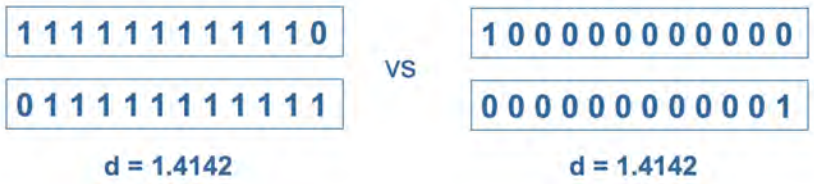


Figure 3: Different values, but using the Euclidean measure equates them to be the same

Solution: Normalize!
Normalizing the vectors to unit length will guarantee that the proper Euclidean measurement is used.

3.5.3 Curse of dimensionality

When using the K-nearest neighbor algorithm, one issue we need to keep in mind is how to deal with larger dimensions. When the dimensions grow, we need to cover a larger space to find the nearest neighbor. This makes it take longer and longer to calculate the nearest points, and the algorithm will run slower. This also means we will need to have a greater number of examples to train on. Currently, there is no best solution to the dimensionality problem.

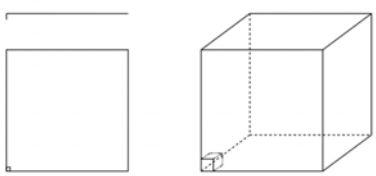


Figure 4: Larger dimensions require long calculation times

Problem: Assume there are 5000 points uniformly distributed in the unit hypercube, and we want to apply 5-NN (Nearest Neighbor). Suppose our query point is at the origin:

- In 1 dimension, we must go a distance of $\frac{5}{5000}$=0.001 on the average to capture 5 nearest neighbors
- In 2 dimensions, we must go $\sqrt{0.001}$ to get a square that contains 0.001 of the volume
- In d dimensions, we must go $(0.001)^{\frac{1}{d}}$

Note: K-nearest neighbors is just one of the many classifiers to choose from. That being said, it's not always easy to choose the "best" model for your data. It's best to think about how well you can generalize your model (i.e., how well does your model generalize from the data it was trained on to a new set?).

3.6 Bias-variance trade-off

:

The key to generalization error is to get low, generalized error, by finding the right number/type of parameters. There are two main components of generalization error: bias and variance. Bias is defined as how much the average model over all the training sets differs from the true model. Variance is defined as how much the models estimated from different training sets differ from each other. We need to find the right balance between bias and variance, hence, the bias-variance trade-off. Models with too few parameters are inaccurate because of a large bias (not enough flexibility). Similarly, models with too many parameters are inaccurate because of a large variance (too much sensitivity to the sample). To types of incorrect fitting will be listed below:

Underfitting: The model is too "simple" to represent all of the relevant class characteristics.

- High bias and low variance
- High training error and high test error

Overfitting: The model is too "complex" and fits irrelevant characteristics (noise) in the data.

- Low bias and high variance
- High training error and and high test error

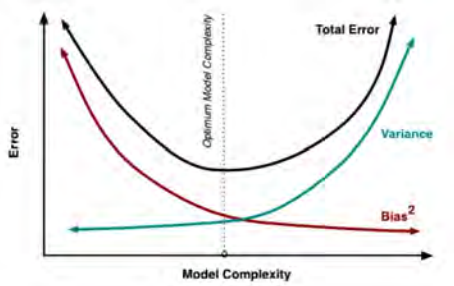


Figure 5: A graph that depicts the trade-off between bias and variance.

Lecture 12: Face Recognition & Dimensionality Reduction

Kyu seo Ahn, Jason Lin, Mandy Lu, Liam Neath, Jintian Liang
Department of Computer Science
Stanford University
Stanford, CA 94305
{kyuseo, jason0, mlu355, lneath, jtliang}@cs.stanford.edu

1 Overview and Motivation

1.1 Overview

Dimensionality reduction is a process for reducing the number of features used in an analysis or a prediction model. This enhances the performance of computer vision and machine learning-based approaches and enables us to represent the data in a more efficient way. There are several methods commonly used in dimensionality reduction. The two main methods covered in this lecture are Singular Value Decomposition (SVD) and Principal Component Analysis (PCA).

1.2 Motivation

Dimension reduction benefits models for a number of reasons.

1. Reduction in computational cost can be achieved. In many data sets, most of the variance can be explained by a relatively small number of input variables and their linear combinations. Focusing on these key components using dimensionality reduction, we can reduce the computational cost without losing much granularity in the data.
2. Reduce the effects of the "curse of dimensionality". In lecture 11 we learned that as we increase the dimension of a feature space, the number of data points needed to "fill in" that space with the same density explodes exponentially. That is to say, the more dimensions used in a machine learning algorithm, the more examples are needed for learning and the longer it takes the algorithm to analyze the same number of data points. By performing dimensionality reduction, we can mitigate the effects of this "curse of dimensionality".
3. Compress data. By reducing the dimensionality of an image, we can dramatically reduce the data storage requirements.

   In such cases the computational cost per data point may be reduced by many orders of magnitude with a procedure like SVD

2 Singular Value Decomposition

2.1 Overview

Intuitively, Singular Value Decomposition (SVD) is a procedure that allows one to represent the data in a new sub-feature space, such that the majority of variations in the data is captured; this is achieved by "rotating the axes" of the original feature space to form new axes which are linear combinations of the original axes/features (e.g. age, income, gender, etc. of a customer). These "new axes" are useful

because they systematically break down the variance in the data points (how widely the data points are distributed) based on each directions contribution to the variance in the data:

The result of this process is a ranked list of "directions" in the feature space ordered from most variance to least. The directions along which there is greatest variance are referred to as the "principal components" (of variation in the data); by focusing on the data distribution along these dimensions, one can capture most of the information represented in the original feature space without having to deal with a high number of dimensions in the original space (but see below on the difference between feature selection and dimensionality reduction).

2.2 Technical Details of Singular Value Decomposition

- SVD represents any matrix A as a product of three matrices: $A = U\Sigma V^T$ where U and V are rotation matrices, and Σ is a diagonal scaling matrix. For example:

$$\underset{U}{\begin{bmatrix}-.39 & -.92\\ -.92 & .39\end{bmatrix}} \times \underset{\Sigma}{\begin{bmatrix}9.51 & 0 & 0\\ 0 & .77 & 0\end{bmatrix}} \times \underset{V^T}{\begin{bmatrix}-.42 & -.57 & -.70\\ .81 & .11 & -.58\\ .41 & -.82 & .41\end{bmatrix}} = \underset{A}{\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\end{bmatrix}}$$

- For many readers, it may be sufficient to extract SVD values by writing: [U, S, V] = numpy.linalg.svd(A). However, the underpinnings of how SVD is computed is useful for later topics. Computers typically compute SVD by taking the following steps:
  - Compute the eigenvectors of AA^T. These vectors are the columns of U. Square root of the eigenvalues are the singular values (entries of Σ)
  - Compute the eigenvectors of A^TA. These vectors are columns of V (or rows of V^T)
- Since SVD relies on eigenvector computation, which are typically fast, SVD can be performed quite quickly; even for large matrices.
- A more detailed, geometric explanation of SVD may be found here[1].

2.3 Applications of Singular Value Decomposition

- One the most utilized applications of SVD is the computation of matrix inverses. If an arbitrary matrix A can be decomposed by way of: $A = U\Sigma V^T$, the inverse of A may be defined as: $A^+ = V^T\Sigma^{-1}U$. Although this inverse is an approximation, it allows one to calculate the inverses of many non-square matrices. MacAusland (2014) discusses the mathematical basis of this inverse, which is named the Moore-Penrose inverse, after its creators[3]. Unsurprisingly, a large variety of matrix problems can be solved be utilizing this approach.
- Singular Value Decomposition can also be used to compute the Principal Components of a matrix. Principal Components are heavily utilized in various data analysis and machine learning routines, hence SVD is typically a core routine within many programs.

3 Principal Component Analysis

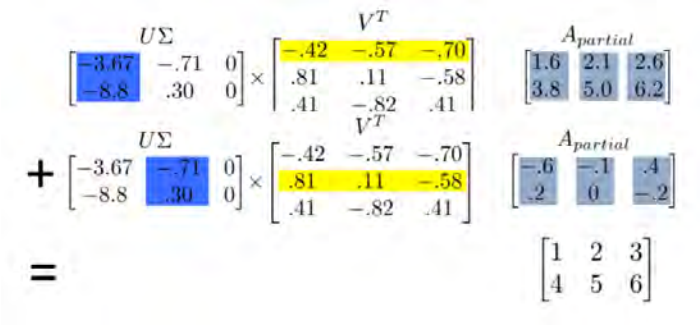
3.1 What are Principal Components?

Continuing with the SVD example we have above, notice that Column 1 of U gets scaled by the first value from Σ.

$$\underset{U\Sigma}{\begin{bmatrix}-3.67 & -.71 & 0\\ -8.8 & .30 & 0\end{bmatrix}} \times \underset{V^T}{\begin{bmatrix}-.42 & -.57 & -.70\\ .81 & .11 & -.58\\ .41 & -.82 & .41\end{bmatrix}} \quad \underset{A_{partial}}{\begin{bmatrix}1.6 & 2.1 & 2.6\\ 3.8 & 5.0 & 6.2\end{bmatrix}}$$

Then, the resulting vector UΣ gets scaled by row 1 of V^T to produce a contribution to the columns of A which is denoted $A_{partial}$. Each product of (column i of U)*(value i from Σ)*(row i of V^T)

produces a component of the final A.



In this process we are building the the matrix A as a linear combination of the columns of U. As seen above, if we use all columns of U, we rebuild A perfectly. However, in real-world data, **we can use only the first few columns of U** to get a good approximation of A. This arises due to the properties of Σ. Σ is a diagonal matrix where the largest value is in the top left corner, and the rest of the values on the diagonal decreases as you move to the right. Thus, the first few columns of U contribute the largest weight towards A. These first few columns of U are called **principal components**.

However, not all matrices can be easily compressed as in the previous example. One way to evaluate the feasibility is Principal Component Analysis. From a high level standpoint, we want to see if it's possible to remove dimensions that don't contribute much to the final image. We achieve this by analyzing the covariance matrix. Although the value of covariance doesn't matter as much, the sign of covariance does, with positive indicating positive correlation and negative indicating negative correlation. A covariance of zero indicates the two are independent of one another.

3.2 Performing Principal Component Analysis

Principal Component Analysis can be performed using the sklearn package: sklearn.decomposition.PCA. However, it was alluded to earlier that SVD can be used to perform Principal Component Analysis. A non-formal approach is outlined below:

1. Format your data into a $m * n$ matrix where m denotes the number of samples and p represents the number of features or variables corresponding to a single sample.

2. Center the matrix X by subtracting the mean and dividing by the standard deviation along each column(feature) in X

3. Diagonalizing X using SVD yields: $X = U\Sigma V^T$

4. Eigenvectors are the principal directions and the projections on these axises are the components. This ultimately means we want to compute XV

5. Since V holds eigenvectors and is thus orthonormal, $XV = U\Sigma V^T V = US$

6. (5) implies we simply need the columns of US, both matrices that are surfaced by SVD

Detailed explanations that elucidate the reasoning behind the above steps are discussed by Moore (1981) and can be found on numerous websites online.

3.3 Applications of Principal Components

- PCA has been extensively used in image compression. Much of the information captured within an image matrix can be extracted using matrices of lower ranks. This allows large images to be compressed without significant loss of quality. An example of PCA based compression, using only the first 16 principal components, is shown below:

Figure 1: Right: Original Image, Left: Compressed Image

With just the first 16 principal components, an image that closely resembles the original image can be reconstructed. The relative error as a result of the dimensions used for PCA for the above image is shown below:

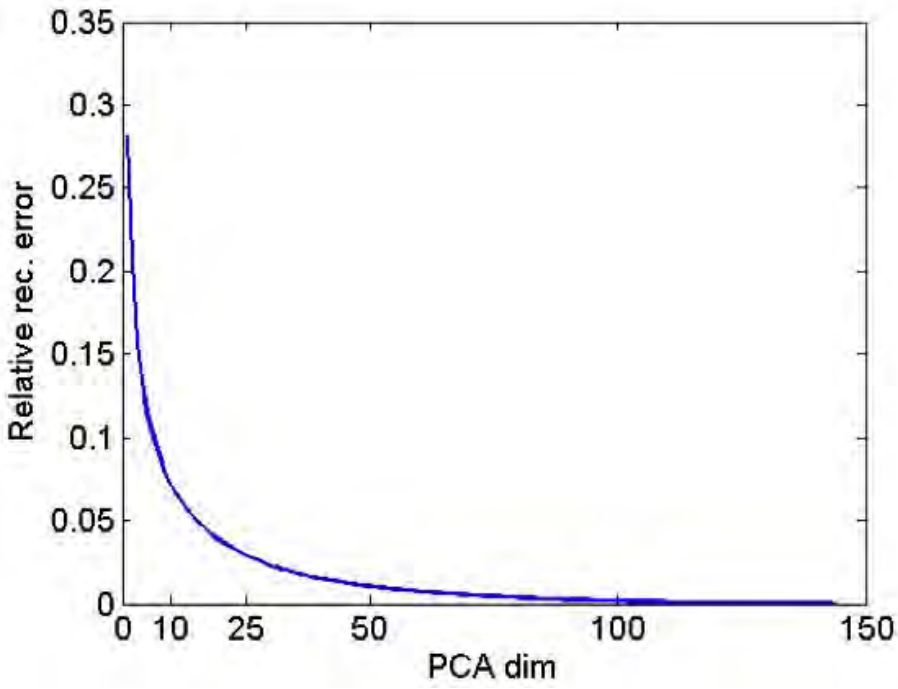


Figure 2: Relative Error as Function of PCA dimensions

- Web search engines also utilize PCA. There are billions of pages on the Internet that may have a non-trivial relation to a provided search phrase. Companies such as Google, Bing and Yahoo typically narrow the search space by only considering a small subset of this search matrix, which can be extracted using PCA[2]. This is critical for timely and efficient searches, and it speaks to the power of SVD.

In essence, PCA represents data samples as weights on various components - allowing one to essentially represent the difference between samples. This can significantly reduce data redundancy and can make algorithms used in a variety of industries more efficient and insightful!

References

[1] http://www.ams.org/samplings/feature-column/fcarc-svd.

[2] Snasel V. Abdulla H.D. Search result clustering using a singular value decomposition (svd). 2009.

[3] R. MacAusland. The moore-penrose inverse and least squares. 2014.

Lecture 13: Face Recognition and LDA

JR Cabansag, Yuxing Chen, Jonathan Griffin, Dunchadhn Lyons, George Preudhomme
Department of Computer Science
Stanford University
Stanford, CA 94305
{cabansag, yxchen28, jgriffi2, dunlyons, gpreud}@cs.stanford.edu

1 Introduction to Facial Recognition

1.1 Neuroscience Background

In the 1960's and 1970's, neuroscientists discovered that depending on the angle of observation, certain brain neurons fire when looking at a face. More recently, they have come to believe that an area of the brain known as the **Fusiform Face Area (FFA)** is primarily responsible for reacting to faces. These advances in the biological understanding of facial recognition have been mirrored by similar advances in computer vision, as new techniques have attempted to come closer to the standard of human facial recognition.

1.2 Applications

Computer facial recognition has a wide range of applications:

- Digital Photography: Identifying specific faces in an image allows programs to respond uniquely to different individuals, such as centering the image focus on a particular individual or improving aesthetics through various image operations (blur, saturation, etc).
- Surveillance: By recognizing the faces of specific individuals, we can use surveillance cameras to detect when they enter a location.
- Album Organization: If we can recognize a specific person, we can group images in which they appear.
- Person tracking: If we can recognize a specific person, we can track their location through frames of video (useful for surveillance or for robots in the home).
- Emotions and Expressions: By detecting emotions or facial expressions, we can build smart devices that interact with us based on mood.
- Security and Warfare: If we can recognize a specific person, we can identify potential threats in drone images and video.
- Teleconferencing: If we can recognize specific people, then teleconferencing applications could automatically provide information to users about who they are communicating with.

1.3 A Key Distinction: Detection vs. Recognition

While face **detection** determines whether an image contains faces and where in the image they are, face **recognition** determines to whom a detected face belongs to (i.e., identifying the identity of the person).

1.4 Space of Faces

If we consider an $m \times n$ image of a face, that image can be represented by a point in high dimensional space ($\mathbb{R}^{mn}$). But relatively few high-dimensional vectors consist of valid face images (images can contain much more than just faces), and thus the region that an arbitrary face image could fall into is a relatively small subspace. The task is to effectively model this subspace of face images.

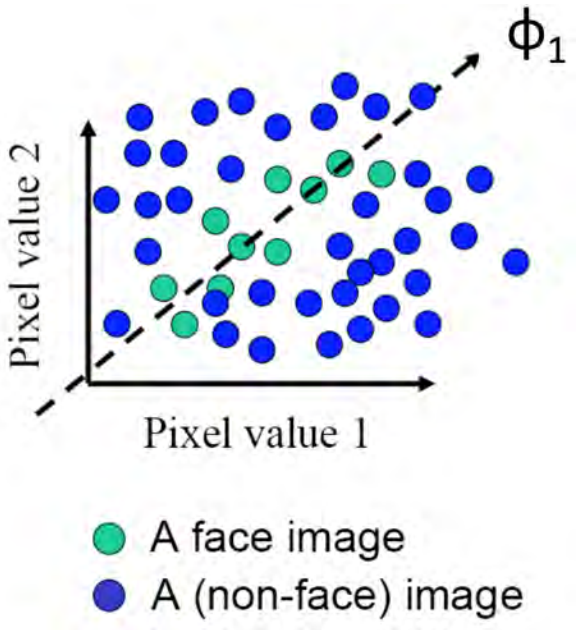


Figure 1: The region occupied by images of faces is a small subspace of the total space of images. Source: Lecture 13, slide 14

In order to model this subspace or "face-space" we compute the k-dimensional subspace such that the projection of the data points onto the subspace has the largest variance among all k-dimensional subspaces. This low-dimensional subspace captures the key appearance characteristics of faces.

2 The Eigenfaces Algorithm

2.1 Key Ideas and Assumptions

- Assume that most face images lie on a low-dimensional subspace determined by the first k directions of maximum variance.
- Use Principle Components Analysis (PCA) to determine the vectors or “eigenfaces” that span that subspace.
- Represent all face images in the dataset as linear combinations of eigenfaces, where eigenfaces are defined as the principal components of SVD decomposition.

2.1.1 What are eigenfaces?

"Eigenfaces" are the visual representations of the eigenvectors in the directions of maximum variance. They often resemble generic-looking faces.

Figure 2: Faces and Eigenfaces. Source: Lecture 13, slide 29

2.2 Training Algorithm

Algorithm 1 Eigenfaces Training Algorithm [2]

1: Align training images $x_1, ..., x_n$
2: Compute the average face:

$$\mu = \frac{1}{N} \sum x_i$$

3: Compute the difference image (the centered data matrix):

$$X_c = X - \mu 1^T = X - \frac{1}{N} X 1 1^T = X(1 - \frac{1}{N} 1 1^T)$$

4: Compute the covariance matrix:

$$\Sigma = \frac{1}{N} X_c {X_c}^T$$

5: Compute the eigenvectors of the covariance matrix Σ using PCA (Principle Components Analysis)
6: Compute each training image x_i's projections as

$$x_i \rightarrow ({x_i}^c \cdot \phi_1, {x_i}^c \cdot \phi_2, ..., {x_i}^c \cdot \phi_k) \equiv (a_1, a_2, ..., a_k)$$

where ϕ_i is the i'th-highest ranked eigenvector
7: The reconstructed face $x_i \approx \mu + a_1 \cdot \phi_1 + ... + a_k \cdot \phi_k$

Figure 3: The reconstructed face after projection. Source: Lecture 13, slide 25

2.2.1 Why can we do this?

Empirically, the eigenvalues (variance along eigenvectors) drop rapidly with the number of principle components, which is why we can reduce dimensionality without much loss of information.

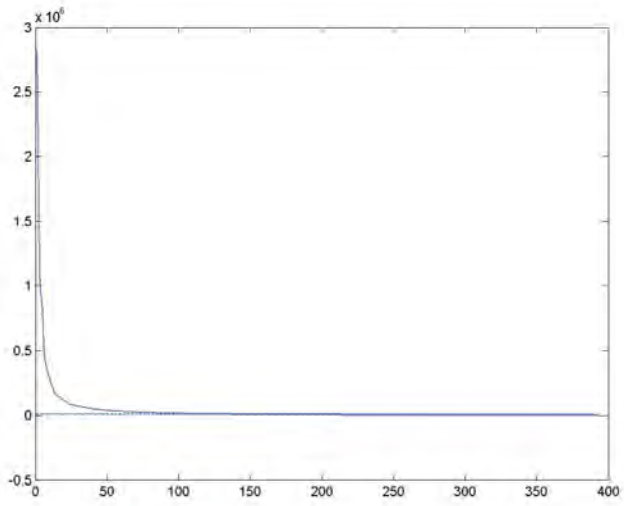


Figure 4: Eigenvalues sorted in descending order of magnitude. Source: Lecture 13, slide 26

2.2.2 Reconstruction and Error

We only select the top k eigenfaces, which reduces the dimensionality. Fewer eigenfaces result in more information loss, and hence less discrimination between faces.

Figure 5: Reconstructed faces with varying number of eigenfaces. Source: Lecture 13, slide 27

2.3 Testing Algorithm

Algorithm 2 Eigenfaces Testing Algorithm [2]

1: Take query image t
2: Project onto eigenvectors:

$$t \rightarrow ((t-\mu)\cdot\phi_1, (t-\mu)\cdot\phi_2, ..., (t-\mu)\cdot\phi_k) \equiv (w_1, w_2, ..., w_k)$$

3: Compare projection w with all N training projections. Use euclidean distance and nearest-neighbors algorithm to output a label

2.4 Advantages

- This method is completely knowledge-free – it does not know anything about faces, expressions, etc.
- It is a non-iterative (fast), globally-optimal solution.

2.5 Disadvantages

- This technique requires carefully controlled data.

    1. All faces must be centered in the frame. Otherwise the results may be noisy.
    2. The images must be the same size.
    3. There is some sensitivity to the face angle.

- Method is completely knowledge free.

    1. It makes no effort to preserve class distinctions.
    2. PCA doesn't take into account who it is trying to represent in this lower dimensional space (it doesn't take into account the labels associated with the faces). Therefore, it might map different faces near the same subspace, making it difficult for classifiers to distinguish between them.

- PCA projection is optimal for reconstruction from a low dimensional basis but may not be optimal for discrimination (the algorithm does not attempt to preserve class distinctions).

2.6 Beyond Facial Recognition: Expressions and Emotions

This technique also generalizes beyond simple facial recognition and can be used to detect expressions and emotions. The subspaces would therefore represent happiness, disgust, or other potential expressions, and the algorithm would remain unchanged.

