“Foundations Of Data Science” Notes

Prof. Fabio Galasso. Notes written by Alessio Lucciola during the a.y. 2022/2023.

Notes may contain errors or typos. If you see one, you can contact me using the links in the Github page. If you find this helpful you might consider buying me a coffee 😄.

1. Introduction

1.1 What is “Data Science”

Data science is an interdisciplinary field that uses scientific methods, processes, algorithms and systems to extract knowledge and insights from noisy, structured and unstructured data, and apply knowledge and actionable insights from data across a broad range of application domains. Data science is related to data mining, machine learning and big data.
There are two steps of working with data:

1. **Collect data**: via computers, sensors, people, events, etc..;
2. **Do something with it**: make decisions, confirm hypotheses, gain insights, predict future, etc..;

Once the data is collected, it is possible to use it in many ways, for example:

- Traffic: it is possible to predict congestions and the traffic flow in the present and the immediate future;
- Recommendation system: it is possible to recommend things (such as films to watch or products to buy) based on user’s interests;
- Sports: it is possible to make predictions on how a match will end up;

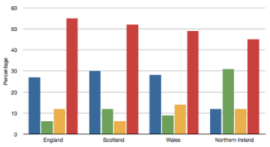
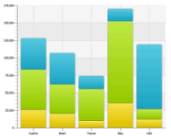
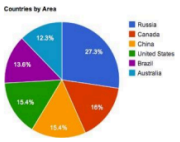
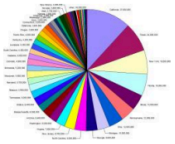
And many more like weather prediction, online advertisement, medical diagnosis, financial markets, resource management, computational social science, smart buildings and cities.

1.2 Data Tools and Techniques

The are some tools and techniques to get and work with data:

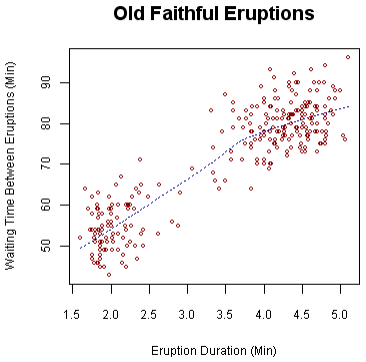
- **Basic Data Manipulation and Analysis**: It is about performing well-defined computations or asking well-defined questions (“queries”) (in order to retrieve particular information).
    - Some examples:
        - Average January low temperature for each country over last 20 years;
        - Number of items over $100 bought by females between ages 20 and 30;
        - Frequency of specific medicine relieving specific symptoms;
    - Some useful tools:
        - Spreadsheets (to visualize data);
        - Relational (SQL) database systems (to visualize data and make queries);
        - “NoSQL” / scalable systems;
        - Programming languages with data support (e.g., Python, R);
- **Data Mining**: Looking for patterns in data.
    - Some examples:
        - Items X,Y,Z are bought together frequently;
        - People who like movie X also like movie Y;
        - Patients who respond well to medicines X and Y also respond well to medicine Z;
    - Some useful tools:
        - Frequent item-sets;

        - Association rules (if..then..);
        - Specialized techniques for graphs, text, multimedia;
- **Machine Learning**: Using data to build models and make predictions.
    - Some examples:
        - Customers who are women over age 20 are likely to respond to an advertisement;
        - Students with good grades are predicted to do well on entrance exams;
        - The temperature of a city can be estimated as the average of its nearby cities, unless some of the cities are on the coast or in the mountains;
    - Some useful tools[1]:
        - Regression;
        - Classification;
        - Clustering;

Note: Basic data analysis and data mining give answers from the available data, while machine learning uses the available data to make predictions about missing or future data

- **Data Visualization**: It is possible to graphically depict data. There are many ways to visualize data, even basic visualizations such as:
    - Bar Charts;
    - Pie Charts;
    - Scatterplots: A scatter plot is a type of plot or mathematical diagram using Cartesian coordinates to display values for typically two variables for a set of data. The data are displayed as a collection of points, each having the value of one variable determining the position on the horizontal axis and the value of the other variable determining the position on the vertical axis;
    - Maps;
- **Data Collection and Preparation**: Collect data from different sources in order to analyze it. There may be some problems with the collection of data such as:
    - Extracting data from difficult sources;
    - Filling in missing values;
    - Removing suspicious data;
    - Making formats, encoding, and units consistent;
    - De-duplicating and matching;