Figure 6: Eigenfaces expressing happiness. Source: Lecture 13, slide 33

Figure 7: Eigenfaces expressing disgust. Source: Lecture 13, slide 34

3 Linear Discriminant Analysis

3.1 PCA vs. LDA

PCA and LDA are similar in that both reduce the dimensions of a sample. However, PCA projections don't consider the labels of the classes. An alternative approach is to move away from PCA toward an algorithm that is optimal for classification (as opposed to reconstruction). Linear Discriminant Analysis (LDA) finds a projection that keeps different classes far away from each other.

- PCA maintains maximum variance.
- LDA allows for class discrimination by finding a projection that maximizes scatter between classes and minimizes scatter within classes.

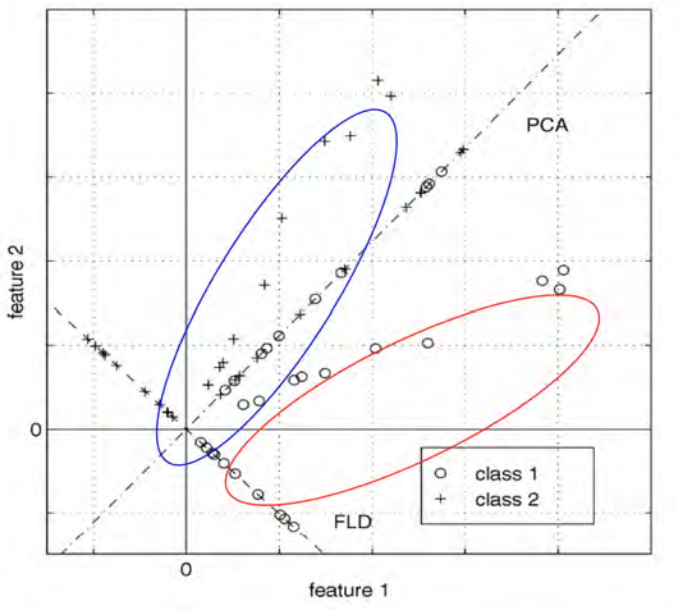


Figure 8: PCA vs. LDA. Source: Lecture 13, slide 41

The difference between PCA and LDA projections is demonstrated in the figure above. PCA preserves the maximum variance and maps the points of the classes along the line with the positive slope, which

makes it difficult to distinguish a points' class. Meanwhile, LDA maps the points onto the line with the negative slope, which results in points being located close to other points in their class and far from points in the opposite class.

3.2 General Idea

LDA operates using two values: between class scatter and within class scatter. Between class scatter is concerned with the distance between different class clusters, whereas within class scatter refers to the distance between points of a class. LDA maximizes the between-class scatter and minimizes the within-class scatter.

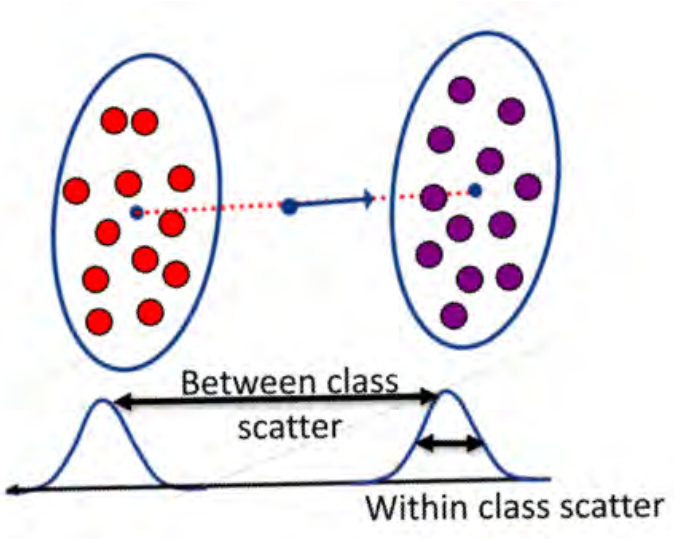


Figure 9: Between Class Scatter vs. Within Class Scatter. Source: Lecture 13, slide 43

3.3 Mathematical Formulation of LDA with 2 Variables

We want to find a projection w that maps points with classes 0 and 1 in the space $x \in R^n$ to a new space $z \in R^m$, such that $z = w^T x$. Ideally, $m < n$, and our projection should maximize the function:

$$J(w) = \frac{S_B \text{ when projected onto } w}{S_W \text{ when projected onto } w}$$

In this equation, S_B represents between class scatter and S_W represents the within-class scatter. Let us then define a variable μ_i that represents the mean of a class' points:

$$\mu_i = E_{X|Y}[X|Y = i]$$

Let us also define a variable Σ_i that represents the covariance matrix of a class:

$$\Sigma_i = E_{X|Y}[(X - \mu_i)(X - \mu_i)^T|Y = i]$$

Using these values, we can redefine our variables S_B and S_W to be:

$$S_B = (\mu_1 - \mu_0)^2 = (\mu_1 - \mu_0)(\mu_1 - \mu_0)^T$$
$$S_W = (\Sigma_1 + \Sigma_0)$$

Plugging these new values of S_B and S_W back into $J(w)$, we get:

$$J(w) = \frac{(\mu_1 - \mu_0)^2 \text{ when projected onto } w}{(\Sigma_1 + \Sigma_0) \text{ when projected onto } w} = \frac{w^T(\mu_1 - \mu_0)(\mu_1 - \mu_0)^T w}{w^T(\Sigma_1 + \Sigma_0)w}$$

We can maximize $J(w)$ by maximizing the numerator, $w^T(\mu_1 - \mu_0)(\mu_1 - \mu_0)^T w$, and keeping its denominator, $w^T(\Sigma_1 + \Sigma_0)w$ constant. In other words:

$$\max w^T(\mu_1 - \mu_0)(\mu_1 - \mu_0)^T w \text{ subject to } w^T(\Sigma_1 + \Sigma_0)w = K$$

Using Lagrange multipliers, we can define the Lagrangian as:

$$L = w^T S_B w - \lambda(w^T S_W w - K) = w^T(S_B - \lambda S_W)w + K$$

We must then maximize L with respect to λ and w. We can do so by taking its gradient with respect to w and finding where the critical points are:

$$\nabla_w L = 2(S_B - \lambda S_W)w = 0$$

Using this equation, we get that the critical points are located at:

$$S_B w = \lambda S_W w$$

This is a generalized eigenvector problem. In the case where $S_W^{-1} = (\Sigma_1 + \Sigma_0)^{-1}$ exists, we obtain:

$$S_W^{-1} S_B w = \lambda w$$

We can then plug in our definition of S_B to get:

$$S_W^{-1}(\mu_1 - \mu_0)(\mu_1 - \mu_0)^T w = \lambda w$$

Notice that $(\mu_1 - \mu_0)^T w$ is a scalar, and thus we can represent it by a term α such that:

$$S_W^{-1}(\mu_1 - \mu_0) = \frac{\lambda}{\alpha} w$$

The magnitude of w does not matter, so we can represent our projection w as:

$$w* = S_W^{-1}(\mu_1 - \mu_0) = (\Sigma_1 - \Sigma_0)^{-1}(\mu_1 - \mu_0)$$

3.4 LDA with N Variables and C Classes

3.4.1 Preliminaries

Variables:

- N sample images: $\{x_1, \cdots, x_N\}$
- C classes: $\{Y_1, Y_2, \cdots, Y_C\}$. Each of the N sample images is associated with one class in $\{Y_1, Y_2, \cdots, Y_C\}$.
- Average of each class: the mean for class i is $\mu_i = \frac{1}{N_i} \sum_{x_k \in Y_i} x_k$
- Average of all data: $\mu = \frac{1}{N} \sum_{k=1}^{N} x_k$

Scatter Matrices:

- Scatter of class i: $S_i = \sum_{x_k \in Y_i} (x_k - \mu_i)(x_k - \mu_i)^T$
- Within class scatter: $S_w = \sum_{i=1}^{c} S_i$
- Between class scatter: $S_b = \sum_{i=1}^{c} N_i(\mu_i - \mu)(\mu_i - \mu)^T$

3.4.2 Mathematical Formulation

We want to learn a projection W such that it converts all the points from $x \in \mathbb{R}^m$ to a new space $z \in \mathbb{R}^n$, where in general m and n are unknown:

$$z = w^T x, x \in \mathbb{R}^m, z \in \mathbb{R}^n$$

After the projection, the between-class scatter is $\widetilde{S}_B = W^T S_B W$, where W and S_B are calculated from our original dataset. The within-class scatter is $\widetilde{S}_W = W^T S_W W$. So the objective becomes:

$$W_{opt} = \underset{W}{\operatorname{argmax}} \frac{\left|\widetilde{S}_B\right|}{\left|\widetilde{S}_W\right|} = \underset{W}{\operatorname{argmax}} \frac{|W^T S_B W|}{|W^T S_W W|}$$

Again, after applying Lagrange multipliers we obtain a generalized eigenvector problem, where we have:

$$S_B w_i = \lambda_i S_W w_i, i = 1, \ldots, m$$

Note that $\operatorname{Rank}(S_B)$ and $\operatorname{Rank}(S_W)$ are limited by the number of classes (C) and the number of sample images (N):

$$\operatorname{Rank}(S_B) \leq C - 1$$
$$\operatorname{Rank}(S_W) \leq N - C$$

And therefore the rank of W_{opt} is limited as well.

3.5 Results: Eigenface vs. Fisherface

Belhumeur, Hespanha, Kriegman performed an experiment comparing the recognition error rates of PCA to LDA ("Eigenface vs Fisherface") using a dataset of 160 images of 10 distinct people. The images contained significant variation in lighting, facial expressions, and eye-wear. Error rates were determined using the "leaving-one-out" strategy, where a single image was classified by removing that image from the dataset and training on the other 159 images, at which point classification was done on the left-out image with a nearest-neighbors classifier. This process was repeated over all 160 images in order to determine an error rate [1].

Figure 10: Variation in Facial Expression, Eyewear, and Lighting. Source: [1]

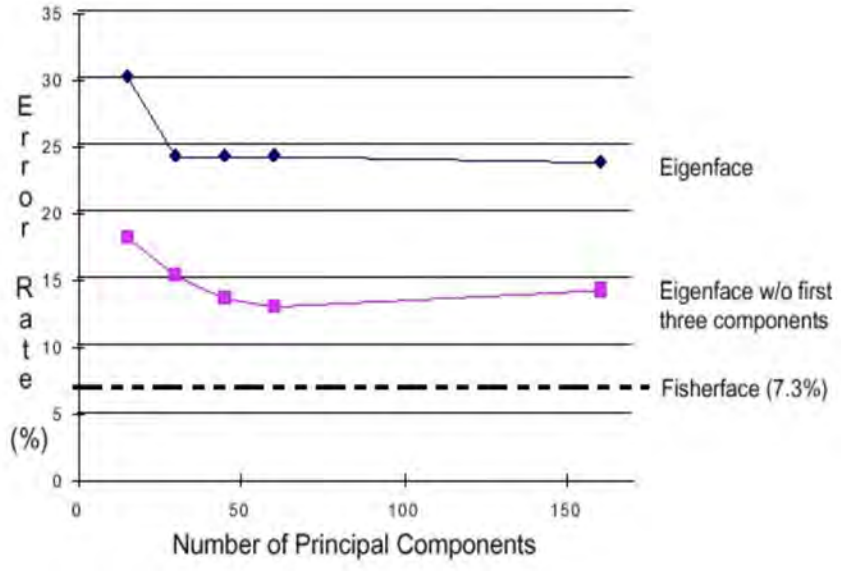


Figure 11: Eigenface vs. Fisherface. Source: Lecture 13, slide 61

Error rates for the two algorithms (and a variation of standard Eigenface) are plotted in the figure above. Eigenface's error rate actually improves by removing the first three principle components. Fisherface, as shown in the figure above, gives the lowest error rate.

References

[1] J. Hespanha P. Belhumeur and D. Kriegman. Eigenfaces vs. fisherfaces: Recognition using class specific linear projection. *IEEE Transactions on pattern analysis and machine intelligence*, 19(7):711–720, 1997.

[2] Matthew Turk and Alex Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3(1):71–86, 1991.

Lecture #14: Visual Bag of Words

Megha Srivastava, Jessica Taylor, Shubhang Desai, Samuel Premutico, Zhefan Wang, Sejal Jhawer
Department of Computer Science
Stanford University
Stanford, CA 94305
{meghas, jtaylor5, shubhang, samprem, zwang141,lbagdas, sejalj}@cs.stanford.edu

1 Introduction

In this lecture, we learn another approach to recognition. To recognize objects in images, we need to first represent them in the form of feature vectors. Feature vectors are mathematical representations of an image's important features. These feature vectors, for example, can be the raw color values of the image or contain information about the position of the pixel in the image as we have seen and implemented in Homework 5. We then create a space representation of the image to view the image values in a lower dimensional space. Every image is then converted into a set of coefficients and projected into the PCA space. The transformed data is classified using a classifier. Some examples of such classifiers include K-means and HAC. This process of going from an image to a useful representation of the image in a lower dimensional space can be achieved in many ways. In this lecture, we discuss another approach entitled Visual Bag of Words.

1.1 Idea of Bag of Words

The idea behind "Bag of Words" is a way to simplify object representation as a collection of their subparts for purposes such as classification. The model originated in natural language processing, where we consider texts such as documents, paragraphs, and sentences as collections of words - effectively "bags" of words. Consider a paragraph - a list of words and their frequencies can be considered a "bag of words" that represents the particular paragraph, which we can then use as a representation of the paragraph for tasks such as sentiment analysis, spam detection, and topic modeling.

Although "Bag of Words" appears to be associated with language, the idea of simplifying complex objects into collections of their subparts can be extended to different types of objects. In Computer Vision, we can consider an **image** to be a **collection of image features**. By incorporating frequency counts of these features, we can apply the "Bag of Words" model towards images and use this for prediction tasks such as image classification and face detection.

There are two main steps for the "Bag of Words" method when applied to computer vision, and these will further be explored in the Outline section below.

1. Build a "dictionary" or "vocabulary" of features across many images - what kinds of common features exist in images? We can consider, for example, color scheme of the room, parts of faces such as eyes, and different types of objects.

2. Given new images, represent them as histograms of the features we had collected - frequencies of the visual "words" in the vocabulary we have built.

1.2 Origins

The origins of applying the "Bag of Words" model to images comes from Texture Recognition and, as previously mentioned, Document Representation.

1. Textures consist of repeated elements, called textons - for example, a net consists of repeated holes and a brick wall consists of repeated brick pieces. If we were to consider each texton a feature, then each image could be represented as a histogram across these features - where the texton in the texture of the image would have high frequency in the histogram. Images with multiple textures, therefore, can be represented by histograms with high values for multiple features.

2. Documents consist of words which can be considered their features. Thus, every document is represented by a histogram across the words in the dictionary - one would expect, for example, the document of George Bush's state of the union address in 2001 to contain high relative frequencies for "economy", "Iraq", "army", etc.

Thus, a "bag of words" can be viewed as a histogram representing frequencies across a vocabulary developed over a set of images or documents - new data then can be represented with this model and used for prediction tasks.

2 Algorithm Summary

Let's describe in detail how the Bag of Words algorithm can be applied to a large set of images. As noted before, there are two main aspects of the Bag of Words algorithm: building our visual vocabulary, and using this vocabulary to represent new images by the frequencies of their visual "words."

2.1 Building a Visual Vocabulary

To build our visual vocabulary, we will make use of all the images in our dataset.

2.1.1 Extracting Interesting Features

The first step in building our visual vocabulary is to extract the features across all images in our dataset. We can extract features from images using a variety of methods. For example, we can simply split each image into a grid, and grab each of the subimages as features (shown below). Or, we can use corner detection of SIFT features as our features.

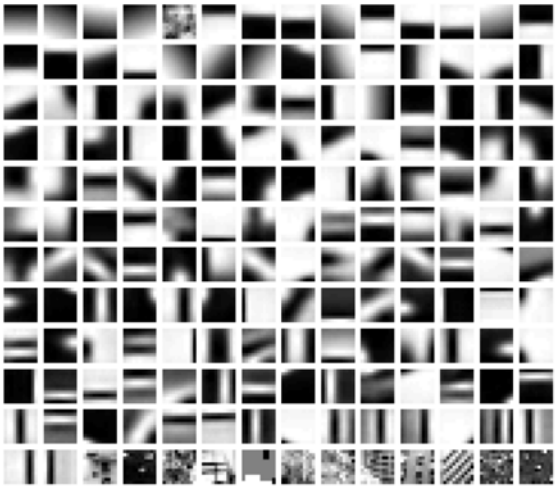
Using grid of subimages as features **?**

2.1.2 Clustering Features for Quantiation

Once we have extracted features from all of the images in the dataset, we must narrow down this large feature set into a smaller, less redundant set of common features, or "words" (to use the Natural Language Processing analogy of the algorithm). As mentioned above, in the Computer Vision application, the "words" are called textons.

To find the common "words" or textons, we simply cluster our full set of features. We can use any clustering technique to accomplish this. K-Means is most common, but Mean Shift or HAC may also work. After clustering, the cluster centers each represent a common "word" or texton in our visual vocabulary. An example of a visual vocabulary is given below.

Note: The word "codevector" is also often used as a synonym for a "word" or cluster center; similarly, "codebook" is the visual vocabulary containing all of our codevectors. This terminology is used to represent the notion of quantizing our features ("words").



Example of a visual vocabulary **?**

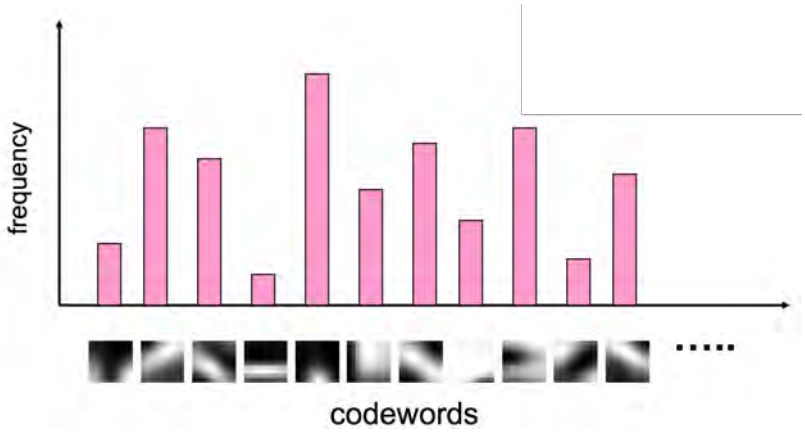
2.1.3 Challenges in Building the Visual Vocabulary

Choosing the size of our visual vocabulary (which is exactly equivalent to amount of clusters in our clustering algorithm) is an important hyperparameter. If it's too small, then our words (codevectors) are not representative of the underlying data. Our visual vocabulary will not be descriptive enough to distinguish between classes, and we will underfit. For example, as an analogy, if we wanted our visual vocabulary to differentiate between cats and dogs, a small vocabulary size might place both cat and dog features into the same cluster of all animals. On the other hand, if the vocabulary size is too large, then it will start to overfit the underlying data and be too descriptive. For example, if we wanted a visual word that represented "birds", an unnecessarily large vocabulary size might end up differentiating between heron and egret features. We must be conscious of this when picking the K value for K-Means (if, of course, we decide to use K-Means as our clustering algorithm).

2.2 Representing New Images by Visual Word Frequencies

Once we have built our visual vocabulary (codebook), we can use it to do interesting things. First, we can represent every image in our dataset as a histogram of visual word or codevector frequencies (shown below). We extract features from a new image using the same method we used to extract features while building our visual vocabulary. We then use our codebook to map the new image's features to the indexes of the closest words (codevectors) (feature quantization).

Our type of problem determines what we do next. If we have a supervised learning problem (i.e. our data has labels), we can train a classifier on the histograms. This classifier will be trained on the appearance of the visual words and hence will be a robust way to distinguish between classes. If we have an unsupervised learning problem (i.e. our data does not have labels), we can further cluster the histograms to find visual themes/groups within our dataset.



Representing our images as a histogram of texton frequencies **?**

We can create our visual vocabulary from a different dataset than the dataset that we are interested in classifying/clustering, and so long as our first dataset is representative of the second, this algorithm will be successful. In other words, this visual vocabulary can be universal.

2.3 Large-Scale Image Search

Large-scale image matching is one of the ways that the Bag-of-words model has been useful. Given a large database, which can hold tens of thousands of object instances, how can one match an images to this database?

The Bag-of-words model can help build the database. First, features can be extracted from the database images. Then we can learn a vocabulary using k-means (typical k:100,000). Next we compute the weights for each word. Going back to the word dictionary example, weighting the words can help us decrease the importance of certain words. If we are trying to find the topic of a document, we can give words like "the", "a", and "is" low weights since they are likely to be common between documents and used frequently within a document. With images we can do the same, giving useless features low weights and the more important features higher weights. Once the features have been weighted, we can create an inverted file mapping words to images.

Term Frequency Inverse Document Frequency (TF-IDF) scoring weights each word by it's document frequency.

The inverse document frequency (IDF) of a word j can be found by

$$IDF = \log(\frac{NumDocs}{NumDocs_{jappears}})$$

To compute the value of bin j in image I:

$$Bin_j = frequncy_{j\,in\,I} * IDF$$

We can create an inverted file that holds the mapping of words to documents to quickly compute the similarity between a new image and all of the images in the database. If we have images that have around 1000 features, and a database of around 100,000 visual words, each histogram will be extremely sparse. We would only consider images whose bins overlap with the new image.

Large-scale image search works well for CD covers and movie posters, and real-time performance is possible. The downside for the large scale image search is that the performance of the algorithm degrades as the database grows. Using the Bag-of-Words model for this problem sometimes results in noisy image similarities due to quantization error and imperfect feature detection.**?**

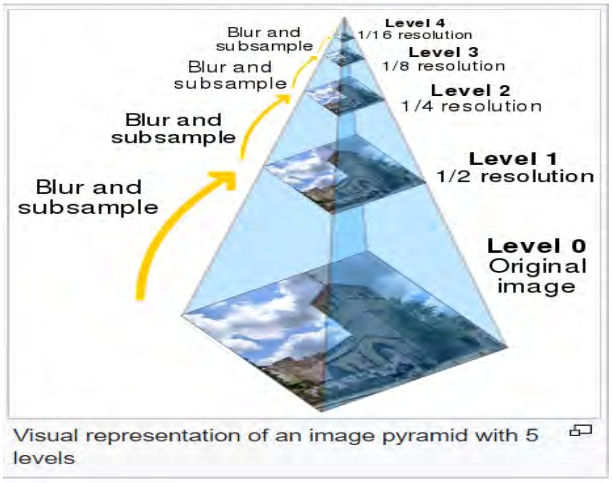
3 Spatial Pyramid Matching

3.1 Motivation

So far, we have not exploited the spatial information. But there is a simple yet smart method to incorporate the spatial information in the model: spatial pyramid matching.