1.3 Correlation and Causation

Data analysis, data mining, and machine learning can reveal relationships between data values:

- **Correlation**: Values track each other (it is simply a relationship where action A relates to action B but one event doesn't necessarily cause the other event to happen). Some examples:
    - Height and Shoe Size;
    - Grades and Entrance Exam Scores;
- **Causation**: One value directly influences/causes another (in other words, action A causes outcome B). Some examples:
    - Education Level → Starting Salary;
    - Temperature → Cold Drink Sales;

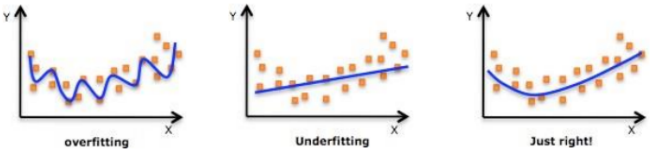
Note that correlation *does not* imply causation:

- Correlation can be result of causation from a hidden "confounding variable";
- A and B are correlated because there's a hidden C such that C → A and C → B; Some examples:
    - Homeless population and crime rate are both related to unemployment (which is the confounding variable). In other words unemployment (C) causes crime rate (A) and the increase in homeless population (B) and the two are correlated events. We can't say that homelessness necessarily increases the crime rate.

[1] More Info: https://www.upgrad.com/blog/clustering-vs-classification/#:~:text=Regression%20and%20Classification%20are%20types,it%20is%20a%20 classification%20problem.

    - Sunny weather causes eating ice cream and getting sunburned. These two events are correlated.
- Correlation is usually “easy” to test while causation is typically impossible to test (if we have two events which we think they’re correlated, it is difficult to determine the variable C).

1.4 Underfitting and Overfitting



Machine learning uses data to create a “model” and uses models to make predictions. There may be some problems with these models:

- **Underfitting**: Model used for predictions is **too simplistic**. For example:
    - 60% of men and 70% of women responded to an advertisement, therefore all future ads should go to women;
    - If a furniture item has four legs and a flat top it is a dining room table;
- **Overfitting**: Model used for predictions is **too specific**. For example:
    - The best targets for an advertisement are married women between 25 and 27 years with short black hair, one child, and one pet dog;
    - If a furniture item has four 100 cm legs with decoration and a flat polished wooden top with rounded edges then it is a dining room table;

Given a scatter plot, it is easy to notice if we're in the case of overfitting or underfitting. First off all the operation of **regression** helps us to create a ML model such that it minimizes the discrepancy between data samples and our estimated curve. So, given data samples we have to make a curve between data: if the curve is too simplistic (we leave out too much data) then we’re underfitting, otherwise (we consider way too much data) we’re overfitting.

1.5 What is “Machine Learning”

Machine learning (ML) is the study of computer algorithms that can improve automatically through experience and by the use of data. It is seen as a part of artificial intelligence. Machine learning algorithms build a model based on sample data, known as training data, in order to make predictions or decisions without being explicitly programmed to do so. Machine learning algorithms are used in a wide variety of applications, such as in medicine, email filtering, speech recognition, and computer vision, where it is difficult or unfeasible to develop conventional algorithms to perform the needed tasks. Some examples can be found in:

- Database mining:
    - Large datasets from growth of automation/web (e.g. web click data, medical records, biology);
- Applications which cannot be programmed by hand:
    - E.g. autonomous driving, handwriting recognition, most of Natural Language, Processing (NLP), Computer Vision;
- Self-customizing programs:
    - E.g. Amazon, Netflix product recommendations;
- Understanding human learning (brain, real AI);

One important definition of ML is the following:
“A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.” - Tom Mitchell (1998)

EX. Suppose your email program watches which emails you do or do not mark as spam and, based on that, learns how to better filter spam. Now:

- Task (T): Classifying emails as spam or not spam;
- Experience (E): Watching you label emails as spam or not spam;
- Performance (P): The number (or fraction) of emails correctly classified as spam/not spam.

There was an evolution on the way of approaching ML:

1. Expert Systems (1980+): Over 20 years ago, we had rule based systems so we had experts that had to obtain a PhD in linguistics, introspect about the structure of their native language and write down the rules. So they had to study all possible cases regarding a specific problem and then deal with the problem with “If-Then-Else”.

EX. Give me direction to Starbucks
If: "give me directions to X";
Then: directions(here, nearest(X));

Experts are good at answering questions about specific arguments but **they are not good at telling how to do it**.

2. Annotate Data and Learn (1990+): So why not just have them tell you what they do on **specific cases** and then let **ML** tell you how to come to the same decisions that they did.
So what is done (also in current ML) is to collect raw sentences $\{x_1, .., x_n\}$ and let experts annotate their meaning $\{y_1, .., y_n\}$.
EX. x_1: How do I get to Starbucks?
y_1: directions(here, nearest(Starbucks))

What's the difference between ML and DS?
Because data science is a broad term for multiple disciplines, machine learning fits within data science. Machine learning uses various techniques, such as regression and supervised clustering. On the other hand, the data in data science may or may not evolve from a machine or a mechanical process. The main difference between the two is that data science, as a broader term, not only focuses on algorithms and statistics but also takes care of the entire data processing methodology (e.g. data collection and manipulation).

1.6 What is "Computer Vision"?

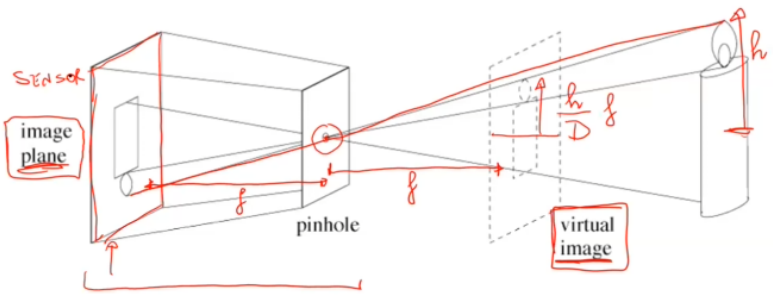
Computer vision is an interdisciplinary scientific field that deals with how computers can gain high-level understanding from digital images or videos. From the perspective of engineering, it seeks to understand and automate tasks that the human visual system can do.
Some examples can be found in:

- License Plate Recognition (for example London congestion charge);
- Security/Surveillance:
  - Face Recognition (Apple's Face ID: chance of 1-in-1-million that a random person could unlock your phone);
  - Biometric passport (aka e-passport) has an embedded electronic chip which contains biometric information;
- Medical Imaging:
  - (Semi-)automatic segmentation and measurements;
- Autonomous driving;
- Robotics;

2. Basic Concepts and Terminology for Image Processing and Computer Vision

Before introducing the concept of image processing, it is necessary to explain what a camera is.
First of all, let's introduce the first model of a camera used by Leonardo Da Vinci to actually invent the camera obscura in around 1520. The **pinhole camera** is the simple standard and abstract model for a camera: a pinhole camera is a simple camera without a lens but with a tiny aperture (the so-called pinhole)—effectively a light-proof box with a small hole in one side. Light (e.g. a candle) from a scene passes through the aperture and projects an inverted image on the opposite side of the box, which is known as the **camera obscura** effect. It is similar to what happens with our brain, in fact the pinhole is our retina which projects an inverted image that is elaborated by the brain.

"When images of illuminated objects ... penetrate through a small hole into a very dark room ... you will see [on the opposite wall] these objects in their proper form and color, reduced in size ... in a reversed position owing to the intersection of the rays" - Leonardo Da Vinci (1519)



Modern pinhole cameras have some sensors that are able to turn the acquired image, through the optic, into a digital image (through a **digitizer**) where colors are represented with arrays of RGB values. There are sensors which are able to sense all different colors (the three principal ones - RGB) because they have different frequencies.