3.2 Pyramids

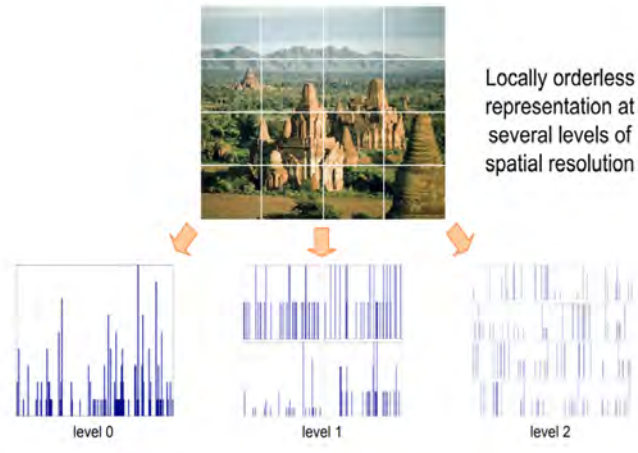
A pyramid is built by using multiple copies of the source image. Each level in the pyramid is $\frac{1}{4}$ of the size of the previous level. The lowest level is of the highest resolution and the highest level is of the lowest resolution. If illustrated graphically, the entire multi-scale representation looks like a pyramid, with the original image on the bottom and each cycle's resulting smaller image stacked one atop the other. **?**



Visual representation of an image pyramid with 5 levels

3.3 Bag of Words + Pyramids

Bag of Words alone doesn't discriminate if a patch was obtained from the top, middle or bottom of the image because it doesn't save any spatial information. Spatial pyramid matching partitions the image into increasingly fine sub-regions and allows us to computes histograms (BoW) of local features inside each sub-region. **?**
If the BoWs of the upper part of the image contain "sky visual words", the BoWs in the middle "vegetation and mountains visual words" and the BoWs at the bottom "mountains visual words", then it is very likely that the image scene category is "mountains".



3.4 Some results

Multi-class classification results
(100 training images per class)

	Weak features (vocabulary size: 16)		Strong features (vocabulary size: 200)	
Level	Single-level	Pyramid	Single-level	Pyramid
0 (1 × 1)	45.3 ±0.5		72.2 ±0.6	
1 (2 × 2)	53.6 ±0.3	56.2 ±0.6	77.9 ±0.6	79.0 ±0.5
2 (4 × 4)	61.7 ±0.6	64.7 ±0.7	79.4 ±0.3	**81.1** ±0.3
3 (8 × 8)	63.3 ±0.8	**66.8** ±0.6	77.2 ±0.4	80.7 ±0.3

Multi-class classification results (30 training images per class)

	Weak features (16)		Strong features (200)	
Level	Single-level	Pyramid	Single-level	Pyramid
0	15.5 ±0.9		41.2 ±1.2	
1	31.4 ±1.2	32.8 ±1.3	55.9 ±0.9	57.0 ±0.8
2	47.2 ±1.1	49.3 ±1.4	63.6 ±0.9	**64.6** ±0.8
3	52.2 ±0.8	**54.0** ±1.1	60.3 ±0.9	64.6 ±0.7

Strong features (ie.larger vocabulary size) is better than weaker features (ie. smaller vocabulary size). Notice also that as expected, incorporating pyramid matching always generate better result than single level feature extraction. This is exactly what we expected because under the same circumstance, pyramid approach encodes more information (ie.spacial information) than single-level approach does.

4 Naive Bayes

4.1 Basic Idea

Once we have produced a visual word histogram, we can use Naive Bayes to classify the histogram. To do so, we simply measure whether a given visual word is present or absent, and assume the presence/absence of one visual word to be conditionally independent of each other visual word given an object class.

Consider some visual word histogram *X*, where x_i is the count of visual word *i* in the histogram. We are only interested in the presence or absence of word *i*, we have $x_i \in \{0, 1\}$.

4.2 Prior

$P(c)$ denotes that probability of encountering one object class versus others. For all m object classes, we then have

$$\sum_{i=1}^{m} P(c) = 1$$

For an image represented by histogram x, and some object class c, we can compute

$$P(x|c) = \prod_{i=1}^{m} P(x_i|c)$$

We are able to make the above statement by assuming the mutual independence of each feature x_i when conditioned on c. This assumption is called the Naive Bayes Assumption.

4.3 Posterior

Using the prior equation, we can now calculate the probability than the image represented by histogram x belongs to class category c using **Bayes Theorem**.

$$P(c|x) = \frac{P(c)P(x|c)}{\sum_{c'} P(c')P(x|c')}$$

Note that we express $P(x)$ as the summation of $P(c')P(x|c')$, or equivalently $P(x, c')$ for all c' by using the law of total probability. Expanding the numerator and denominator, we can rewrite the previous equation as

$$P(c|x) = \frac{P(c)\prod_{i=1}^{m} P(x_i|c)}{\sum_{c'} P(c')\prod_{i=1}^{m} P(x_i|c')}$$

by once again using the Naive Bayes Assumption of conditional independence of each x_i on c'.

4.4 Classification

In order to classify the image represented by histogram x, we simply find the class c^* that maximizes the previous equation. That is to say, we look for the label $c = c^*$ such that the label that c has the highest probability of taking on given its feature data is c^*:

$$c^* = argmax_c P(c|x)$$

Since we end up multiplying together a large number of very small probabilities, we will likely run into unstable values as they approach 0. As a result, we use logs to calculate probabilities:

$$c^* = argmax_c log P(c|x)$$

Now consider two classes c_1 and c_2:

$$P(c_1|x) = \frac{P(c_1)\prod_{i=1}^{m} P(x_i|c_1)}{\sum_{c'} P(c')\prod_{i=1}^{m} P(x_i|c')}$$

and

$$P(c_2|x) = \frac{P(c_2)\prod_{i=1}^{m} P(x_i|c_2)}{\sum_{c'} P(c')\prod_{i=1}^{m} P(x_i|c')}$$

Since the denominators are identical, we can ignore it when calculating the maximum. Thus

$$P(c_1|x) \propto P(c_1)\prod_{i=1}^{m} P(x_i|c_1)$$

and

$$P(c_2|x) \propto P(c_2)\prod_{i=1}^{m} P(x_i|c_2)$$

and for the general class c:

$$P(c|x) \propto P(c)\prod_{i=1}^{m} P(x_i|c)$$

and using logs:

$$logP(c|x) \propto logP(c) + \sum_{i=1}^{m} logP(x_i|c)$$

Now, classification becomes

$$c^* = argmax_c P(c|x)$$
$$c^* = argmax_c logP(c|x)$$
$$c^* = argmax_c logP(c) + \sum_{i=1}^{m} logP(x_i|c)$$

Lecture 15: Detecting Objects by Parts

David R. Morales, Austin O. Narcomey, Minh-An Quinn, Guilherme Reis, Omar Solis
Department of Computer Science
Stanford University
Stanford, CA 94305
{mrlsdvd, aon2, minhan, greis, osolis9}@stanford.edu

1 Introduction to Object Detection

Previously, we introduced methods for detecting objects in an image; in this lecture, we describe methods that detect and localize generic objects in images from various categories such as cars and people. The categories that are detected depend on the application domain. For example, self-driving cars need to detect other cars and traffic signs.

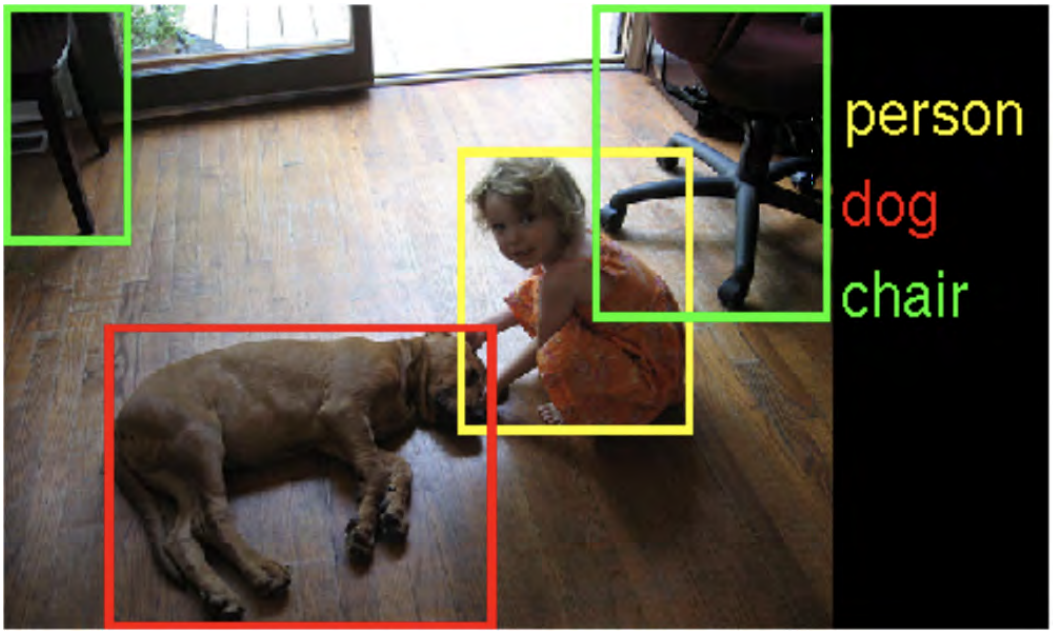


Figure 1: An example of an object detection algorithm detecting the categories of a person, dog, and a chair

1.1 Challenges

Object detection, however, faces many challenges. The challenges include the varying illumination conditions, changes in the viewpoints, object deformations, and intra-class variability; this makes objects of the same category appear different and makes it difficult to correctly detect and classify objects. In addition, the algorithms introduced herein only give the 2D location of the object in the image and not the 3D location. For example, the algorithms cannot determine if an object is in front

or behind another object. Additionally, the following object detectors do not provide the boundary of an object it finds; the object detector just provides a bounding box of where the object was found.

2 Current Object Detection Benchmarks

Today, object detection has practically been addressed for certain applications such as face detection. To evaluate the performance of an object detector, researchers use standardized object detection benchmarks. Benchmarks are used to make sure we are moving forward and performing better with new research.

2.1 PASCAL VOC

The first widely used benchmark was the PASCAL VOC Challenge [2], or the Pattern Analysis, Statistical Modeling, and Computational Learning Visual Object Classes challenge. The PASCAL VOC challenge was used from 2005 to 2012 and tested 20 categories. PASCAL was regarded as a high quality benchmark because its test categories had high variability within each category. Each test image also had bounding boxes for all objects of interest like cars, people, cats, etc. PASCAL also had annual classification, detection, and segmentation challenges.

2.2 ImageNet Large Scale Visual Recognition Challenge

The benchmark that replaced PASCAL is the ImageNet Large Scale Visual Recognition Challenge (ILSVR) [5]. The ILSVR Challenge tested 200 categories of objects, significantly more than what PASCAL tested, had more variability in the object types, and had many objects in a single image.

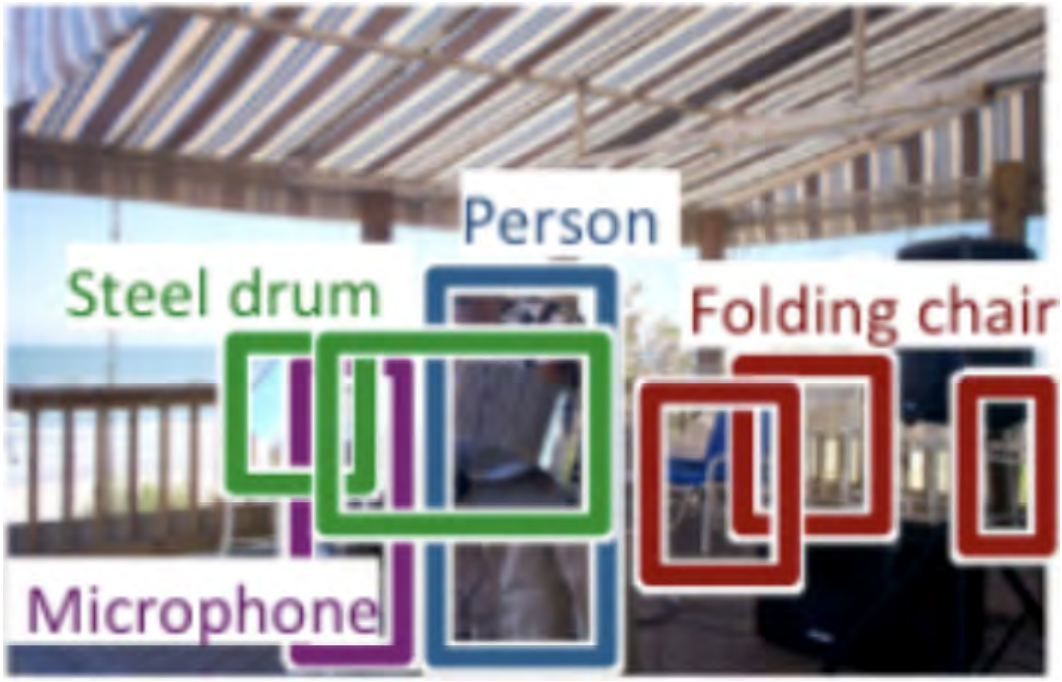


Figure 2: An example of a labeled ILSVR test image.

2.3 Common Objects in Context

Another benchmark that is still used today is the Common Objects in Context (COCO) challenge [3]. The COCO challenge tests 80 categories, but in addition to testing bounding boxes of locations of objects, it also tests object segmentation, which are detailed bounding areas of an object. Creating the test images for the dataset used in the COCO challenge is very time consuming because each object that is tested needs to be traced almost perfectly.

Figure 3: Object segmentation used for the COCO challenge.

3 Evaluating Object Detection

When evaluating an object detection algorithm, we want to compare the predictions with ground truth. The ground truth is provided by humans who manually classify and locate objects in the images.

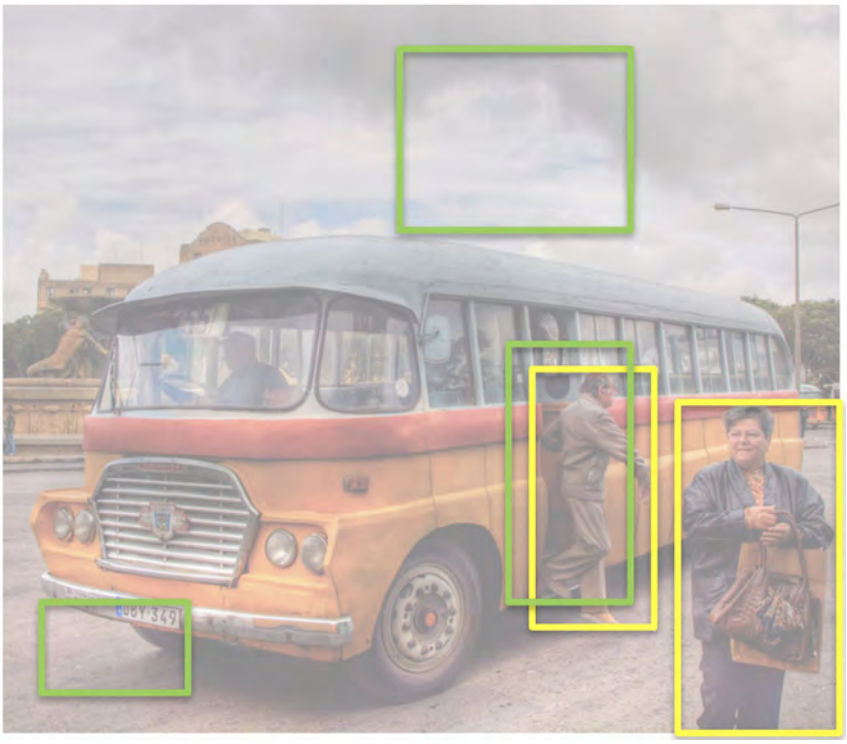


Figure 4: Yellow boxes represent ground truth while green boxes are predictions.

When comparing predictions with ground truth, there are four different possibilities:

1. True Positive (TP)
True positives are objects that both the algorithm (prediction) and annotator (ground truth) locate. In order to be more robust to slight variations between the prediction and ground truth, predictions are considered true positives when the overlap between the prediction and ground truth is greater than 0.5 (Figure 5a). The overlap between the prediction and ground truth is defined as the intersection over the union of the prediction and ground truth. True positives are also sometimes referred to as hits.

2. False Positive (FP)
False positives are objects that the algorithm (prediction) locates but the annotator (ground truth) does not locate. More formally, false positives occur where the overlap of the prediction and ground truth is less than 0.5 (Figure 5b). False positives are also referred to as false alarms.

3. False Negative (FN)
False negatives are ground truth objects that our model does not find (Figure 5b). These can also be referred to as misses.

4. True Negative (TN)
True negatives are anywhere our algorithm didn't produce a box and the annotator did not provide a box. True negatives are also called correct rejections.

(a) (b) (c)

Figure 5: Example classifications using an overlap threshold of 0.5. (a) True positive, because ground truth (yellow box) and prediction (green box) overlap is more than 0.5. (b) False positive, since the prediction boxes (green) do not overlap with any ground truth boxes. (c) False negative, since the ground truth box (yellow) is not detected by the model.

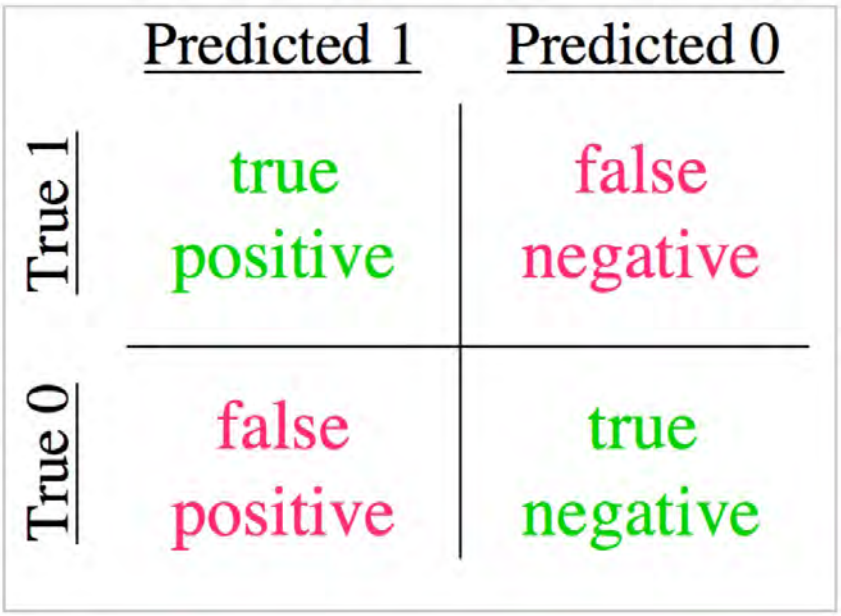


Figure 6: Summary chart of classifications, with green being counts you want to maximize and red being counts you want to minimize.

In general, we want to minimize false positives and false negatives while maximizing true positives and true negatives (Figure 6.

Using the counts of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN), we can calculate two measures: precision and recall.

$$Precision = \frac{TP}{TP + FP}$$

Precision can be thought of as the fraction of correct object predictions among all objects detected by the model.

$$Recall = \frac{TP}{TP + FN}$$

Recall can be thought of as the fraction of ground truth objects that are correctly detected by the model.

Figure 7: Predictions are green boxes while ground truth is yellow. All the ground truths are correctly predicted, making recall perfect. However, precision is low, since there are many false positives.

For every threshold we use to define true positives (In Figure 5 the overlap threshold was set to 0.5), we can measure the precision and recall. Using that information, we can create a Precision-Recall curve (PR curve). Generally, we want to maximize both precision and recall. Therefore, for a perfect model, precision would be 1 and recall would be 1, for all thresholds. When comparing different models and parameters, we can compare the PR curves. The better models have more area under the curve.

However, depending on the application, we may want specific values for precision and recall. Therefore, you can choose the best model by fixing, say the recall, and finding the model that has the best precision at that recall.

We can also use the counts of TP, FP, TN, and FN to see how our model is making errors.

4 A Simple Sliding Window Detector

The detection can be treated as a classification problem. Instead of attempting to produce the location of objects in an image by processing the entire image at once, slide a window over the image and classify each position of the window as either containing an object or not (Figure 9).

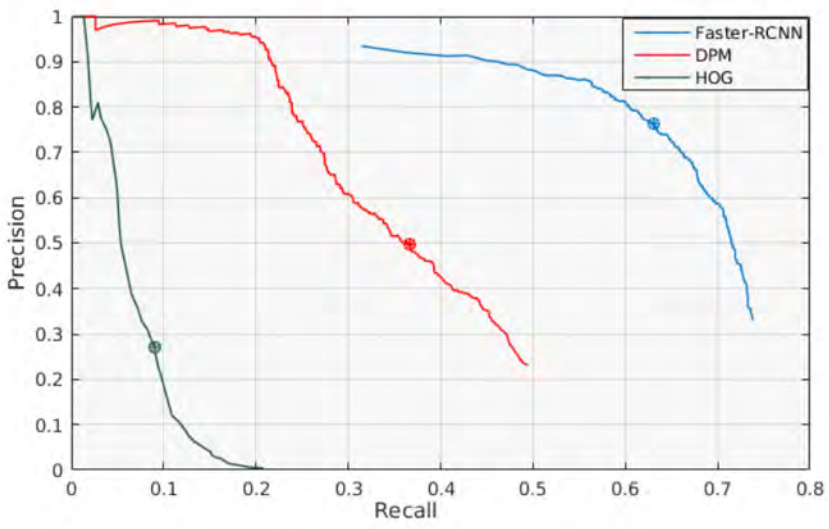
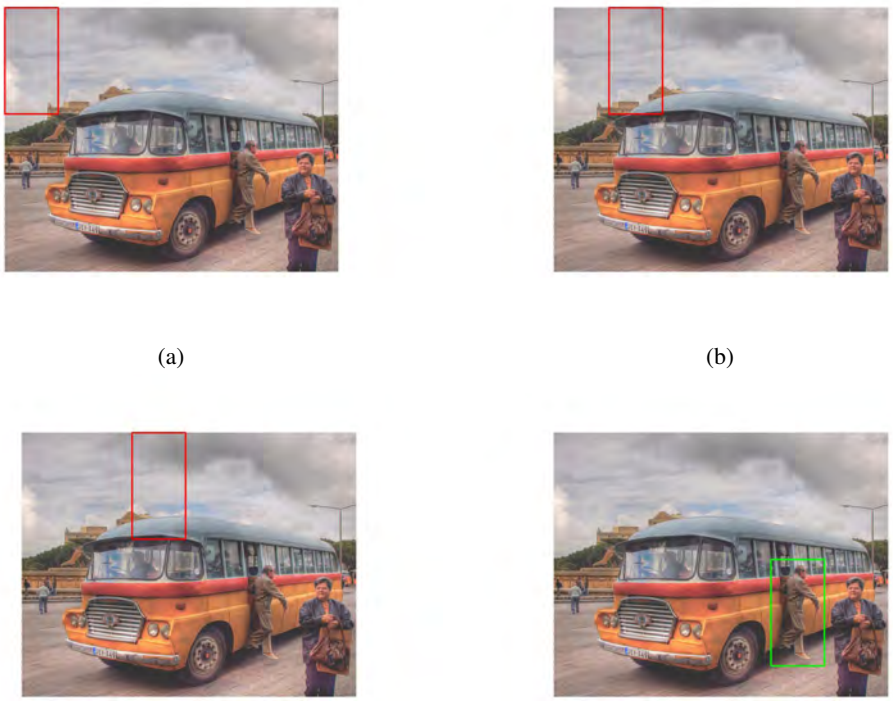


Figure 8: Faster-RCNN model is the best of the three models since it has the most area under the curve.



(a) (b)

(c) (d)

Figure 9: Consider the problem of detecting people in an image. (a) - (c) sliding window across image and at each position classifying window as not containing a person. (b) window over person and classifying window as containing a person. Image source: Flickr user neilalderney123

4.1 Feature Extraction and Object Representation

Dalal and Triggs [1] showed the effectiveness of using Histograms of Oriented Gradient (HOG) descriptors for human detection. Although their feature extraction methods were focused on human detection, they can be applied for detecting various objects.

Recall HOG descriptors from lecture 8. An image window is divided into blocks; the magnitude of the gradients of the pixels in each block are accumulated into bins according to the direction of the gradients. These local histograms of the blocks are then normalized and concatenated to produce a feature representation of the image window.

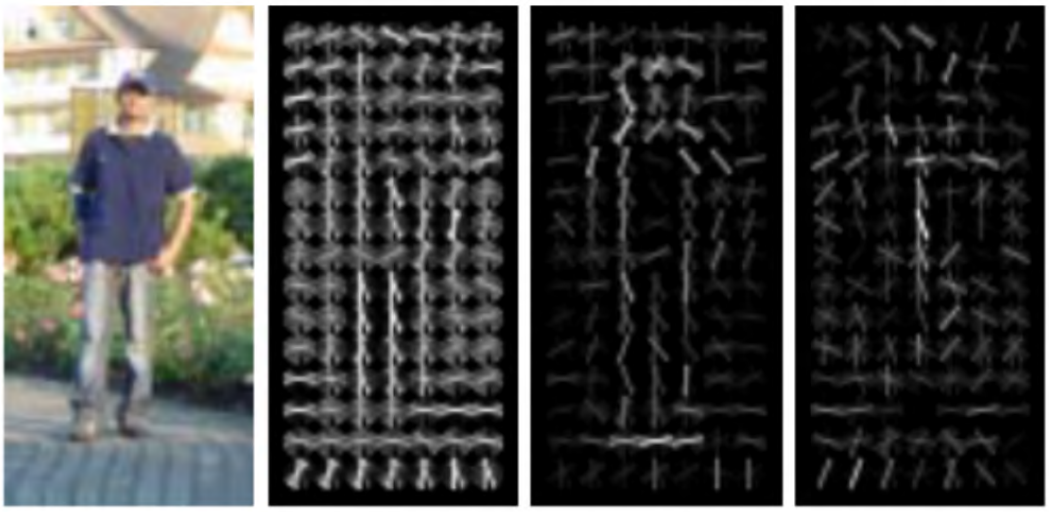


Figure 10: An image of a person along with its respective HOG descriptor. Note that the last two images are the HOG descriptor weighted by the positive and negative weights of the classifier using them. The outline of the person is very visible in the weighted descriptors. Image source: Dalal and Triggs [1]

Figure 10 shows an example of transforming an image into a HOG feature space. Producing a prototypical representation of an object would then involve considering many image windows labeled with containing that object. One approach to creating this representation would be to train a classifier on the HOG descriptors of these many labeled image windows and then proceed to use the trained classifier to classify the windows in images of interest. In their aim to improve human detection, for example, Dalal and Triggs [1] train a linear Support Vector Machine on the HOG descriptors of image windows containing people.

A more simple approach, and the approach that will be assumed below, is that of averaging the window images containing an object of interest and then extracting the HOG descriptor of that average image to create a template for the object (Figure 11).

4.2 Classifying Windows

Now that we have a method for extracting useful features from an image window and for constructing object templates, we can proceed to detecting objects in images. The idea is to compare each window with the object template and search for matches. That is, the object template itself acts as a filter that slides across the image. At each position, the HOG descriptor of the window being compared is extracted and a similarity score between the two HOG descriptors is computed. If the similarity score at some location is above a predefined detection threshold, then an object can be said to have been detected in the window at that location.

The similarity score can be as simple as the dot product of the window HOG descriptor and the template HOG descriptor. This is the scoring method that is assumed in following sections.

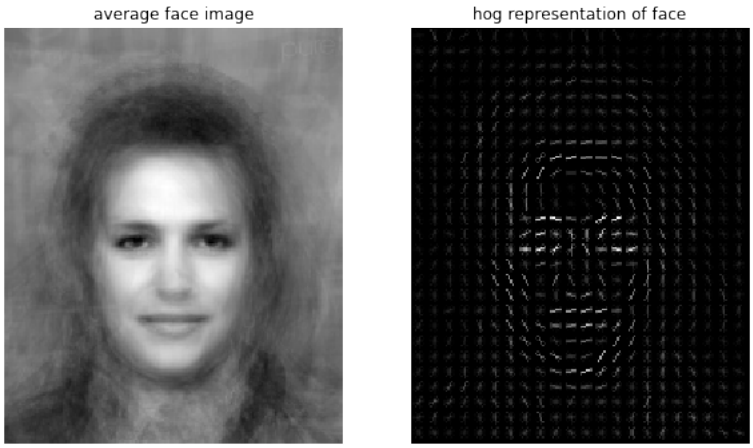


Figure 11: The average face image above is created by averaging 31 aligned face images of the same size. The HOG descriptor of this average face image can then be used as a template for detecting faces in images.

As effective as the method seems so far, it's success is very limited by the size of the sliding template window. For example, consider the case where objects are larger than the template being used to detect them (Figure 12).

(a) (b)

Figure 12: Consider, again, the problem of detecting people, except this time our sliding window is much smaller. (a) The template and sliding window are still large enough to detect the smaller, distant person. (b) The person in the foreground is a lot bigger than our window size and is not being detected.

4.3 Multi Scale Sliding Window

To account for variations in size of the objects being detected, multiple scalings of the the image are considered. A feature pyramid (Figure 13) of different image resizings is created. The sliding window technique is then applied as usual over all the the pyramid levels. The window that produces the highest similarity score out of the resizings is used as the location of the detected object.

5 The Deformable Parts Model (DPM)

The simple sliding window detector is not robust to small changes in shape (such as a face where the eyebrows are raised or the eyes are further apart, or cars where the wheel's may be farther apart or the car may be longer) so we want a new detection model that can handle these situations. Recall the

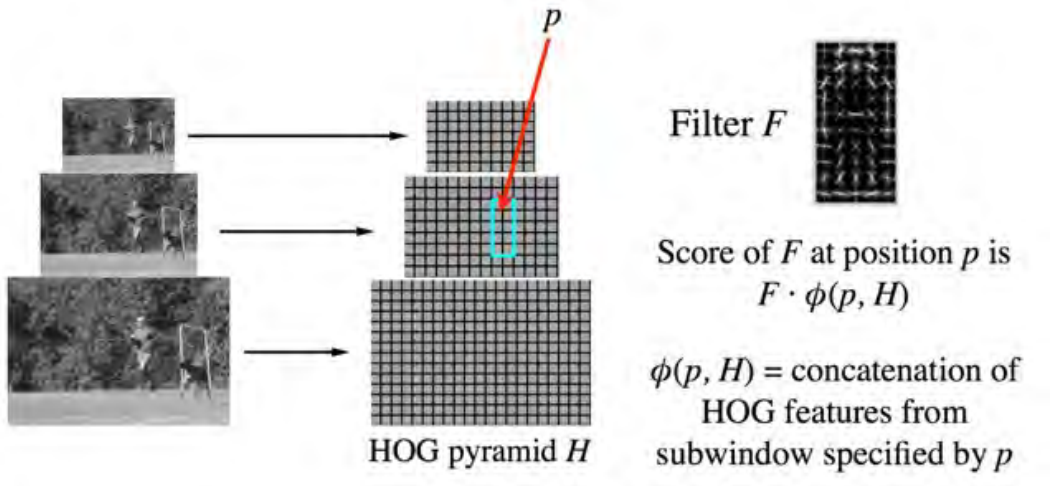


Figure 13: Using a feature pyramid of different image resizings allows the object template to match with objects that might have originally been bigger or much smaller than the the template. Image source: Lecture 15, Slide 40

.

bag of words approach, in which we represent a paragraph as a set of words, or an image as a set of image parts. We can apply a similar idea here and detect an object by its parts instead of detecting the whole singular object. Even if the shape is slightly altered, all of the parts will be present and in approximately the correct position with some minor variance.

5.1 Early Deformation Model for Face Detection

In 1973, Fischler and Elschlager developed a deformable parts model for facial recognition [4]: the parts of the face (such as eyes, nose, and mouth) are detected individually, and there are spring-like connections between each part that allows each part to move slightly, relative to the other parts, but still largely conform to the typical configuration of a face. This allows a face to be detected even if the eyes are farther apart or a change in orientation pushes some parts closer together, since each part has some flexibility in its relative location in the face. See Figure 14 for illustration

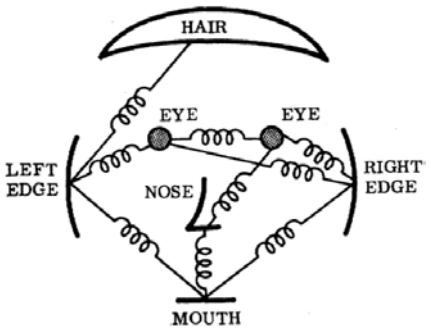


Figure 14: An illustration of Fischler and Elschlager's deformable face model[4]

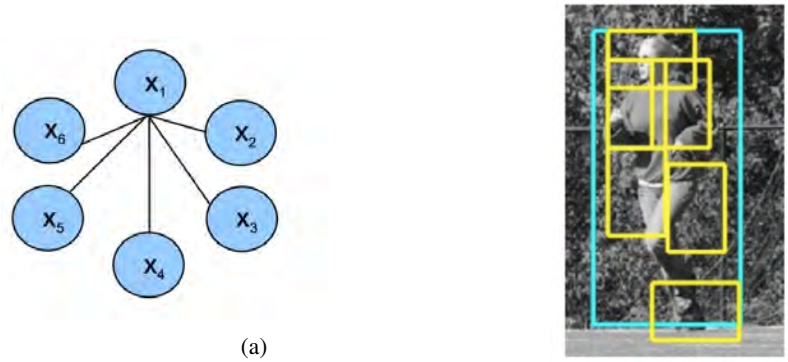
.

More formally, the springs indicate that there is a desired relative position between two parts, and just like a spring stretches or compresses, we allow some deviations from that desired position, but

apply an increasing penalty for larger deviations, much like a string pulls back harder and harder as it is stretched further.

5.2 More General Deformable Parts Models

The deformable model depicted in Figure 14 is one specific deformable model for face detection, where Fischler and Elschlager chose springs between parts that worked well with their methods for face detection. For a more general deformable parts model, one popular approach is the star-shaped model, in which we have some detector as the root and we have springs between every other part and the root (see Figure 15a for illustration)



(a)

(b)

Figure 15: On the left is a visualization of the star model, with x1 as the root, and on the right is an example of person detection with a star model for deformations

In face-detection, for example, we could have a detector for the entire body as the root and thus define the location of all body parts relative to the location of entire body. This is shown in Figure 15b, where the cyan box is the location of the bounding box from global person detection (which we use as the root) and the yellow boxes are the bounding boxes resulting from the detection of each part.

This means that the head should be near the top-center of the global location of the person, the feet should be near the bottom, the right arm should be near the left of image (if detector is for a front facing person), etc. In this class we will assume that we already know which parts to use (such as head, arms, and legs if we are detecting people), but it is possible to learn the parts for optimal object detection through machine learning

5.3 Examples of Deformable Parts Models

It is typical to use a global detector for the desired object (such as a person or a bike) as the root, and to use smaller and more detailed filters to detect each part. As a simple example, see Figure 16

It is also common to use a multi-component model for multiple orientations of an object, in which we have a global filter and multiple parts filters for each orientation. A deformable model will protect against the changing positions of parts due to mild rotation, but for more significant rotations such as 90 degrees, all of the parts looks different and require different detectors for each rotation. As an example, in figure 17, each row corresponds to an orientation. Also, the left column is a global car detector for a particular orientation, the middle column contains all of the finer detailed parts filters for that orientation, and the right column shows the deformation penalties for each part in that orientation (where darker grey in the center is smaller penalties and white further from the center is larger penalties).

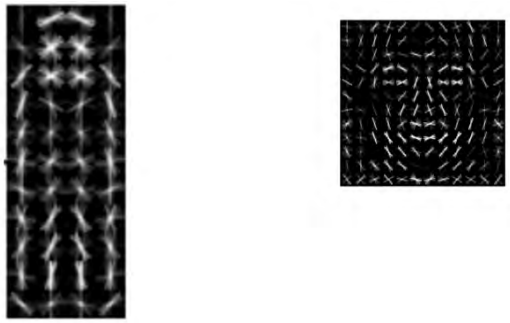


Figure 16: A global HOG filter for a person and a more detailed HOG filter for the head

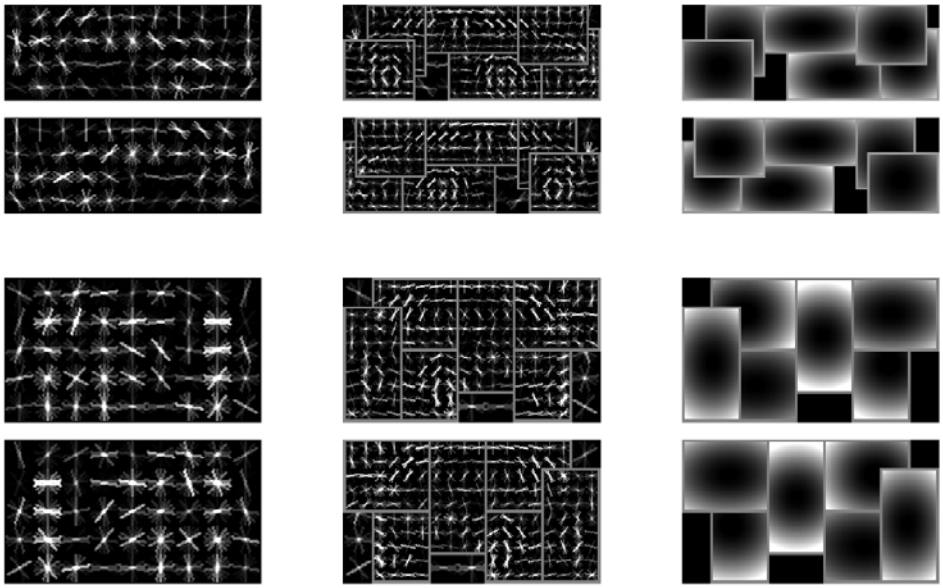


Figure 17: Deformable Parts model of a car with multiple orientations

As another example, consider 2 orientations for bike detection. A bicycle looks very different from the front and from the side so we want multiple detectors. Here we can see the parts identified in the original image alongside the deformable model

5.4 Calculating Score for Deformable Parts Models

To implement a deformable parts model we need a formal way to calculate score. To do this we will calculate a global score from the global object detector, and then calculate a score for each part, determined by it's deformation penalty. The final score is the global score minus all of the deformation penalties, such that an object that is detected strongly but has many of the parts very far from where they should be will be highly penalized.

To express this more formally we will first define some variables: The entire model with n parts is encompassed by an (n+2) tuple:

$$(F_0, P_1, P_2, ...P_n, b)$$

where F_0 is the root filter, P_1 is the model for the first part, and b is a bias term. Breaking it down further, each part's model P_i is defined by a tuple

$$(F_i, v_i, d_i)$$

where F_i is the filter for the i-th part, v_i is the "anchor" position for part i relative to the root position, and d_i defines the deformation cost for each possible placement of the part relative to the anchor

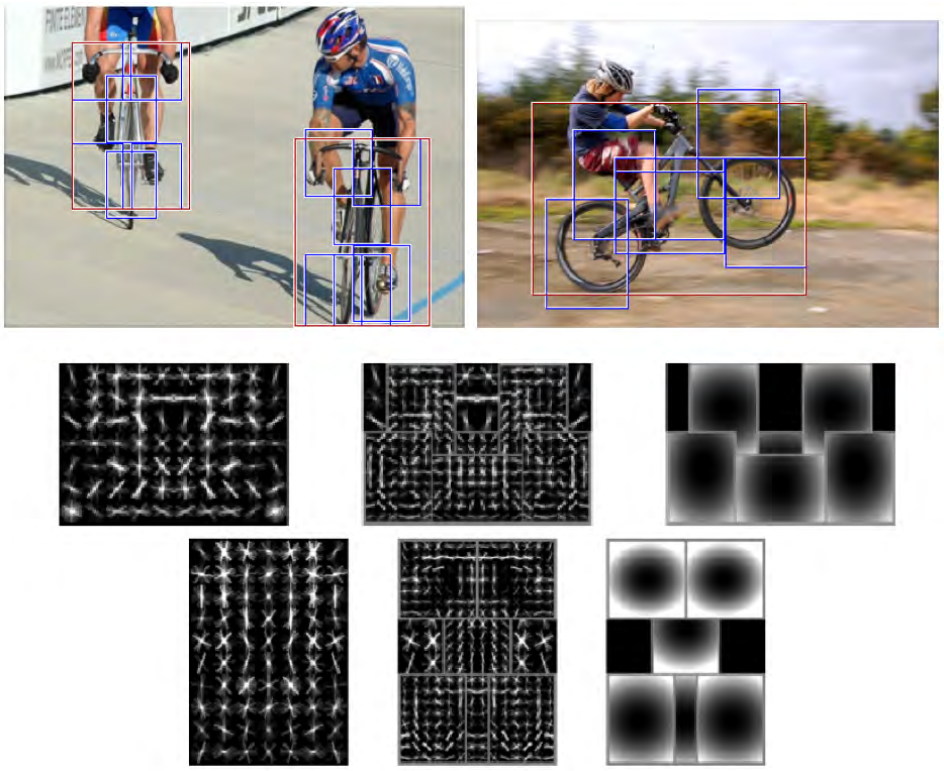


Figure 18: Deformable Parts model of a bike with original image shown

position.

We can calculate the location of the global filter and each part filter with a HOG pyramid (see Figure 19). We run the global HOG filter and each part's HOG filter over the image at multiple scales so that the model is robust to changes in scale. The location of each filter is the location where we see the strongest response. Since we are taking the location of responses across multiple scales we have to take care that our description of the location of each part is scale-invariant (one way this can be done is by scaling the maximum response map for each part up to the original image size and then taking scale-invariant location).

We calculate the detection score as

$$\prod_{i=0}^{n} F_i \cdot \phi(p_i, H) - \sum_{i=1}^{n} d_i(dx_i, dy_i, dx_i^2, dy_i^2)$$

Taken as a whole, this means that we are finding detection score of the global root and all the parts and subtracting all of the deformation penalties.

The left term represents the product of the scores for the global filter and each part filter (note that this is identical to the simpler Dalal and Triggs [1] or sliding window method explained previously). Recall that the score for each filter is the inner product of the filter (as a vector) and $\phi(p_i, H)$ (defined as the HOG feature vector of a window defined by position p_i of the filter). Note that the windows can be visualized in the HOG pyramid in Figure 19: the window for the root is the cyan bounding box, and the window for each of the parts is the yellow bounding box corresponding to that part. We are taking the HOG feature vector of the portion of the image enclosed in these bounding boxes and seeing how well it matches with the HOG features of the template for that part.

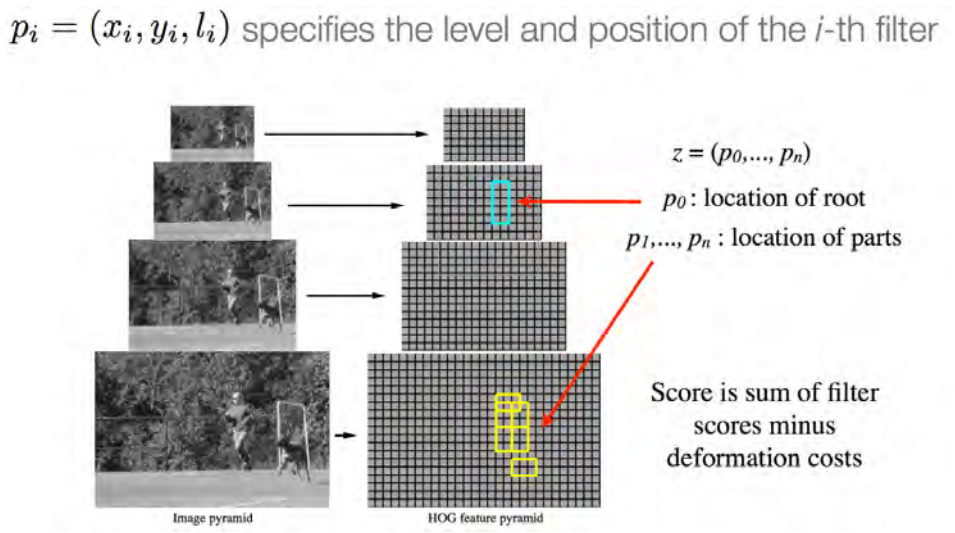


Figure 19: Illustration of HOG pyramid for deformable parts detector

Returning to the score formula, the right term represents the sum of the deformation penalties for each part. We have d_i representing the weights of each penalty for part i, corresponding to quantities dx_i (the distance in x direction from the anchor point where the part should be), dy_i (the distance in y direction from the anchor point where the part should be), as well as dx_i^2 and dy_i^2. As an example, if $d_i = (0, 0, 1, 0)$, then the deformation penalty for part i is the square of the distance in the x direction of that part from the anchor point. All other measures of distance are ignored.

6 The DPM Detection Pipeline

The Deformable Parts Model detection pipeline has several stages. We must first use the global filter to detect an object. Then, the parts filters are used to calculate the overall score of that detection.

1. Generate copies of the original image at different resolutions (so that the fixed window size can capture objects at varied scales); store the HOGs for these different resolutions, as that is what the filters will be applied to.