Given an image, the main goal of **computer vision** is to understand how it is possible to recognize something from an array of (gray-scale) numbers and also how to perceive depth. It is quite the *opposite* of **computer graphics**, whose goal is to find a way to generate an array of (gray-scale) numbers that looks like a certain object.

EX. CV: Given an image (an array of (gray-scale) numbers), how can we recognize fruits?
CS: How can we generate an array of (gray-scale) numbers that look like fruits?

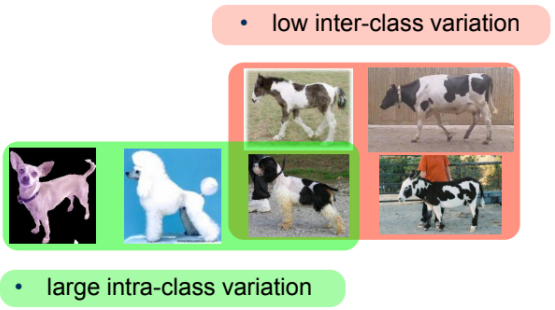
So, the main case study of computer vision is how it is possible to recognize objects and how we can make computers do this. There are some problems like missing data, ambiguities and multiple possible explanations. There are different types of **recognition problems**:

- **Object Identification:** recognize a specific object
  - e.g. recognize your apple, your cup, your dog;
- **Object Classification** (also called generic object recognition or "basic level category") recognize any objects of a specific type:
  - e.g. recognize any apple, any cup, any dog;

Some basic level categories are:

- The highest level at which category members have **similar perceived shape**;
- The highest level at which a **single mental image** can reflect the entire category;
- The highest level at which a person uses similar **motor actions** to interact with category members;
- The level at which human subjects are usually **fastest** at identifying category members;
- The first level named and understood by **children**;

There are a lot of challenges for object classification like: multi-scale, multi-view, multi-class, varying illumination, occlusion (the object is partially covered), cluttered background, articulation, **high intra-class variance** (given a specific class of objects, there is a lot of variation, e.g. dog breeds), **low inter-class variance** (there may be different classes that are not so different, e.g. look how the cow, the horse and the dog are quite similar even if they belong to different classes).



We can also talk about recognition and:

- **Segmentation**: separate pixels belonging to the foreground (object and the background);
- **Localization/Detection**: position of the object in the scene, pose estimate (orientation, size/scale, 3D position);

So, given a specific object in an image, segmentation aims to find the object an to separate it from the background while localization aims to find the object in the image according to its position, orientation and 3D position (given a model of an object, it may have different form and position so we have to find a way to match the object with its model, e.g. the razor example in the slide).

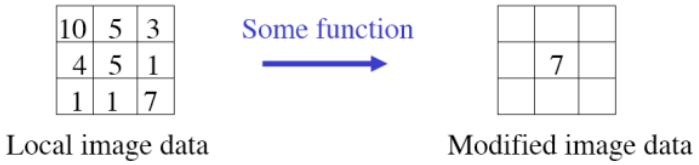
3. Basics of Digital Image Filtering

Before going on, remember that computer vision is the "reverse" of the image processing so, given a 2D digital image processing there is a "**pattern recognition**" analysis that leads to the image understanding. Remember that during the process of "digitization" **we lose some information** like:

- Colors: since the camera has some sensors that recognize only the 3 main colors (Red, Green, Blue) and the other ones are expressed as a combination of the three;
- Dynamic range: since we get a 2D image, we eventually lose the idea of distance;

3.1 Image Filtering

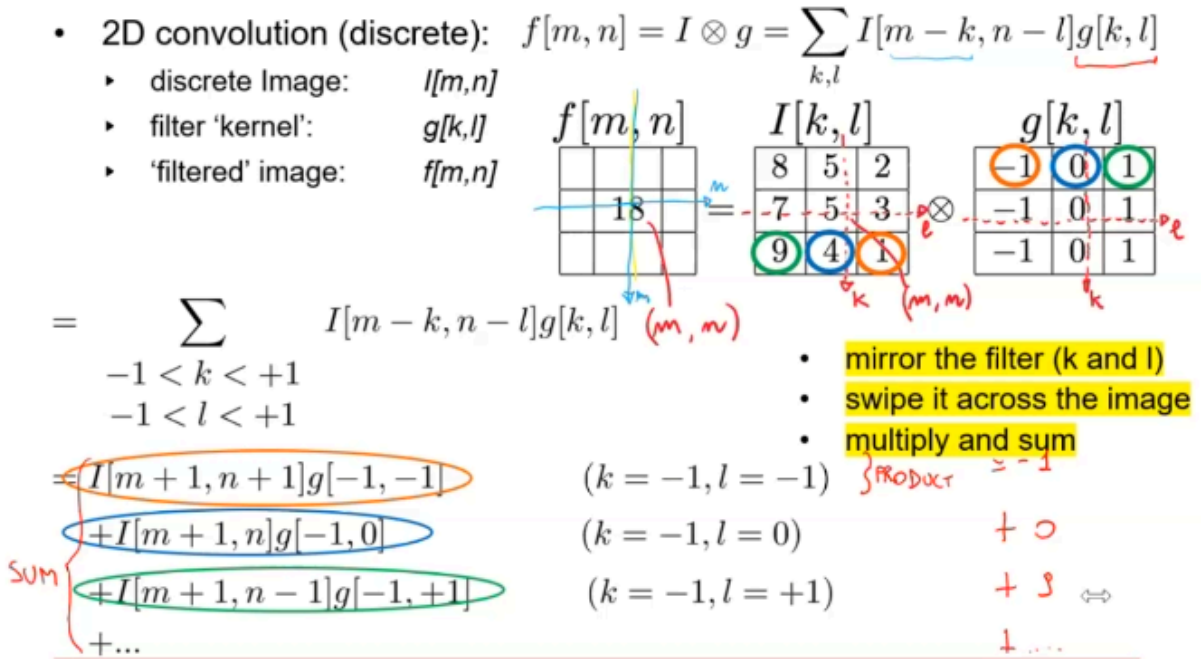
Image filtering is about **applying some function to a local image patch** (a small fragment in the image) to obtain a certain result and recover, for example, pieces of information that have been lost during the digitization process. For example: given a 3x3 block (with some values in it), it is possible to extract a particular number:



With image filtering it is possible to:

- **Reduce the noise**: based on the assumption that there is some smoothness, if in the sensor of the camera there is dead pixel (that is a black pixel) it is possible to change it with a pixel that has a similar color to the surrounding ones;
- **Fill-in missing values/information**: same as the noise but here we have no information. We can, for example, fill a missing value by deciding it based on the surrounding information;
- **Extract image features**: it can be used to extract edges/corners keeping in mind that colors are "smooth" except when there is some edge (a sudden change of color);

One simple case of image filtering is **linear filtering** which consists in replacing each pixel by a **linear combination** of its neighbors. One example of linear combination is **2D convolution**:



Given a discrete image I, pixels in the image are represented by two coordinates [m, n] (e.g. the number in the center is I[0, 0] while the most right one above is I[-1, -1]). So, given a 3x3 local patch, the indexes l and k vary from -1 to 1. We can now introduce what convolution is: A **digital filter** (a small 2D weight **mask**, also called **kernel**) slides over the different input positions. For each position, an output value is generated by replacing the source pixel with a weighted sum of itself and its nearby pixels. The mask slides over our data from left to right covering the entire input matrix. **Convolution** is the process of **adding each element of the image to its local neighbors, weighted by the kernel**. Notice that this is strongly related to the form of mathematical convolution. The matrix operation being performed (convolution) is not traditional matrix multiplication, despite being similarly denoted by *.