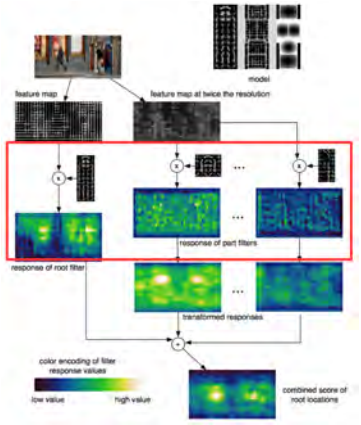


Figure 21: Step 2 of the detection pipeline

2. Apply the global filter to these images. Upon a detection by the global filter, apply the parts filters. This step represents the section of the pipeline depicted in Fig. 21, and contributes the term:

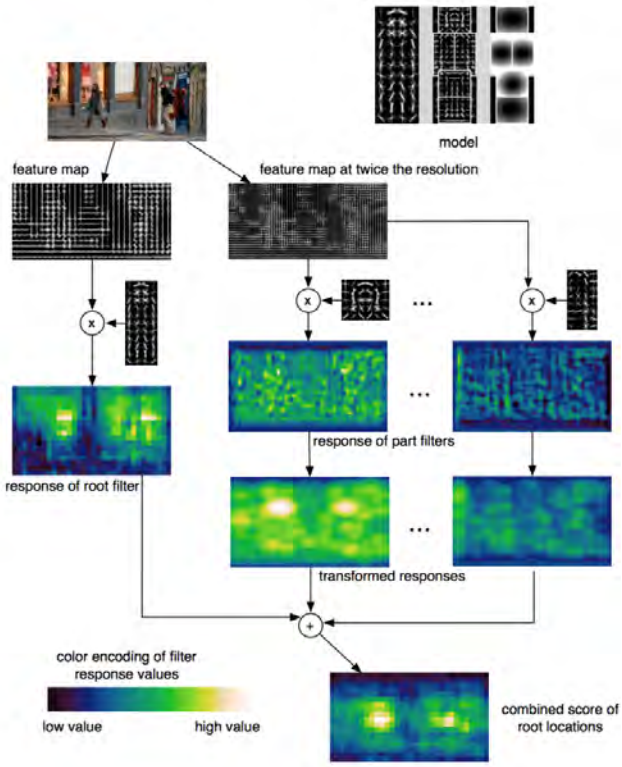


Figure 20: Illustration of full DPM Pipeline

$$\prod_{i=0}^{n} F_i \cdot \phi(p_i, H)$$

3. Having applied the parts filter, we now calculate the spatial costs (i.e., a measure of the deformation of the parts with regards to the global):

$$\sum_{i=1}^{n} d_i(dx_i, dy_i, dx_i^2, dy_i^2)$$

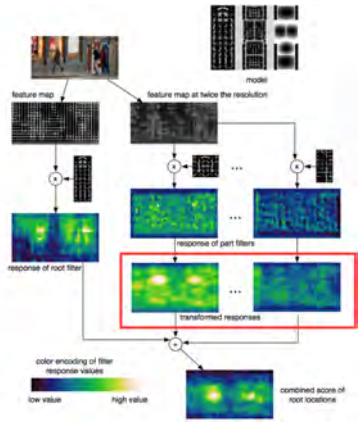


Figure 22: Step 3 of the detection pipeline

4. For every detection, we now sum these components to calculate the *detection score*:

$$F_0 + \prod_{i=1}^{n} F_i \cdot \phi(p_i, H) - \sum_{i=1}^{n} d_i(dx_i, dy_i, dx_i^2, dy_i^2)$$

5. These scores then represent the strength of the detection of the given object at each coordinate of the image; hence, we can plot the response scores throughout the image:

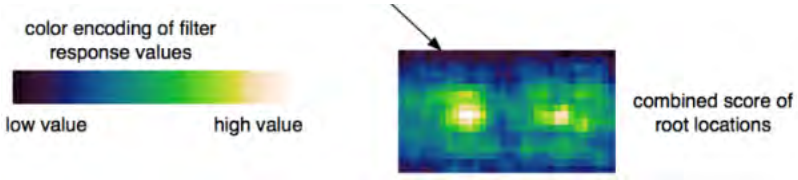


Figure 23: Response scores

7 DPM Detection Results

The Deformable Parts Model makes some key assumptions: an object is defined by a relationship between a global representation (e.g., a car) and representations of parts (e.g. wheels, or a bumper); that the strength of this detection increases with the decrease in deformation between the root and the parts; and that if a high response score is achieved, that object is indeed present (regardless of the potential presence of different categories of objects). Hence, DPM is vulnerable to error in situations where these assumptions are violated:

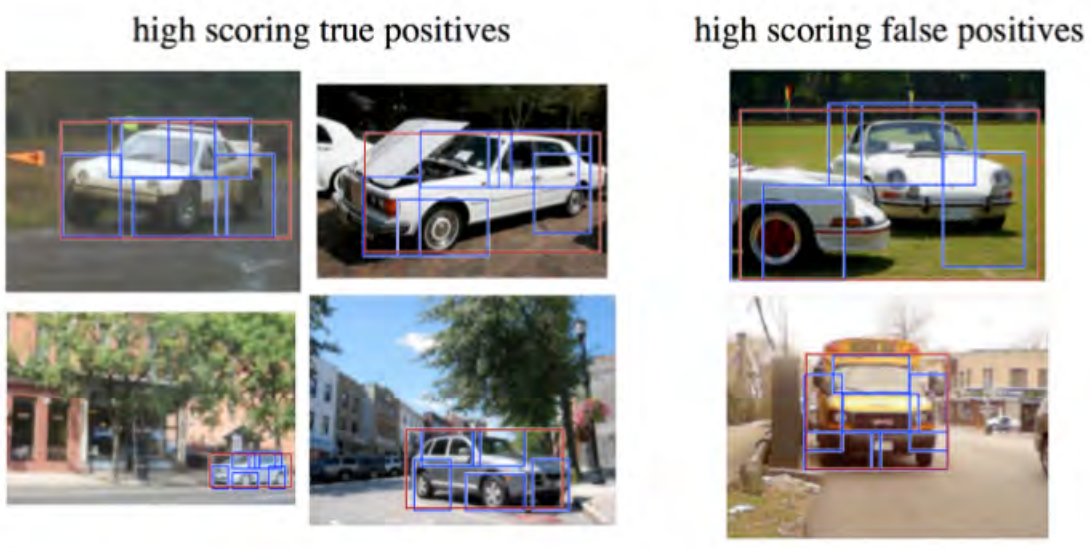


Figure 24: Typical errors for the DPM model

Note how in the top image on the right, DPM has successfully found sections matching the parts filters: wheels and a windshield. DPM has also successfully found one true positive for the global filter (the car in the background). However, DPM assumes that these parts are related to each other because they are spatially close (i.e., they fit the deformation model), and that they correspond to the car identified as the global filter – when in reality, there is not one but two cars in the image providing the parts.

Similarly, with the bottom image, DPM indeed detects an object very close to a car. However, since it does not take into account that the object is even closer to being a bus than a car, and does not take

into account features explicitly *not* present in a car (e.g., the raised roof spelling "School bus"), DPM results in a wrong detection.

8 DPM Summary

Approach

- Manually selected set of parts: a specific detector is trained for each part
- Spatial model is trained on the *part* activations
- Joint likelihood is evaluated for these activations

Advantages

- Parts have an intuitive meaning
- Standard detection approaches can be used for each part
- Works well for specific categories

Disadvantages

- Parts need to be selected manually
- Semantically motivated parts sometimes don't have a simple appearance distribution
- No guarantee that some important part hasn't been missed
- When switching to another category, the model has to be rebuilt from scratch

Notably, the Deformable Parts Model was state-of-the-art 3-4 years ago, but has since fallen out of favor. Its "fully-connected" extension, pictured below, is particularly no longer used in practice:

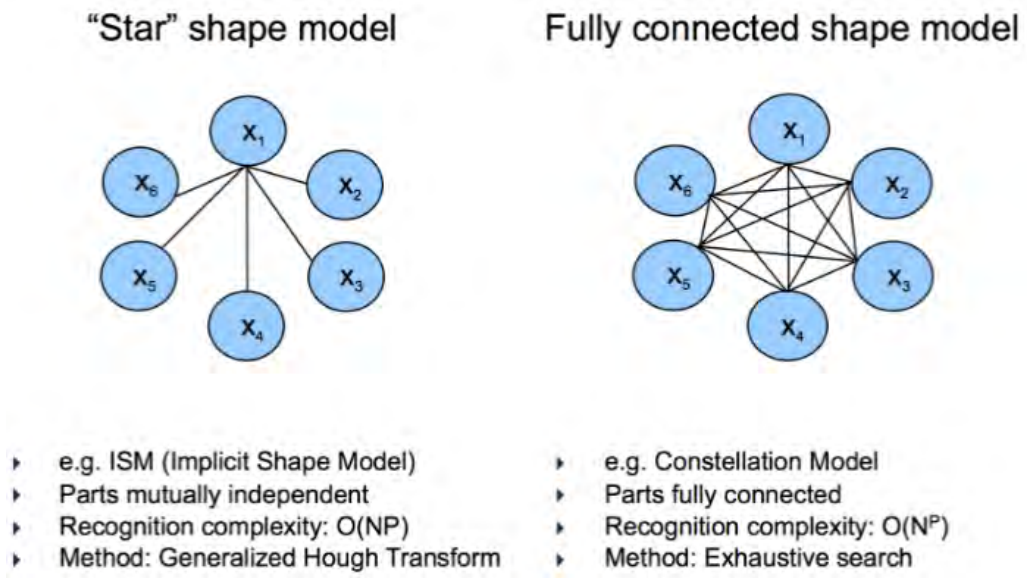


Figure 25: DPM extension: fully-connected shape model

References

[1] Bill Triggs Dalal Navneet. Histograms of oriented gradients for human detection. *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, 1, 2005.

[2] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results. http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[3] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[4] Robert A. Elshlager Martin A. Fischler. The representation and matching of pictorial structures. *IEEE Transactions on Computers (Volume: C-22, Issue: 1, Jan. 1973)*, 1973.

[5] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

Lecture 16: Recognizing Objects by Parts

Tyler Dammann, Patrick Mogan, Qiqi Ren, Cody Stocker, Evin Yang
Department of Computer Science
Stanford University
Stanford, CA 94305
{tdammann, pjmogan, qiqiren, cstock2, evin}@stanford.edu

1 Introduction

One overarching goal of computer vision is to not only determine an object's class, but also relevant information about that object. Motivating examples include:

- Determining the number of calories in a sushi using its image?
- Using the image of a tiger to identify the safest course of actio?
- Given a chair, can we find out where we can buy it online?
- using computer vision and images of mushrooms to identify if they are edible?

In general, computer vision should be able to give helpful information about an object beyond just basic characteristics.

2 What can computers recognize today?

Computer vision algorithms can easily detect and recognize faces in images.

Object identification software such as Google Goggles can find very specific kinds of objects (e.g. logos, landmarks, books, text, contact info, and artwork), but can only find exact matches. Finding a generic object is much harder for today's systems.

3 What's next to work on?

A principle milestone in computer vision yet to be reached is universal class recognition. That is, we would like an algorithm that could, for an image of any object, identify its class. Examples include:

- For an image of an orange mug, we would like coffee mugs to be the result, but image search found similar images by looking for something orange in the center – not mugs.
- Given an image of someone with a gas pump, we'd like to recognize that the gas pump is in the picture, but Google image search gives us images of people standing in same area of picture, maybe wearing the same color clothing.

The PASCAL VOC challenge[1] taught models how to classify 20 classes of objects. However, a particular model can only recognize a finite number of object classes. A model cannot recognize an arbitrary object, such as a coffee mug, if it is outside the model's set of training classes. We want to be able to recognize everything, but there are a lot of "things", and the agreed-upon number of "things" is increasing over time, so it is hard to decide which ones to focus on. In addition, there is not even an agreed-upon number of "things" in the universe – there are 60K+ product categories on eBay, 80K+ English nouns on WordNet, 3.5M+ unique tags on Flickr, and 4.1M+ articles on Wikipedia.

4 Big Data from the Internet

Because of the sheer amount of available data, the Internet should be able to teach us a huge number of things. Internet traffic has been steadily increasing, and vast majority (86%) is visual data.

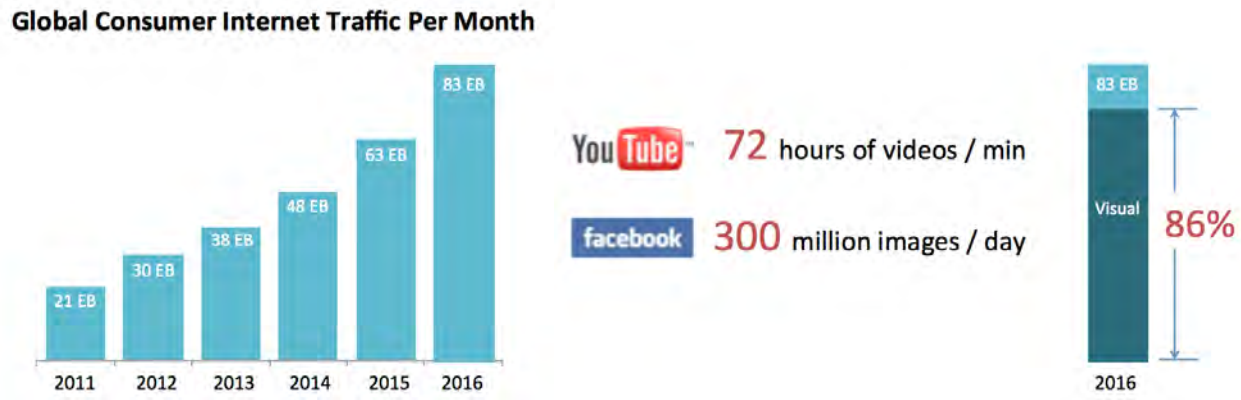


Figure 1: Global Internet Traffic. Source: Lecture 16

Visual data is known as the dark matter of the Internet because we can't autoclassify or auto-analyze it. Trying to categorize a photoshopped "pitbullfrog" shows that machines struggle to generalize, improvise, and extrapolate with the ease that humans do. Thus, images are not enough; just as important is leveraging the crowd, which can provide answers to questions that machines would otherwise have difficulty tackling themselves, such as in the following example:

Figure 2: Example Image Recognition Problem. Source: Lecture 16

Vitally, behind the big data is all of the users who are contributing their domain knowledge. When an individual cannot find the answer to a problem, it is easy to find others that can help. Image recognition systems, on the other hand, do not necessarily achieve this right now. The Internet gives an environment in which these two resources can be combined.

5 ImageNet and Confusion Matrices

Since models are limited by the number of the classes in the training set, one possible method of advancing universal class recognition is to create datasets with many more than the 20 classes the PASCAL VOC provided.

ImageNet is a large image database created by Jia Deng in 2009 that has 22,000 categories and 14 million images. It has become so important to computer vision that most projects now train on ImageNet before tackling other challenges. Other well-known databases include: Caltech101 (9K images, 101 categories), LabelMe (30k images), SUN (131K images).

Deng applied four top classification methods from PASCAL VOC to ImageNet, but found that increasing the number of categories on which the models trained decreased their accuracy greatly. He plotted his findings as the following confusion matrix:

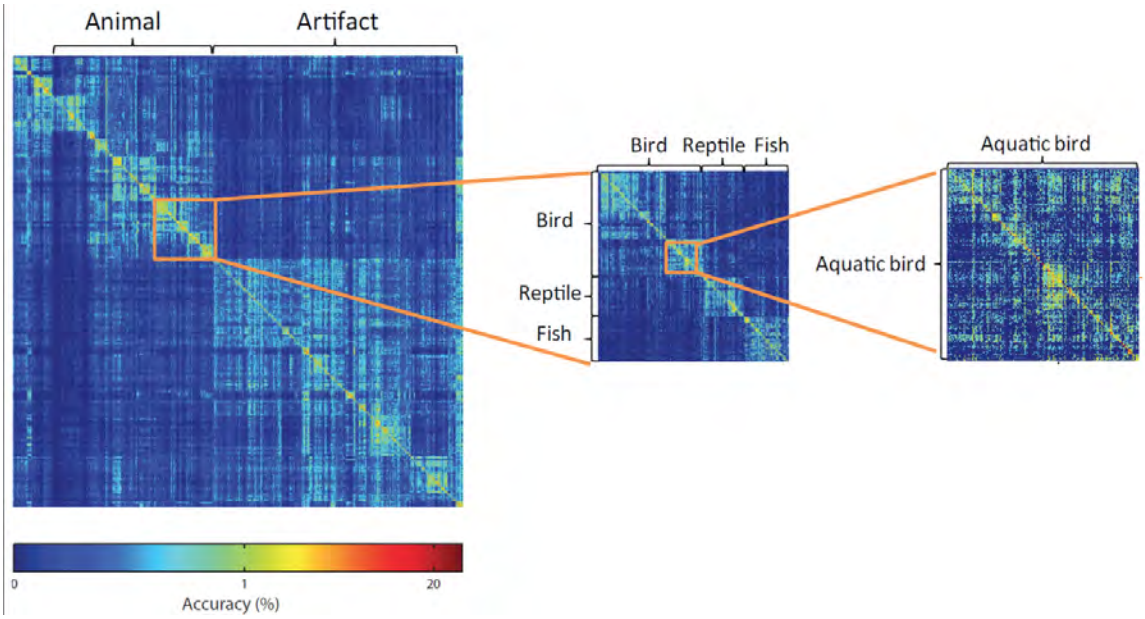


Figure 3: ImageNet Confusion Matrix. Source: Lecture 16

A confusion matrix plots categories on the x and y axis and measures the degrees to which objects in a category are correctly or incorrectly classified. If we had a perfect detector, the only bright areas would be along the center diagonal line; this means everything is correctly classified (i.e., no misclassification).

We see from this matrix that models are more prone to making classification errors when presented with items whose categories are very similar; for example, models struggle with discerning between different types of aquatic birds, while it can more easily categorize birds versus dogs. In other words, distinguishing between "fine-grained" categories is very difficult, since the distance between those categories is much smaller than between larger categories.

6 Challenges and Solutions

6.1 Semantic Hierarchy

One method for solving the issue of correctly classifying similar classes without making wrong guesses is the idea of a "semantic hierarchy". An example can be seen in Figure 4. Essentially, a tree is created, where every child is a more specific subclass of the parent. As the category gets more specific, uncertainty increases. The system will attempt to be as specific as possible (as low down on the tree as possible) without an unacceptable probability of misclassifying the object. This concept is called "hedging" - the system tries to identify where in the uncertainty tree to make a guess such that it is as informative as possible with few mistakes.

To formally define this problem, we will assume that the training and test set have the same distribution. In addition, we will assume that we have access to a base classifier g that gives

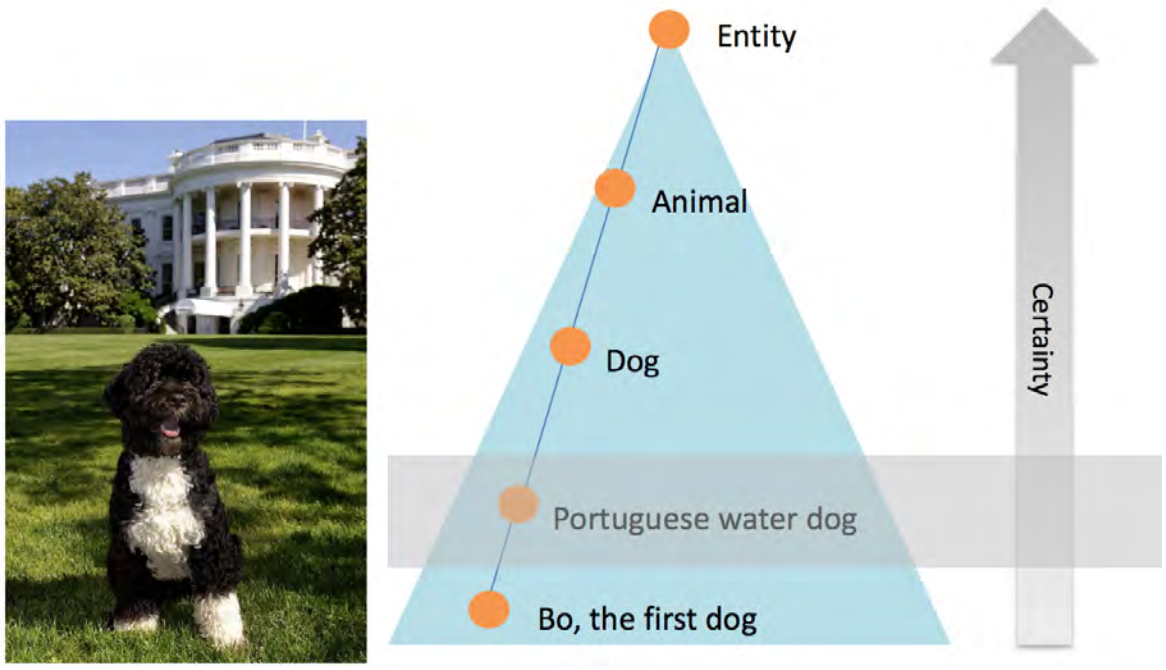


Figure 4: Example Semantic Hierarchy. Source: Lecture 16

a posterior probability on the hierarchy. We then define a reward function $R(f)$. We make $R(f)$ to have higher values at points farther down the hierarchical tree (when it classifies objects more specifically). Then, we also define an expected accuracy function $A(f)$. This function decreases as we move down the tree (since we are less certain about more specific guesses). Our problem statement is then to maximize $R(f)$ subject to the constraint that $A(f) \geq 1 - \epsilon$, where ϵ is a predetermined constant that represents the allowable error of our classifier over all of our examples.

However, potential non-optimal solutions (in terms of $R(f)$) can arise given this method. In order to fix this, we can define a global, fixed, scalar parameter $\lambda \geq 0$. For each node, we add λ to the reward value of that node, then normalize the posterior distribution. The process is then as follows:

1. Pick a λ.
2. Find the decision rule f with λ.
3. Measure the performance on the validation set.
4. Check if $A \approx 1 - \epsilon$. Repeat from step one if necessary.

We can use binary search to find the best λ quickly.

6.2 Fine-grained Classes

Existing work selects features from all possible locations in an image, but it can fail to find the right feature. For example, the difference between the Cardigan Welsh Corgi and the Pembroke Welsh Corgi is the tail. The computer may be unable to discover that this is the distinguishing feature. The easiest way to solve this type of problem is through crowd-sourcing.

What seems like a simple solution, however, is actually difficult – what is the best method for asking a crowd which features distinguish classes of images?

Bubble study: In order to detect the difference between smiling and neutral faces, we display only small bubbles of an image to people in an experiment to determine which bubbles allow them to detect when the image is of a person smiling. An example of how the bubble method works can be seen in figure 5. However, this method is costly and time consuming.