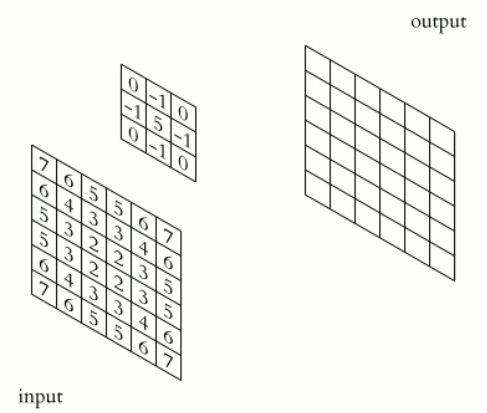
For example, if we have two 3x3 matrices, the first a kernel, and the second an image piece, convolution is the process of flipping both the rows and columns of the kernel and multiplying locally similar entries and summing. Notice that the operation of flipping the kernel is not mandatory but it can be useful if we do the math by hand. (Looking at the convolution formula it is easily noticeable that the flipping operation is done implicitly). The element at coordinates [0, 0] (that is, the central element) of the resulting image would be a weighted combination of all the entries of the image matrix, with weights given by the kernel.

EX. So in the exemple above, we are computing the convolution for the centered element (so k, l = 0) so we just have to multiply each values of the filter g for the respective value in the image I (remember to mirror the filter) then, once all multiplications are done, just sum them the partial results:

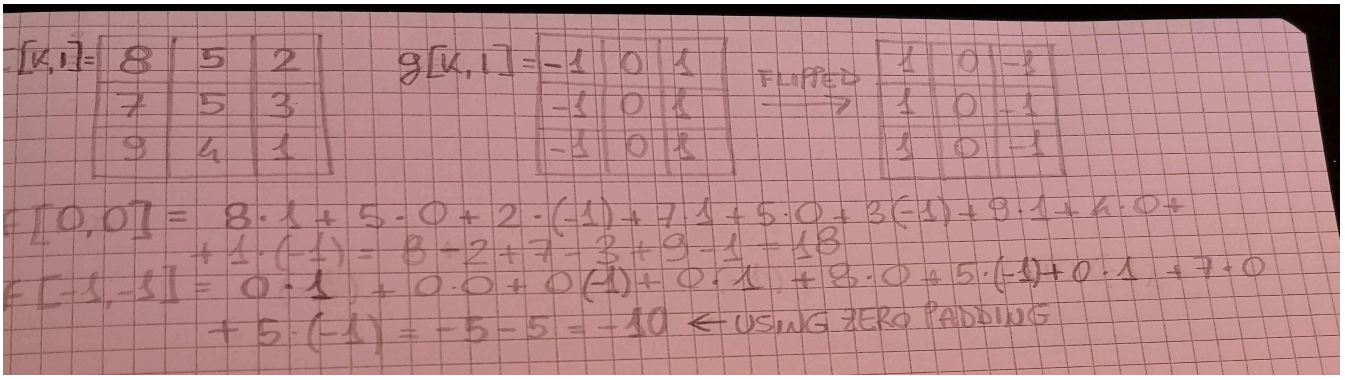
+ k = -1, l = -1 → I[m+1, n+1] g[-1, -1] = 1*(-1) = -1;
+ k = -1, l = 0 → I[m+1, n] g[-1, 0] = 0*4=0;
+ k = -1, l = +1 → I[m+1, n-1] g[-1, +1] = 9*1=9;
+ …
+ k = +1, l = +1 → I[m-1, n-1] g[+1, +1] = 8*1=8;

Then:

= 18;

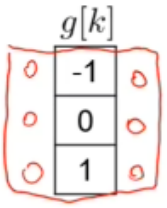


Notice that in order to be consistent with the definition, we use **zero padding**: this practice consists in adding a border of pixels all with value zero around the edges of the input image. So, for example, if we compute f[-1, -1] then we should set I[-1, -1], I[-1, 0], I[0, -1] and I[-1, +1] equal to 0 because, basically, the kernel would slide of the image so we don't consider the kernel values.