Another idea is to ask people to annotate images themselves (to simply point out where the im-

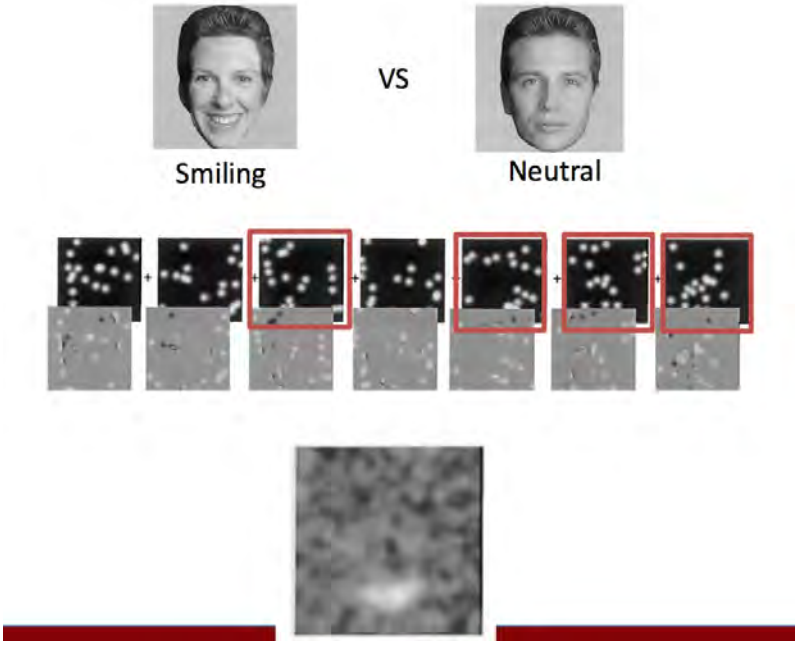


Figure 5: Bubble Method. Source: Lecture 16

portant features are). However, this method has no way of easily assuring the quality of these annotations.

Crowd-sourced bubble games: online bubble games are similar in nature to the bubble study, but are fun and have monetary rewards for players. Notice that this solves both problems of the above methods – it is cost effective to have people play, while quality is assured by the reward system of the game. Two examples of games include the following:

- Peekaboom: Peekaboom was a two-player game in which one player reveals parts of an image, while the other player attempts to guess what the image represents. This game worked well for finding important parts of an image, but was not good for distinguishing between fine-grained classes (since often the important parts of the classes are the same).
- The Bubbles Game: This game, created by Jia Deng, is an online game in which players are given example images of two classes. Then, a blurred-out image is shown, and the player attempts to guess which class the blurred image belongs to, while revealing only small parts of the image. The less of the image revealed, the more reward is given to the player. This game is much better for finding differences between fine-grained classes, and it led to bubble heatmaps that pinpointed the features that distinguished fine-grained classes, as seen in figure 6. The creators of the game pooled bubbles from multiple images in a category together and the resulting information was provided to a model, which performed with higher precision than other models at the time.

References:

[1] Everingham, M. and Van-Gool, L. and Williams, C. K. I. and Winn, J. and Zisserman, A. *The PASCAL Visual Object Classes Challenge 2012. (VOC2012) Results*. Retrieved from: http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

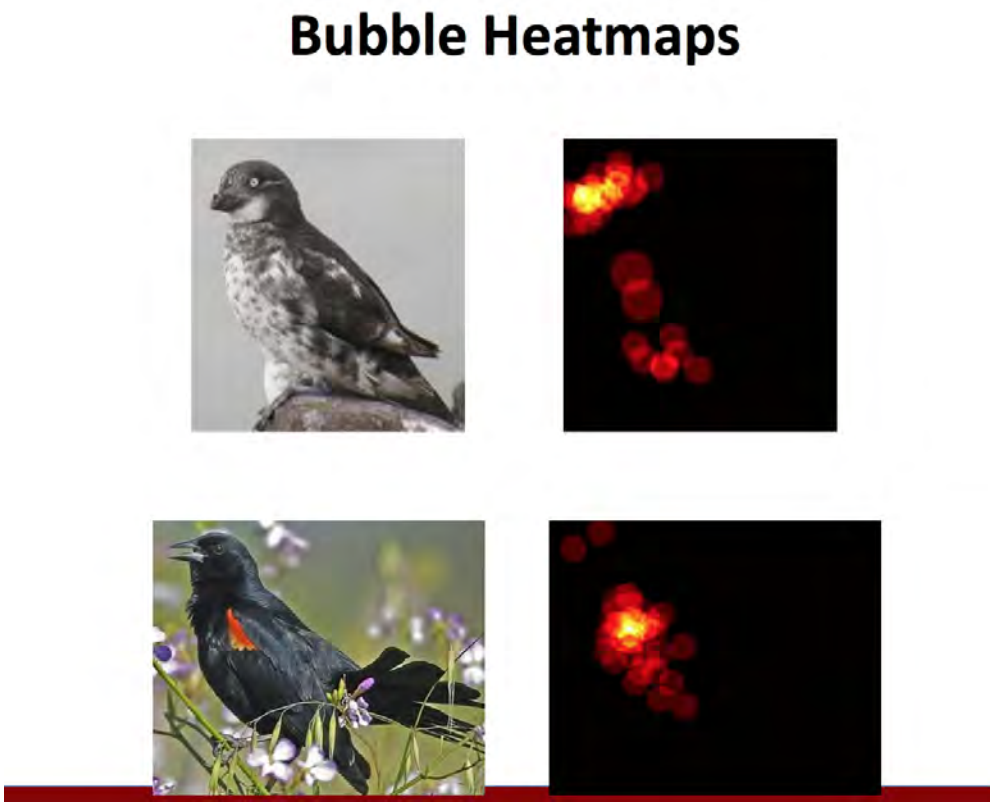


Figure 6: Bubble Heatmaps. Source: Lecture 16

Lecture #17: Motion

Kevin Chavez, Ben Cohen-Wang, Garrick Fernandez, Noah Jackson, Will Lauer
Department of Computer Science
Stanford University
Stanford, CA 94305
{kechavez, bencw, garrick, noahjax, wlauer}@cs.stanford.edu

1 Introduction

In this class so far we have learned a variety of tools that enable us to detect key points, recognize objects, and use segmentation in images. However, in many cases we want to be able to perform similar operations on video. Specifically, we are often interested not only in the location of certain objects, but also the movement of these objects over time. This lecture focuses on how we can apply previously covered techniques along with new methods to effectively track the motion of pixels across many images, with applications in areas such as self-driving cars, robots, and security systems to name a few.

2 Optical Flow and Key Assumptions

2.1 Optical Flow

Put simply, optical flow is the movement of pixels over time. The goal of optical flow is to generate a motion vector for each pixel in an image between t_0 and t_1 by looking at two images I_0 and I_1. By computing a motion vector field between each successive frame in a video, we can track the flow of objects, or, more accurately, "brightness patterns" over extended periods of time. However, it is important to note that while optical flow aims to represent the motion of image patterns, it is limited to representing the *apparent* motion of these patterns. This nuanced difference is explained more in depth in the **Assumptions and Limitations** section.

2.2 Assumptions and Limitations

2.2.1 Apparent Motion

Given a two dimensional image, optical flow can only represent the *apparent* motion of brightness patterns, meaning that the movement vectors of optical flow can be the result of a variety of actions. For instance, variable lighting can cause strong motion vectors on static objects, and movement into or out of the frame cannot be captured by the 2D motion vectors of optical flow. One example of an issue poorly dealt with by optical flow is the aperture problem.

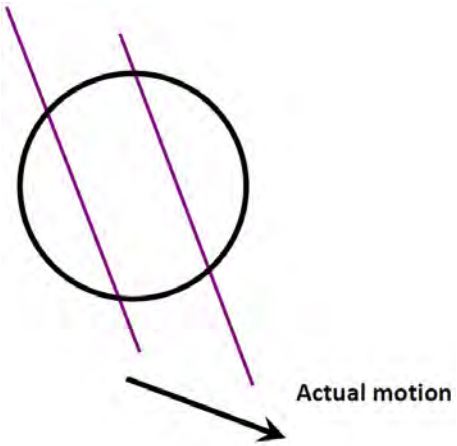


Figure 1: In the aperture problem, the line appears to have moved to the right when only in the context of the frame, but the true motion of the line was down and to the right. The aperture problem is a result of optical flow being unable to represent motion along an edge–an issue that can lead to other errors in motion estimation as well.

2.2.2 Brightness Consistency

As optical flow can only represent apparent motion, to correctly track the motion of points on an image we must assume that these points remain at the same brightness between frames. The equation for this brightness consistency equation is as follows

$$I(x, y, t-1) = I(x + u(x, y), y + v(x, y), t)$$

where u(x,y) represents the horizontal motion of a point and v(x,y) represents the vertical motion.

2.2.3 Small Motion

Optical flow assumes that points do not move very far between consecutive images. This is often a safe assumption, as videos are typically comprised of 20+ frames per second, so motion between individual frames is small. However, in cases where the object is very fast or close to the camera this assumption can still prove to be untrue. To understand why this assumption is necessary, we must consider the **Brightness Consistency** equation defined above. When trying to solve this equation, it is useful to linearize the right side using a Taylor expansion. This yields

$$I(x + u(x, y), y + v(x, y), t) \approx I(x, y, t-1) + I_x \cdot u(x, y) + I_y \cdot v(x, y) + I_t$$

Linearizing in this way allows us to solve for the u and v motion vectors we want, but in this case we have only included the first order Taylor series terms. When motion is large between frames, these terms do a poor job of capturing the entire motion, thus leading to inaccurate u,v. More information about higher order derivative constraints can be found in references [1], page 12.

2.2.4 Spatial Coherence

Spatial coherence is the assumption that nearby pixels will move together, typically because they are part of the same object. To see why this assumption is necessary, consider the equation for optical flow as defined above

$$I(x + u(x, y), y + v(x, y), t) \approx I(x, y, t-1) + I_x \cdot u(x, y) + I_y \cdot v(x, y) + I_t$$
$$I(x + u(x, y), y + v(x, y), t) - I(x, y, t-1) = I_x \cdot u(x, y) + I_y \cdot v(x, y) + I_t$$

Giving us

$$I_x \cdot u + I_y \cdot v + I_t \approx 0$$
$$\bigtriangledown I \cdot [u\, v]^T + I_t = 0$$

Ignoring the meaning of this derivation for the moment, it is clear that we do not have enough equations to find both u and v at every single pixel. Assuming that pixels move together allows us to use many more equations with the same [u v], making it possible to solve for the motion of pixels in this neighborhood.

3 Lucas-Kanade

Recovering image motion given by (u, v) in the above equation requires at least two equations per pixel. To achieve this, the Lucas-Kanade [1] technique for image tracking relies on an additional constraint — spatial coherence.

The spatial coherence constraint is applied to a pixel using a window of size $k \times k$. The assumption is that the neighboring pixels in this window will have the same (u, v). For example, in a $5x5$ window the following equations apply:

$$0 = I_t(\mathbf{p_i}) + \nabla I(\mathbf{p_i}) \cdot [u \quad v]$$

$$\begin{bmatrix} I_x(\mathbf{p_1}) & I_y(\mathbf{p_1}) \\ I_x(\mathbf{p_2}) & I_y(\mathbf{p_2}) \\ \vdots & \vdots \\ I_x(\mathbf{p_{25}}) & I_y(\mathbf{p_{25}}) \end{bmatrix}$$

This produces an overly-constrained system of linear equations of the form $Ad = b$. Using a least squares method for solving over-constrained systems, we reduce the problem to solving for d in $(A^T A)d = A^T b$. More explicitly the system to solve is reduced to

$$\underbrace{\begin{bmatrix} \sum I_x I_x & \sum I_x I_y \\ \sum I_y I_x & \sum I_y I_y \end{bmatrix}}_{A^T A} \begin{bmatrix} u \\ v \end{bmatrix} = - \underbrace{\begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix}}_{A^T b}$$

3.1 Condition for an Existing Solution

In order to solve the system following conditions should hold:

- $A^T A$ should be invertible
- $A^T A$ should not be too small due to noise.

  Eigenvalues λ_1 and λ_2 of $A^T A$ should not be too small
- $A^T A$ should be well-conditioned

  i.e λ_1/λ_2 should not be too large (for $\lambda_1 > \lambda_2$)

3.2 Geometric Interpretation

It should be evident that the least squares system of equations above produce a second moment matrix $M = A^T A$. In fact, this is the Harris matrix for corner detection.

$$A^T A = \begin{bmatrix} \sum I_x I_x & \sum I_x I_y \\ \sum I_y I_x & \sum I_y I_y \end{bmatrix} = \sum \begin{bmatrix} I_x \\ I_y \end{bmatrix} [I_x \quad I_y] = \sum \nabla I (\nabla I)^T = M$$

We can relate the conditions above for solving the motion field $[u \quad v]$ to tracking corners detected by the Harris matrix M. In particular, the eigenvectors and eigenvalues of $M = A^T A$ relate to the direction and magnitude of a possible edge in a region.

Using this interpretation, it is apparent that an ideal region for Lucas-Kanade optical flow estimation is a corner. Visually, if λ_1 and λ_2 are too small this means the region is too "flat". If $\lambda_1 >> \lambda_2$, the method suffers from the aperture problem, and may fail to solve for correct optical flow.

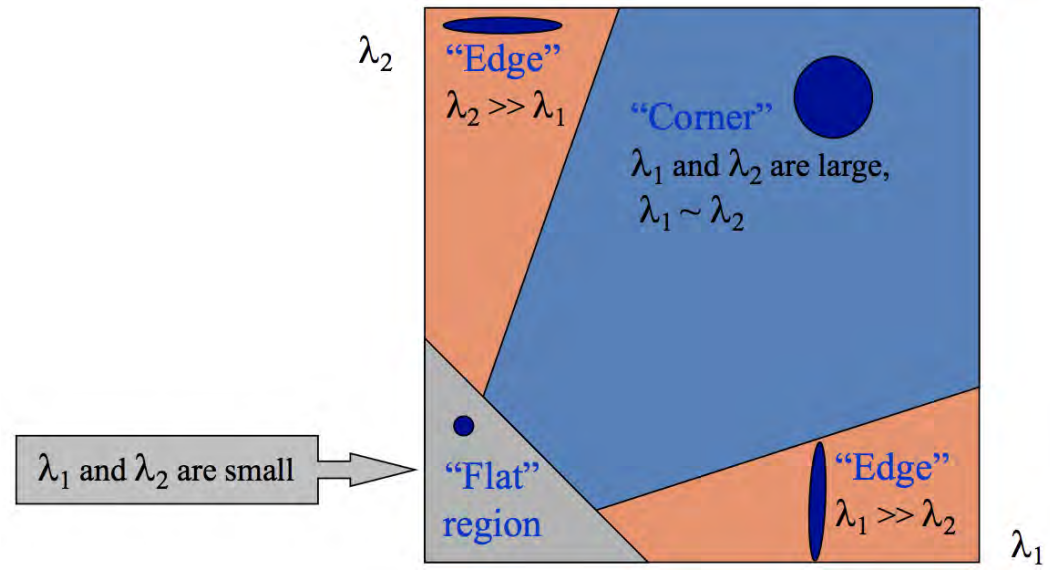


Figure 2: Conditions for a solvable matrix $A^T A$ may be interpreted as different edge regions depending on the relation between λ_1 and λ_2. Corner regions produce more optimal conditions.

Figure 3: Example of regions with large λ_1 and small λ_2 (left), small λ_1 and smallλ_2 (center, low texture region), large λ_1 and large λ_2 (right, high texure region)

3.3 Error in Lucas-Kanade

The Lucas-Kanade method is constrained under the assumptions of optical flow. Supposing that $A^T A$ is easily invertible and that there is not much noise in the image, errors may still arise when:

- Brightness constancy is not satisfied, meaning that a pixel may change intensity from different time steps.
- The motion is not small or and does not change gradually over time.
- Spatial coherence is not satisfied, meaning neighboring pixels do not move alike.
  This may arise due to in an inappropriately sized window (choosing bad k).

3.4 Improving Accuracy

From the many assumptions made above, Lucas-Kanade can improve its accuracy by including the higher order terms previously dropped in the Taylor expansion approximation for the brightness constancy equation. This loosens the assumptions of small motion and more accurately reflects optical flow. Now, the problem to be solved is:

$$I(x+u, y+v) = I(x,y) + I_x u + I_y v + \text{higher order terms} - I_{t-1}(x,y)$$

This is a polynomial root finding problem and can be solved with an iterative approach using Newton's method.

In summary, the refined Iterative Lucas-Kanade Algorithm may be applied as:

1. Estimate velocity at each pixel by solving Lucas-Kanade equations.
2. Warp $I(t-1)$ towards $I(t)$ using the estimated flow field and image warping techniques.
3. Repeat until convergence.

4 Horn-Schunk

4.1 Horn-Schunk Method for Optical Flow

The Horn-Schunk method for computing optical flow formulates flow as the following global energy function which should be minimized with respect to $u(x,y)$ and $v(x,y)$.

$$E = \int\int [(I_x u + I_y v + I_t)^2 + \alpha^2(||\nabla u||^2 + ||\nabla v||^2)]\mathrm{d}x\mathrm{d}y$$

The first term of this energy function reflects the brightness constancy assumption, which states that the brightness of each pixel remains the same between frames, though the location of the pixel may change. According to this assumption, $I_x u + I_y v + I_t$ should be zero. The square of this value is included in the energy function to ensure that this value is as close to zero as possible, and thus u and v comply with the brightness constancy assumption.

The second term of this energy function reflects the small motion assumption, which states that the points move by small amounts between frames. The squares of the magnitudes of u and v are included in the energy function to encourage smoother flow with only small changes to the position of each point. The regularization constant α is included to control smoothness, with larger values of α leading to smoother flow.

To minimize the energy function, we take the derivative with respect to u and v and set to zero. This yields the following two equations

$$I_x(I_x u + I_y v + I_t) - \alpha^2 \Delta u = 0$$
$$I_y(I_x u + I_y v + I_t) - \alpha^2 \Delta v = 0$$

where $\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$ is called the Lagrange operator, which in practice is computed as

$$\Delta u(x,y) = \bar{u}(x,y) - u(x,y)$$

where $\bar{u}(x,y)$ is the weighted average of u in a neighborhood around (x,y). Substituting this expression for the Lagrangian in the two equations above yields

$$(I_x^2 + \alpha^2)u + I_x I_y v = \alpha^2 \bar{u} - I_x I_t$$
$$I_x I_y u + (I_y^2 + \alpha^2)v = \alpha^2 \bar{v} - I_y I_t$$

which is a linear equation in u and v for each pixel.

4.2 Iterative Horn-Schunk

Since the solution for u and v for each pixel (x,y) depends on the optical flow values in a neighborhood around (x,y), to obtain accurate values for u and v we must recalculate u and v iteratively once the neighbors have been updated. We can iteratively solve for u and v using

$$u^{k+1} = \bar{u}^k - \frac{I_x(I_x \bar{u}^k + I_y \bar{v}^k + I_t)}{\alpha^2 + I_x^2 + I_y^2}$$
$$v^{k+1} = \bar{v}^k - \frac{I_y(I_x \bar{u}^k + I_y \bar{v}^k + I_t)}{\alpha^2 + I_x^2 + I_y^2}$$

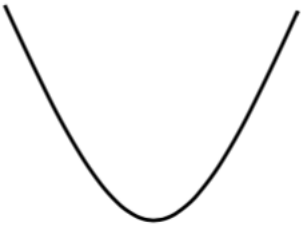
where $\bar{u}^k$ and $\bar{v}^k$ are the values for $\bar{u}$ and $\bar{v}$ calculated during the k'th iteration, and u^{k+1} and v^{k+1} are the updated values for u and v for the next iteration.

4.3 Smoothness Regularization

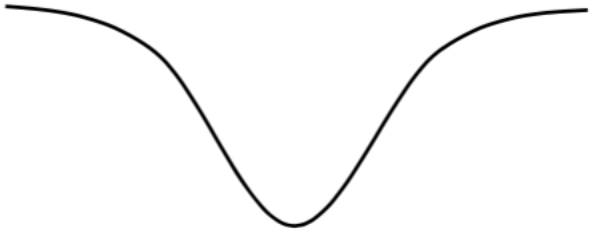
The smoothness regularization term $||\nabla u||^2 + ||\nabla v||^2$ in the energy function encourages minimizing change in optical flow between nearby points. With this regularization term, in texture free regions there is no optical flow, and on edges, points will flow to the nearest points, solving the aperture problem.

4.4 Dense Optical Flow with Michael Black's Method

Michael Black extended the Horn-Schunk method by replacing the regularization term $||\nabla u||^2 + ||\nabla v||^2$ which is a quadratic function of the magnitudes of the gradients of u and v

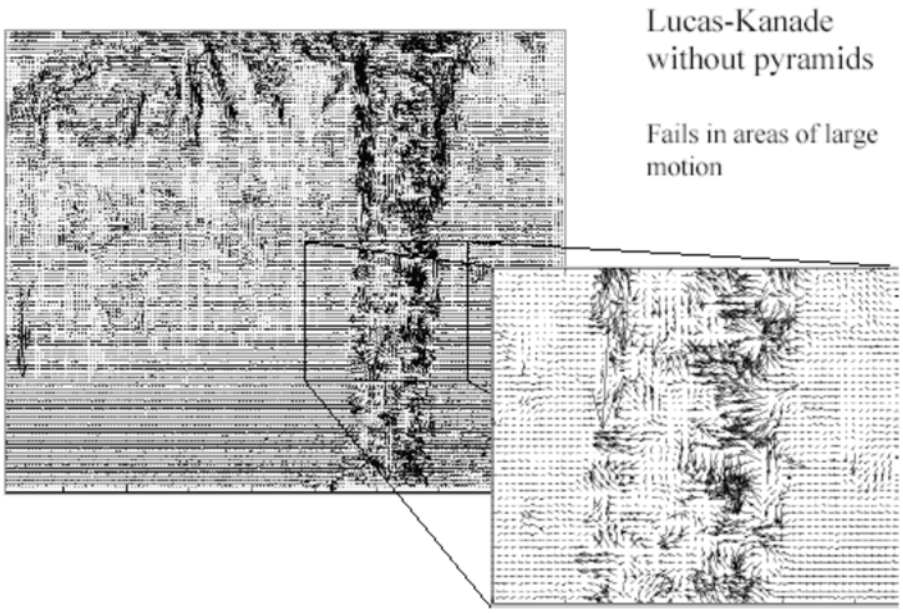


with the following function

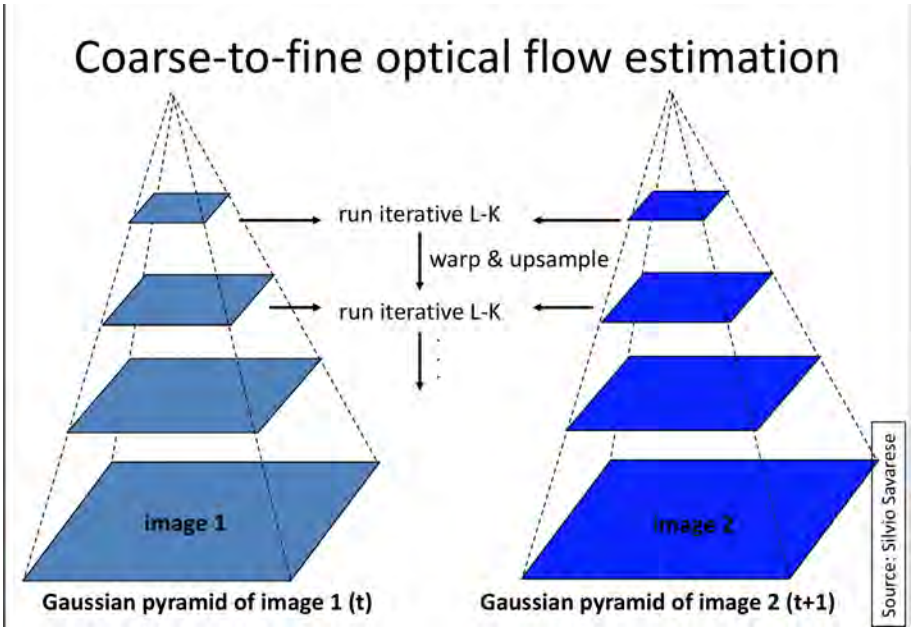


5 Pyramids for Large Motion

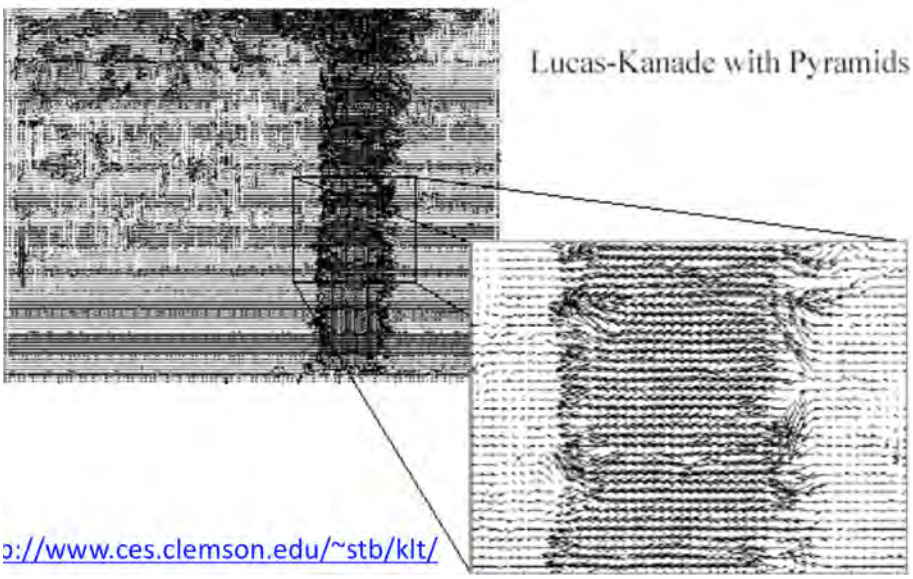
Revisiting Lucas-Kanade, recall that one of our original assumptions was that there would be small motion of points between consecutive frames. This assumption causes the algorithm to fall apart when dealing with large motion:



Notice in the graphic above, Lucas-Kanade can't find a consistent vector for the flow of the tree trunk. In order to correct for this, we can apply a tactic where we apply Lucas-Kanade iteratively to a lower-resolution version of the image, similar to how we created image pyramids for our sliding-window feature detector.



Now, when we try to find the flow vector, the small motion condition is fulfilled, as the downsampled pixels move less from frame to consecutive frame than pixels in the higher resolution image. Here is another example from the slides using Lucas-Kanade with pyramids:



Notice how the flow vectors now point mostly in the same direction, indicating that the tree trunk is moving in a consistent direction.

6 Common Fate

We can gain more information about an image by analyzing it through the the lens of common fate, which in this context is the idea that each pixel in a given segment of the image will move in a similar manner. Our goal is to identify the image segments, or "layers", that move together.

6.1 Identify Layers

We compute layers in an image by dividing the image into blocks and grouping based on the similarity of their affine motion parameters. For each block, finding the vector a that minimizes

$$Err(a) = \sum [I_x(a_1 + a_2x + a_3y) + I_y(a_4 + a_5x + a_6y) + I_t]^2$$

for all pixels (x, y) in each block.

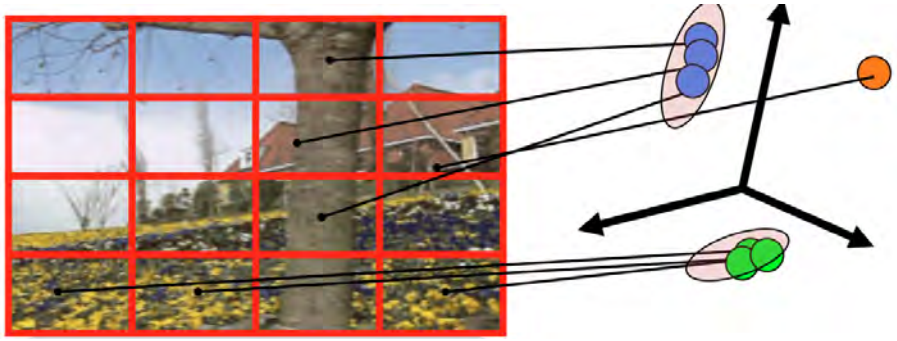
The above equation is derived from two parts: (1) the brightness constancy equation, and (2) the components of affine motion:

$$I_x u(x, y) + I_y v(x, y) + I_t \approx 0$$

I_x, I_y, I_t are the gradients of the image with respect to the x direction, y direction, and time, respectively. $u(x, y)$ and $v(x, y)$ are the components of affine motion in the horizontal and vertical directions:

$$u(x, y) = a_1 + a_2x + a_3y$$
$$v(x, y) = a_4 + a_5x + a_6y$$

From there, we map our parameter vectors a_i into motion parameter space and perform k-means clustering on the affine motion parameter vectors.



The final centers of the k-means clustering are the parameters $a_1 ... a_6$ that minimize the above error function, and the vectors a_i in each grouping correspond to the original blocks that should be grouped in a single layer. Intuitively, layers should be comprised of blocks that have similar parameters, as that implies their affine motion is similar.

References

[1] Bruce D Lucas, Takeo Kanade, et al. An iterative image registration technique with an application to stereo vision. 1981.

Lecture 18: Tracking

Khaled Jedoui, Jamie Xie, Michelle Guo, Nikhil Cheerla.
Department of Computer Science
Stanford University
Stanford, CA 94305
{thekej, jaimiex, ncheerla, mguo95}@stanford.edu

1 Introduction: What is tracking?

1.1 Definition

Visual tracking is the process of locating a moving object (or multiple objects) over time in a sequence.

1.2 Objective

The objective of tracking is to associate target objects and estimate target state over time in consecutive video frames.

1.3 Applications

Tracking has a variety of application, some of which are:

- Human-computer interaction: Eye Tracking Systems[1]
- Security and surveillance[3]
- Augmented reality[2]
- Traffic control: Road transportation systems[5]
- Medical imaging[4]

1.4 Challenges

Note, that tracking can be a time consuming process due to the amount of data that in video. Besides, tracking relies on object recognition algorithms for tracking, which might become more challenging and prone to failure for the following reasons:

- Variations due to geometric changes like the scale of the tracked object
- Changes due to illumination and other photometric aspects
- Occlusions in the image frame
- Motion that is non-linear
- Blurry videos or videos with bad resolution might make the recognition fail
- Similar objects in the scene

2 Feature Tracking

2.1 Definition

Feature tracking is the detection of visual feature points (corners, textured areas, ...) and tracking them over a sequence of frames (images).

2.2 Challenges of feature tracking

- Figure out which features can be tracked
- Efficiently track across frames – Some points may change appearance over time (e.g., due to rotation, moving into shadows, etc.)
- Drift: small errors can accumulate as appearance model is updated
- Points may appear or disappear: need to be able to add/delete tracked points

2.3 What are good features to track?

What kinds of image regions can we detect easily and consistently? We need a way that can measure "quality" of features from just a single image. Intuitively, we want to avoid smooth regions and edges. A solution for such a problem is to use Harris Corners. Detecting Harris corners as our key points to track guarantees small error sensitivity!

Once we detect the features we want to track, we can use optical flow algorithms to solve our motion estimation problem and track our features.

2.4 Example

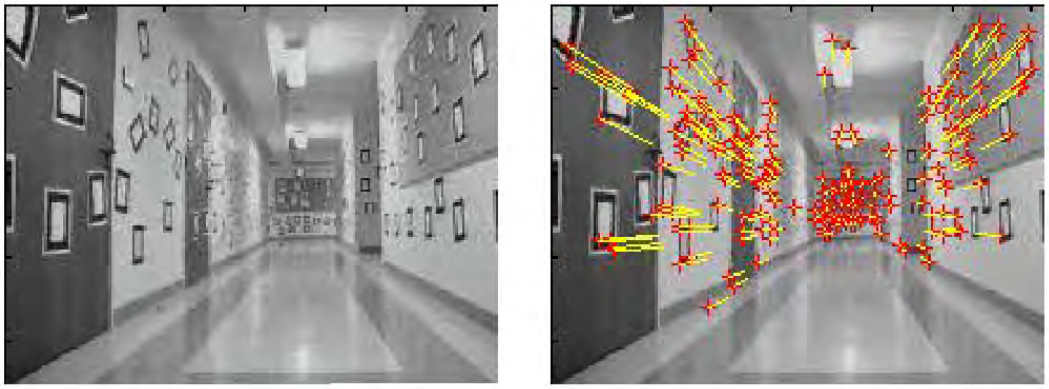


Figure 1: Courtesy of Jean-Yves Bouguet–Vision Lab, California Institute of Technology

2.5 Tracking methods

2.5.1 Simple Kanade–Lucas–Tomasi feature tracker

The Kanade–Lucas–Tomasi (KLT) feature tracker is an approach to feature extraction. KLT makes use of spatial intensity information to direct the search for the position that yields the best match. Its algorithm is:

1. Find a good point to track (harris corner)
   - Harris corner points have sufficiently large eigenvalues, so the optical flow equation is solvable.
2. For each Harris corner compute motion (translation or affine) between consecutive frames.
3. Link motion vectors in successive frames to get a track for each Harris point

- If the patch around the new point differs sufficiently from the old point, we discard these points.

4. Introduce new Harris points by applying Harris detector at every (10 or 15) frames
5. Track new and old Harris points using steps 2-3

In the following frames from tracking videos, arrows represent the tracking motion of the harris corners.

Figure 2: Frames from fish-tracking video, courtesy of Kanade

Figure 3: Frames from man-tracking video, courtesy of Kanade

3 2D Transformations

3.1 Types of 2D Transformations

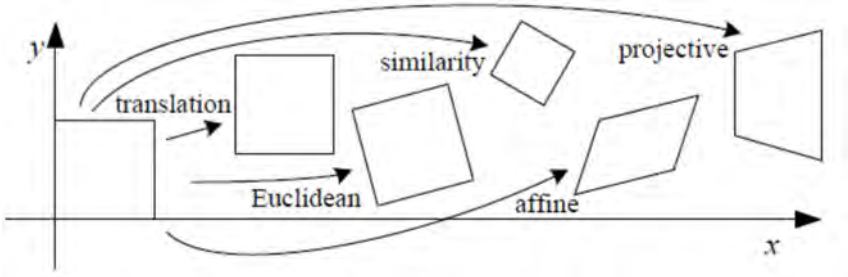


Figure 4: Types of 2D transformations.

There are several types of 2D transformations. Choosing the correct 2D transformations can depend on the camera (e.g. placement, movement, and viewpoint) and objects. A number of 2D transformations are shown in Figure 3.1. Examples of 2D transformations include:

- **Translation Transformation**. (e.g. Fixed overhead cameras)
- **Similarity Transformation**. (e.g. Fixed cameras of a basketball game)
- **Affine Transformation**. (e.g. People in pedestrian detection)
- **Projective Transformation**. (e.g. Moving cameras)

In this section we will cover three common transformations: translation, similarity, and affine.

3.2 Translation

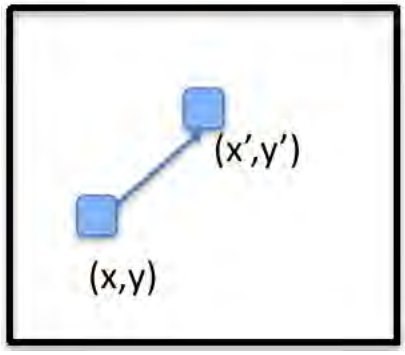


Figure 5: Translation

Translational motion is the motion by which a body shifts from one point in space to another. Assume we have a simple point m with coordinates (x, y). Applying a translation motion on m shifts it from (x, y) to (x', y') where

$$\begin{aligned} x' &= x + b_1 \\ y' &= y + b_2 \end{aligned} \tag{1}$$

We can write this as a matrix transformation using homogeneous coordinates:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & 0 & b_1 \\ 0 & 1 & b_2 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{2}$$

Let W be the above transformation defined as:

$$W(\boldsymbol{x}; \boldsymbol{p}) = \begin{pmatrix} 1 & 0 & b_1 \\ 0 & 1 & b_2 \end{pmatrix} \tag{3}$$

where the parameter vector is $p = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$.

Taking the partial derivative of W with respect to $\boldsymbol{p}$ we get:

$$\frac{\partial W}{\partial \boldsymbol{p}}(\boldsymbol{x}; \boldsymbol{p}) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{4}$$

This is called the Jacobian.

3.3 Similarity Motion

Similarity motion is a rigid motion that includes scaling and translation.

We can define the similarity as:

$$\begin{aligned} x' &= ax + b_1 \\ y' &= ay + b_2 \end{aligned} \tag{5}$$

The similarity transformation matrix W and parameters p are defined as the following:

$$\begin{aligned} W &= \begin{pmatrix} a & 0 & b_1 \\ 0 & a & b_2 \end{pmatrix} \\ \boldsymbol{p} &= \begin{pmatrix} a & b_1 & b_2 \end{pmatrix}^T \end{aligned} \tag{6}$$

The Jacobian of the similarity transformation is then:

$$\frac{\partial W}{\partial \boldsymbol{p}}(\boldsymbol{x}; \boldsymbol{p}) = \begin{pmatrix} x & 1 & 0 \\ y & 0 & 1 \end{pmatrix} \tag{7}$$

3.4 Affine motion

Affine motion includes scaling, rotation, and translation. We can express this as the following:

$$\begin{aligned} x' &= a_1 x + a_2 y + b_1 \\ y' &= a_3 x + a_4 y + b_2 \end{aligned} \tag{8}$$

The affine transformation can be described with the following transformation matrix W and parameters p:

$$\begin{aligned} W &= \begin{pmatrix} a_1 & a_2 & b_1 \\ a_3 & a_4 & b_2 \end{pmatrix} \\ p &= (a_1 \quad a_2 \quad b_1 \quad a_3 \quad a_4 \quad b_2)^T \end{aligned} \tag{9}$$

Finally, the Jacobian for affine motion is the following:

$$\frac{\partial W}{\partial \boldsymbol{p}}(\boldsymbol{x}; \boldsymbol{p}) = \begin{pmatrix} x & y & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x & y & 1 \end{pmatrix} \tag{10}$$

4 Iterative KLT tracker

4.1 Problem formulation

Given a video sequence, find the sequence of transforms that maps each frame to the next frame. Should be able to deal with arbitrary types of motion, including object motion and camera/perspective motion.

4.2 Approach

This approach differs from the simple KLT tracker by the way it links frames: instead of using optical-flow to link motion vectors and track motion, we directly solve for the relevant transforms using feature data and linear approximations. This allows us to deal with more complex (such as affine and projective) transforms and link objects more robustly.

Steps:

1. First, use Harris corner detection to find the features to be tracked.
2. For each feature at location $x = [x, y]^T$: Choose a feature descriptor and use it to create an initial template for that feature (likely using nearby pixels): $T(x)$.
3. Solve for the transform p that minimizes the error of the feature description around $x_2 = W(x; p)$ (your hypothesis for where the feature's new location is) in the next frame. In other words, solve the equation
$$\sum_x [T(W(x; p)) - T(x)]^2$$
4. Iteratively reapply this to link frames together, storing the coordinates of the features as the transforms are continuously applied. This should give you a measure of how objects move through frames.
5. Just as before, every 10-15 frames introduce new Harris corners to account for occlusion and "lost" features.

4.3 Math

We can in fact analytically derive an approximation method for finding p (in Step 3). Assume that you have an initial guess for p, p_0, and $p = p_0 + \Delta p$.

Now,

$$E = \sum_x [T(W(x; p)) - T(x)]^2 = \sum_x [T(W(x; p_0 + \Delta p)) - T(x)]^2$$

But using the Taylor approximation, we see that this error term is roughly equal to :

$$E \approx \sum_x [T(W(x;p_0)) + \nabla T \frac{\partial W}{\partial p} \Delta p - T(x)]^2$$

To minimize this term, we take the derivative with regard to p_0 and set it equal to 0, then solve for p_0.

$$\frac{\partial E}{\partial p} \approx \sum_x [\nabla T (\frac{\partial W}{\partial p})^T][T(W(x;p_0)) + \nabla T \frac{\partial W}{\partial p} \Delta p - T(x)] = 0$$

$$\Delta p = H^{-1} \sum_x [\nabla T \frac{\partial W}{\partial p}^T][T(x) - T(W(x;p_0))]$$

, where H is $\sum_x [\nabla T \frac{\partial W}{\partial p}]^T [\nabla T \frac{\partial W}{\partial p}]$

By iteratively setting $p_0 = p_0 + \Delta p$, we can eventually converge on an accurate, error-minimizing value of p, which tells us what the transform is.

4.4 Link to Harris Corner Detection

For translation motion,

$$\frac{\partial W}{\partial p}(x;p) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

, so it is easy to show that

$$H = \begin{pmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{pmatrix}$$

. However, the Harris corner detector assumes that whenever H has large eigenvalues (i.e it is stably invertible), then the point within the window is a corner. Thus, corners tend to be good features for computing translations, precisely because the resulting matrices produced by corners are stably invertible.

References

[1] S. Chandra, G. Sharma, S. Malhotra, D. Jha, and A. P. Mittal. Eye tracking based human computer interaction: Applications and their uses. pages 1–5, Dec 2015.

[2] A. I. Comport, E. Marchand, M. Pressigout, and F. Chaumette. Real-time markerless tracking for augmented reality: the virtual visual servoing framework. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):615–628, July 2006.

[3] A. Hampapur, L. Brown, J. Connell, A. Ekin, N. Haas, M. Lu, H. Merkl, and S. Pankanti. Smart video surveillance: exploring the concept of multiscale spatiotemporal tracking. *IEEE Signal Processing Magazine*, 22(2):38–51, March 2005.

[4] P. Mountney, D. Stoyanov, and G. Z. Yang. Three-dimensional tissue deformation recovery and tracking. *IEEE Signal Processing Magazine*, 27(4):14–24, July 2010.

[5] O. Rostamianfar, F. Janabi-Sharifi, and I. Hassanzadeh. Visual tracking system for dense traffic intersections. In *2006 Canadian Conference on Electrical and Computer Engineering*, pages 2000–2004, May 2006.

Lecture #19: Introduction to Deep Learning

Robert Yang
Department of Computer Science
Stanford University
Stanford, CA 94305
{bobyang9}@cs.stanford.edu

1 History of Deep Learning

Deep Learning is an important technique that can be used for numerous applications including computer vision. How were deep learning methods developed? Historically, the perceptron, the first algorithm that led to deep learning models, was published in a paper by Rosenblatt in 1957. As we will see later, multiple perceptrons could be used together to form a linear classifier. In the 1980s, further improvements on the algorithm such as backpropagation were made and researchers started training the linear classifiers, albeit not very well. In the 1990s, researchers focused on graphical models, and in the 2000s, the support vector machine was very popular. However, in the 2010s, we have revisited the 1980s and the reliance on backpropagation to automatically learn our models. Due to vastly improved computing power (e.g. GPUs) and a large amount of data, the formerly clunky algorithms in the 1980s have become useful and accurate. But to understand today's advances in deep learning, we must first understand its foundation based in history: the perceptron model, linear classifier, and backpropagation.

2 The Perceptron

The structure of the perceptron model is defined by a few key aspects. First, the perceptron takes in some input which we can call x. x is a vector, because there can be multiple inputs (e.g. x_1, x_2, $\ldots$, x_n). Second, the perceptron returns one output which we can call y. In the original version of the perceptron, y would be a scalar (one output only). Between the input and the output, the perceptron contains a weighting function that combines the inputs of the perceptron (x_1, x_2, $\ldots$, x_n) with weights w. w is a vector, because there is one weight for each input. Lastly, between the weighting function and the output, there is a sign function, which is usually the following:

$$\begin{cases} 1 & xw \geq 0 \\ 0 & otherwise \end{cases}$$

The perceptron algorithm was inspired from biology, but it is not how the brain works! The brain has neurons which gets input through its dendrites, processes that input in the cell body, and then outputs using the axon which might be sent to other neurons. Similarly, the perceptron takes inputs, processes that input, and then outputs the processed result. The weights of the perceptron are analogous to the strength of the connection across the synapses between neurons. However, neurons are probably much more complex than a simple weighting function, which is why the perceptron algorithm is only loosely inspired.

We can gain intuition of the perceptron algorithm by looking at an application related to computer vision. For example, the inputs to the perceptron might represent some sort of image (and we can do this in many different ways, such as using raw pixels, PCA, and BoW, among others). We can convert this input into an output that might signify whether the input is some type of class (e.g. airplane, bird, etc.)

3 Linear Classifier

A linear classifier consists of multiple perceptrons put together. For example, you could use a linear classifier to make predictions for multiple classes.

The structure of a linear classifier is similar to the structure of the perceptron. Continuing our computer vision example, we see that the input is the same – a vector x that may perhaps contain information about an image, whether that be a raw pixel representation or some other kind of feature representation. However, the output is now a set of scores (not just one score). For example, you could have a score for the airplane class, for the bird class, etc. We could classify by using the class that has the highest score.

The conversion from input to output in a linear classifier is also defined by the weights w, similar to those in a perceptron. w is a 2-D matrix with dimensions (number of classes, number of input elements). We will show why in the example below.

Example 1.

For a certain linear classifier, let there be 10 classes, and let the input be a 32 x 32 x 3 image. Assume that we are using a raw pixel representation. What is the shape of the weight matrix for the classifier?

Answer: First, we must find how many inputs there are. Because we are using a raw pixel representation, the number of inputs will be the number of pixels, which is 32 * 32 * 3 = 3072 inputs. We also know that there are 10 outputs because there are 10 classes (1 output for each class). For each class, we have a set of 3072 weights, and we have 10 sets in total. This will be organized in a (3072, 10) shaped matrix.

As in the example, each class has a set of n_{input} weights, where n_{input} is the dimension of the input vector. You can actually visualize the weights as images:

Sometimes, you also add a bias vector to your results. Intuitively, this helps learn the bias for or against one class in the dataset. For example, if there were more birds in the dataset, you would have a big bias term for the element corresponding to the bird class.

4 Loss Function

We need to pick the weights so that they minimize the error our model makes on the dataset. Formally, this is min_W Loss(y, $\hat{y}$), where y is the true label and $\hat{y}$ is the predicted label.

Given several training examples $(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)$ and a perceptron $\hat{y} = wx$, where x_i is an image and y_i is an integer label (0 for dog, 1 for cat, etc.)

A loss function $L_i(y_i, \hat{y_i})$ tells us how good our current classifier is. When the classifier predicts correctly (y_i is close to $\hat{y}$), the loss should be low. When the classifier predicts incorrectly (y_i is far

from $\hat{y}$), the loss should be high. We can average the loss over the entire dataset to see how well our model is doing in general.

4.1 Different Types of Loss Functions

L2 is the squared error loss function. The goal is to minimize the squared difference between the ground truth labels and the predictions. However, the L2 loss is not very robust to outliers. One relatively strange example will increase the loss by a lot.

Formula for L2 loss: $L_i(y_i, \hat{y_i}) = (y_i - \hat{y_i})^2$

L1 is a loss function similar to L2 which aims to minimize the absolute value of the difference between the ground truth labels and the predictions. The absolute value is there because you want loss to be positive, and zero loss means that you are able to perfectly classify your dataset.

Formula for L1 loss: $L_i(y_i, \hat{y_i}) = |y_i - \hat{y_i}|$

Zero-One is a loss function that only measures if the predictions are correct or incorrect, and does not measure how correct or incorrect the predictions are. If the model makes an error, then the loss is 1, and if the model does not make an error, then the loss is 0. This loss function has a few weaknesses in that models using this loss function takes longer to train (it is less informative on the correctness of the predictions).

Formula for Zero-One loss: $L_i(y_i, \hat{y_i}) = 1||y_i \neq \hat{y_i}||$

Hinge is another loss function that grows bigger when the difference between the prediction and the ground truth increases.

Formula for Hinge loss: $L_i(y_i, \hat{y_i}) = max(0, 1 - y_i\hat{y_i})$

5 The KL Divergence Loss Function and the Softmax Linear Classifier

Sometimes we want the model to output probabilities instead of scores because we can visualize probabilities better. We need to convert the vector of scores into a probability distribution.

There are no limits to the output space for the scores of the linear classifier model, which mean that values might appear which do not signify probabilities (values greater than 1 or less than 0). Softmax is a method to convert the output into probability ranges [0, 1]. It does so by taking the exponential of the outputs and then normalizing them, as you will see below in the formula.

$Prob[f(x_i, W) == k] = \frac{e^{\hat{y}_k}}{\sum_j e^{\hat{y}_j}}$

The KL divergence allows you to calculate the distance between two probability distributions. First, we can define $P(y)$ as the ground truth distribution, and $Q(y)$ as the model's output score distribution (the predictions). KL divergence is defined as the following:

$D_{KL} = \sum_y P(y) log \frac{P(y)}{Q(y)}$

Usually, $P(y)$ is probability 1 for one class and 0 for all other classes. In such cases, since $P(y)$ is 0 for all the other classes, [LaTeX from slide 46] can be rewritten as $log(\frac{1}{Q(y)})$, which can be rewritten as $-log(Q(y))$, where y is the class with probability 1 in $P(y)$.

Notice that softmax and KL divergence work together very well because the exponent and ln will cancel out. Furthermore, inputting the probabilities given by softmax into KL divergence gives a loss from 0 to infinity, which is a good range.

Example 2.

Let C be the number of classes in the classifier. In the scenario where the classifier outputs the same result for all the classes, and that softmax is used to normalize output, what would be the KL Divergence loss?

Answer: If the classifier outputs the same result for all the classes, then each class will have a probability of $\frac{1}{C}$ after softmax. We know that the KL divergence loss would be $-log(Q(y))$, and we

know that Q(y) is $\frac{1}{C}$ because no matter which of the classes has a 1 in the ground truth, the prediction would be $\frac{1}{C}$. Thus, we would find that the KL divergence loss would be $-log(\frac{1}{C})$, which is $log(C)$.

A summary:

Softmax allows us to convert scores into probabilities.

KL Divergence allows us to calculate the distance between the predicted probabilities and the ground truth.

6 Gradient Descent

Gradient Descent is a technique used to find optimal weights that minimize the loss.

6.1 Intuition of Gradient Descent

In Gradient Descent, you iterate through the following loop:

1. You first compute the derivative of the loss with respect to the weights, and with this, you would know which direction to move the weights in order to lower the loss.

2. You would move toward that direction by updating the weights.

You would repeat the two steps until you get to a local minima. In this way, the gradient descent algorithm will slowly decrease the loss.

6.2 Theory

We can calculate what the gradient looks like given our loss function. First, we have a training data point (x, y), and we are using the linear classifier where $\hat{y} = wx$. Let us define k as the class which is correct. We know that the loss is -log(softmax($\hat{y}(k)$)), where $\hat{y}(k)$ is the score given to the correct class by the linear classifier.

Evaluating that expression, we get that Loss = $L(\hat{y}, y) = -log\frac{e^{\hat{y}_k}}{\sum_j e^{\hat{y}_j}} = -\hat{y}_k + log\sum_j e^{\hat{y}_j}$.

Now that we have our loss function, we need the derivative of the loss function with respect to each weight, $\frac{dL}{dW}$. However, since the loss function is in terms of $\hat{y}$, we will need to use the chain rule:

$\frac{dL}{dW} = \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dW}$.

First, we need to calculate $\frac{dL}{d\hat{y}}$. To do this, we need to consider 2 cases.

Case 1: We are calculating the derivative respect to $\hat{y}$ for the class k. In this case, the derivative of the loss function is

$\frac{dL}{d\hat{y}_k} = \frac{d}{d\hat{y}_k}(-\hat{y}_k) + \frac{d}{d\hat{y}_k}(log\sum_j e^{\hat{y}_j})$

$= -1 + \frac{e^{\hat{y}_k}}{\sum_j e^{\hat{y}_j}}$.

Case 2: We are calculating the derivative respect to $\hat{y}$ for a class l where $l \neq k$. In this case, the derivative of the loss function is

$\frac{dL}{d\hat{y}_k} = \frac{d}{d\hat{y}_k}(-\hat{y}_k) + \frac{d}{d\hat{y}_k}(log\sum_j e^{\hat{y}_j})$

$= \frac{e^{\hat{y}_l}}{\sum_j e^{\hat{y}_j}}$.

We also know that since $\hat{y} = Wx$, $\frac{d\hat{y}}{dW} = x$. Thus, we see that $\frac{dL}{dW} = \begin{bmatrix} \frac{e^{\hat{y}_0}}{\sum_j e^{\hat{y}_j}} \\ \frac{e^{\hat{y}_1}}{\sum_j e^{\hat{y}_j}} \\ \cdots \\ -1 + \frac{e^{\hat{y}_k}}{\sum_j e^{\hat{y}_j}} \\ \cdots \\ \frac{e^{\hat{y}_{n-2}}}{\sum_j e^{\hat{y}_j}} \\ \frac{e^{\hat{y}_{n-1}}}{\sum_j e^{\hat{y}_j}} \end{bmatrix}$

where n is the number of classes.

6.3 Pseudocode of Gradient Descent

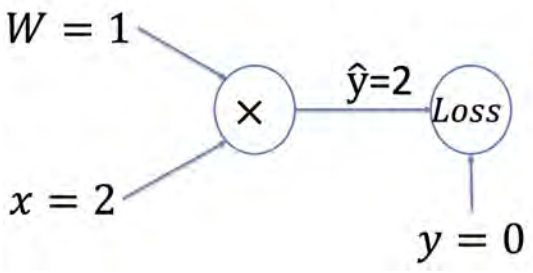
Now, we can translate the theory into code. First, we calculate the loss by looping through each data point and averaging the losses of each data point. Then, we calculate the derivative of the loss with respect to each weight, and update each weight.

```
for _ in {0,…,num_epochs}:
    L = 0
    for x_i, y_i in data:
        ŷ_i = f(x_i, W)
        L += L_i(y_i, ŷ_i)
    dL/dW = We know how to calculate this now!
    W := W − α dL/dW
```

α is a hyperparameter that represents step size. You can tune this hyperparameter to find the optimal step size. If α is too small, then your algorithm will take a long time (many iterations) to finish, while if α is too big, your algorithm might overshoot the local minima.

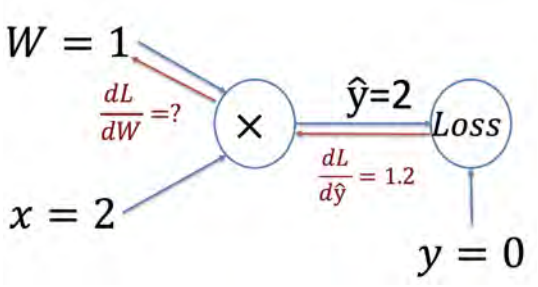
7 Backpropagation

Backpropagation is a method to compute the gradients which visualizes the computation as a graph. For example, you would first have a graph to compute the loss:



You would first apply the multiplication operator on W and x to get $\hat{y}$. Afterwards, you would calculate loss from $\hat{y}$ and y. This step of going forward to find the loss is called the forward pass.

Afterwards, you go backwards to calculate the gradients:



You first calculate $\frac{dL}{d\hat{y}}$ using the methods described in section 6.2. Then, you can calculate values such as $\frac{dL}{dW}$ by using the chain rule ($\frac{dL}{dW} = \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dW}$). In the above example, because $\hat{y} = Wx$, $\frac{d\hat{y}}{dW} = x = 2$. Thus, $\frac{dL}{dW} = \frac{dL}{d\hat{y}}\frac{d\hat{y}}{dW} = (1.2)(2) = 2.4$.

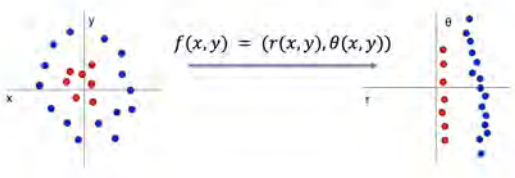
8 Neural Networks

8.1 Basics of Neural Networks

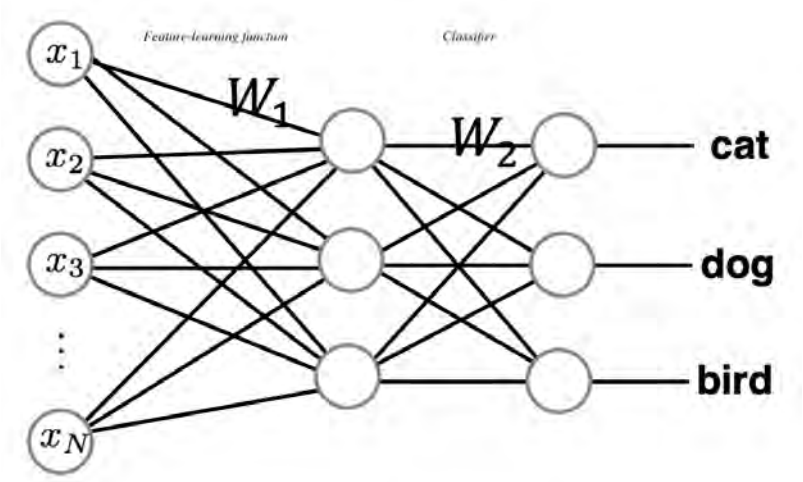
We have many ways of producing features for a dataset, including using the raw pixels, injecting positional information using the raw pixels in addition to the (x, y) coordinates, using LDA or PCA to find features that produce the most variance in a supervised or unsupervised manner, or using a set of interesting areas in an image, called a "Bag of Words". If we are using the dataset for classification, we could then use a linear classifier to separate the data. However, sometimes, the data isn't linearly separable, leading the classifier to perform poorly, as in the example below.

Finding good, linearly separable features for use in classification still is a difficult and somewhat finicky task. Namely, we don't know what data we are going to get, and each dataset might require a different feature encoding method to produce the best results possible. However, there is a solution.

The strategy is to design a function to convert features which are not linearly separable into features that are. For example, in the picture below, by applying a transformation to (r, θ) coordinates, you could now linearly separate the data.



In other words, the function will learn what features to input into our linear classifier. The outputs of that function would be the inputs of the linear classifier, which would look something like this:



This is a neural network with 2 layers, where the first layer generally works to learn better features and the second layer classifies the better features. Neural networks with n layers have the first $n-1$ layers to learn better features and the last layer, usually a softmax, for the purpose of classification.

For the 2-layer network, we can calculate $\hat{y}$ using the following formula:

$\hat{y} = W_2 max(0, W_1 x)$.

Why is the max function necessary? Notice that one property of linear functions like matrix multiplication is that no matter how many of them you apply, the transformation on a whole is linear. Thus, if you got rid of the max function, you would have $\hat{y} = W_2 W_1 x$, but that would just collapse to $\hat{y} = Wx$ where $W = W_2 W_1$.

We can also calculate the shape of the matrices that hold the weights.

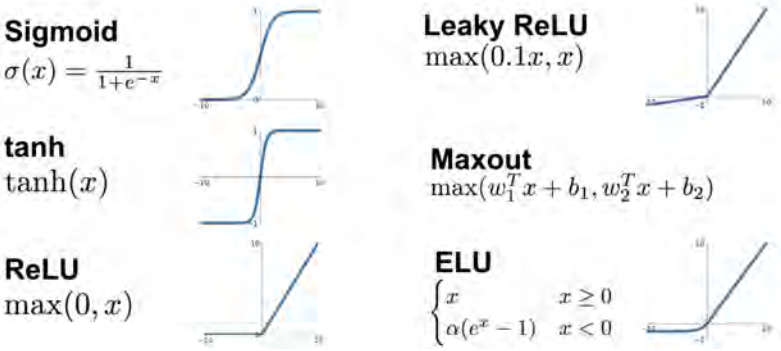
Example 3. Let the shape of x, the input, be (3072, 1), let the shape of y, the output, be (10, 1), and let the shape of a, the hidden units, be (h, 1). What is the shape of W_1 and W_2?

Answer: Since $a = W_1 x$, W_1 must be shape (h, 3072) so that $W_1 x$ is of shape (h, 1). Also, since $y = W_2 a$, W_2 must be shape (10, h) so that $W_2 a$ is of shape (10, 1).

Note: When designing the structure of the network, we determine the value of h, so h is another hyperparameter.

8.2 Activation Function

The max function in the formula $\hat{y} = W_2 max(0, W_1 x)$ for calculating the prediction of a 2-layer neural network is one of many activation functions (also called nonlinearities because they prevent the neural network from collapsing back into one linear combination). Such functions allow models to learn more complex transformations for features. There are many activation functions, as seen below:



The sigmoid function is very popular and widely used but one of the slower functions, while tanh is similar to sigmoid in terms of properties but is centered at 0. ReLU (the rectified linear unit) is the function used in most neural networks today because of its double function in providing nonlinearity and regularization, and it is one of the faster functions. Leaky ReLU is an advanced version of ReLU. Finally, Maxout and ELU are less seen but still used in some cases. Choosing the right activation function is another hyperparameter, so try different ones to see which ones work best.

9 Conclusion

Understanding the history of deep learning and how the algorithms and models developed will be crucial to understanding how deep learning works. The models of deep learning are made of stacks of linked linear classifiers. Deep learning algorithms evaluate their performance using loss functions. Softmax is still popular in the output of deep learning algorithms. Furthermore, backpropagation and gradient descent (advanced versions of it), which give the model the ability to learn weights, is crucial for performance at increasingly complex tasks in computer vision and elsewhere.

Lecture #20: Convolutional Neural Networks

Kendall Beache, Sammy Mohammed, Hannah Zhang
Department of Computer Science
Stanford University
Stanford, CA 94305
{kbeache, sammym, hzhang16}@stanford.edu

1 Introduction

Throughout this class, we have explored several different computer vision techniques, including edge detection, clustering methods, classifiers, and feature detectors/descriptors. These techniques still have one main drawback, in that they require humans to hand-design features. In this lecture, we will go over backpropagation in neural networks, a method to recursively find the ideal weights for the neural network, and convolutional neural networks, a new method designed to solve any image processing problem the network is trained for.

A convolutional neural network is an algorithm that performs *end-to-end learning*, directly mapping raw inputs (images) to a desired output, such as labels or predictions. The history of convolutional neural networks spans decades, but the biggest breakthroughs were in 2010 and 2012, with the publication of Mohamed et al's Acoustic Modeling using Deep Belief Networks, and Dahel et al's Context-Dependent Pre-trained Deep Neural Networks for Large Vocabulary Speech Recognition. Now, convolutional neural networks and deep learning have led to rapid progress in computer vision, which we will explore. Deep learning approaches usually involve combining a small set of simple tools to build a network and then training that network on data for the specific problem you're trying to solve; they can often be adapted to different problems by simply swapping out the training data.

2 Backpropagation in Neural Networks

2.1 Fundamentals of Backpropagation

Backpropagation is an algorithm used to build arbitrarily large neural networks, and gradients only require local information to calculate gradients, via recursive implementation of the chain rule. It allows any gradient in the network to be computed by solving the gradients in later layers. Backprop is necessary, as calculating the gradients for intermediate variables manually is difficult even with just a few layers, and does not scale well to neural networks with many layers.

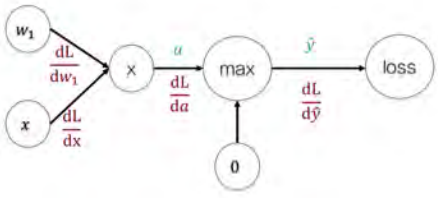


Figure 1: 1D example (CS 131 lecture slide 20-10)

In order to demonstrate backprop, we will use a 1-d neural network as an example. In this case, $a = wx$, and $\hat{y} = \max(0, a)$. Our goal is to calculate dL/dW. Using the chain rule, we can see that

$dL/dw_1 = dL/da * da/dw_1$, and $dL/da = dL/d\hat{y} * d\hat{y}/da$. We can then apply the chain rule to solve.

2.2 Rules for Calculating Gradients

There are some general rules that can help when calculating gradients of input features. If the operation is addition, then the gradients of the input features are distributed in that $\frac{dL}{dx_i} = \frac{dL}{dy}$ for all input features x_i. If the operation is multiplication, then each gradient will be proportional to the values of the other input features. For example, if there are two input features w and x, then $\frac{dL}{dw} = x\frac{dL}{dy}$ and $\frac{dL}{dx} = w\frac{dL}{dy}$. If the operation is a maximum function, like $y = \max(x, w)$, then $\frac{dL}{dw} = \frac{dL}{dy}$ if $w \geq x$ and 0 otherwise; $\frac{dL}{dx} = \frac{dL}{dy}$ if $x \geq w$ and 0 otherwise. If the function is $y = e^x$, then $\frac{dL}{dx} = y\frac{dL}{dy}$, since $\frac{dy}{dx} = e^x$.

3 Convolutional Neural Networks

Convolutional neural networks are a type of deep learning models that are often used for object recognition and classification. At a high level, CNNs start with an original image, and convolve it with multiple filters. The parameters for convolution are also chosen via backpropagation, ensuring that we do not have to hand-choose features.

Convolutional neural networks can be broken down into 3 steps 1) Convolution, 2) Non Linearity, 3) Pooling

3.1 Convolution:

We use convolution in this step as a means to extract features from the original input image. A CNN learns the values of the filters or kernels during the training process. Typically we use 1 2D convolution layer on black and white images; however, if you wanted to extract features from a color image you can use 3 2D convolution layers (one for each channel).

The size of the convolved feature is controlled by three parameters: depth, stride, and zero-padding that we need to decide before the convolution step is performed. Depth corresponds to the number of filters we use for the convolution operation. Stride is the number of pixels by which we slide our filter matrix over the input matrix. We use zero-padding to apply the filter to bordering elements of our input image matrix. Zero padding also allows us to control the size of the feature maps.

Introduce Non-Linearity (ReLu Function): After every convolution, the ReLu function is applied to introduce non-linearity in our model (because most real-world data we would want our ConvNet to learn would be non-linear.) ReLU stands for Rectified Linear Unit and its output is given by the following:

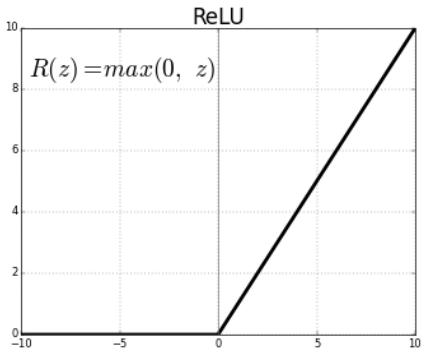


Figure 2: The ReLU function. (Kanchan Sarkar, Medium))

Other non-linear functions that can be applied to the convolved images include the sigmoid and tanh function; however, the ReLu function tends to give the best results.

3.2 Pooling

Modern convolutional neural networks incorporate pooling layers – layers in the neural network that downsize the dimensionality but retain the most important information. This works by defining a spatial neighborhood, such as a 2x2 window, and then taking different data from the neighborhood depending upon the pooling algorithm. One example of this sort of pooling is MaxPool, where the max value in the window is the value that is retained. There are other pooling algorithms, such as AveragePool and SumPool, where the average is taken and the sum of the window is taken.

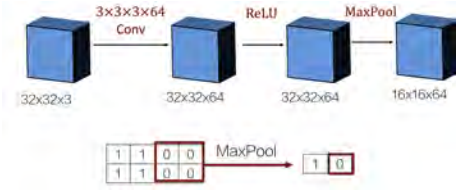


Figure 3: Max pooling reduces dimensionality by a factor of 2. (CS 131 lecture slide 20-38)

3.3 Architecture

These different layers can be stacked, feeding the input of one layer into the other in order to help improve output quality. These layers extract relevant information, and then classify.

Figure 4: CNN Architecture (CS 131 lecture slide 20-41)

This basic structure can be expanded – consider GoogLeNet, a CNN with an Inception layer, a hand-designed network within a network.

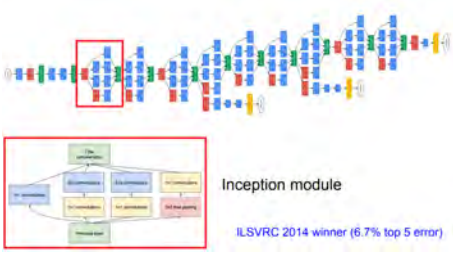


Figure 5: GoogLeNet architecture and Inception module. (CS 131 lecture slide 19-101)

However, this layering process comes with a drawback. Convolutional neural networks are not shift-invariant, due to the pooling layer – minor shifts in the image can result in dramatically different classification outputs. (Best demonstrated here: https://richzhang.github.io/antialiased-cnns/). One potential solution would be the BlurPool algorithm, where the window is blurred and shifted before MaxPool is ran, resulting in better outcomes. However, this does not solve the shift-invariance problem, so results may still be unpredictable.

4 Conclusion

While CNNs meet the basic architecture described above, there are still many variations between different networks. CNNs can differ by the amount of layers or values of the hyperparameters used.

If you are interested in learning more about different CNNs, read more on GoogLeNet, VGGNet, or ZF Net.