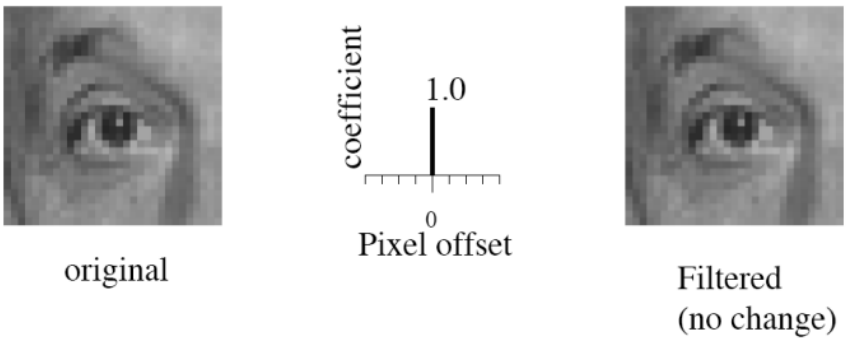
It is also possible to make a convolution (discrete) of a 2D image using a **1D filter**:

$$f[m,n] = I \otimes g = \sum_k I[m-k,n]g[k]$$



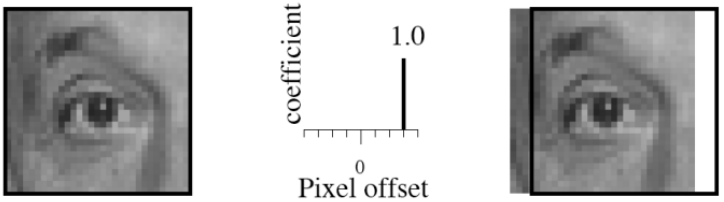
In this case we still swipe the kernel but the only direction that matters is the one represented by the k index. It gives us the same result as using a 3x3 matrix with the 1D filter in the center and all zeros on both sides (we can build this matrix using zero padding starting from the original 1D filter). Of course, the 1D filter can also be horizontal, and in this case we'll change the l index.
A 2D filter is often separable in two 1D filters (one horizontal and one vertical) and it is usually preferable.

Some examples:



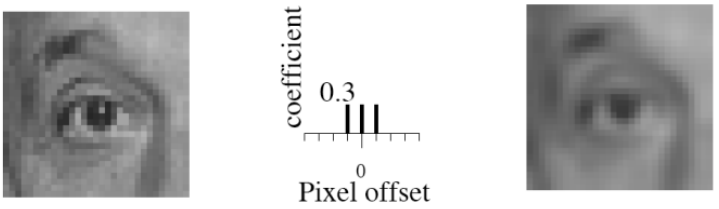
Our function f is representent like f = I⊗g (where I is the original input and g is the kernel).
Note that g can be represented as a 1D matrix [0, 0, 0, 1, 0, 0, 0]. Note that using this filter (remember to mirror it!), the image doesn't change at all. Given a certain position in the image, what we do is take the filter, we mirror it and swipe across the k (neighbor) image pixels making the product between the value of the image and the one

of the filter. Finally the sum of the products and, since we only had one central 1 in the filter, the final result will be the value of the initial position. Applied to the whole image, this gives us the image with no actual change.



original shifted

In this case, since g=[0, 0, 0, 0, 0, 0, 0, 1], we take the pixel further to the right and we place it to the left (because of the mirroring). The result is that we shift the image to the left by 6 positions.



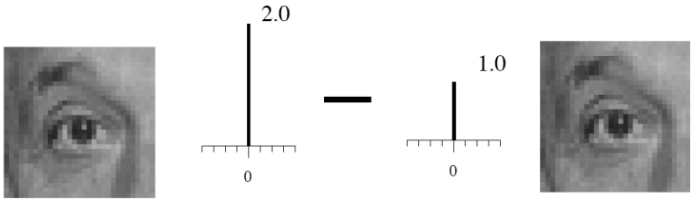
original Blurred (filter applied in both dimensions).

In this case g=[0, 0, 0.3, 0.3, 0.3, 0, 0] and what we do is get three adjacent pixels and mix them with the same coefficient getting the same color. Note that the overall level of luminosity remains the same.
Given an **impulse** in the image (that is a peak in the image), blurring the image allows the impulse to spread across the three pixels in the middle.
Given an **edge**, blurring the image will make the edge less sharp (an edge is a sudden change of color in the image and the transition between the colors will get smoother with blurring).

Note that **convolution respect associative ownership**, so the same result can be achieved in these ways:

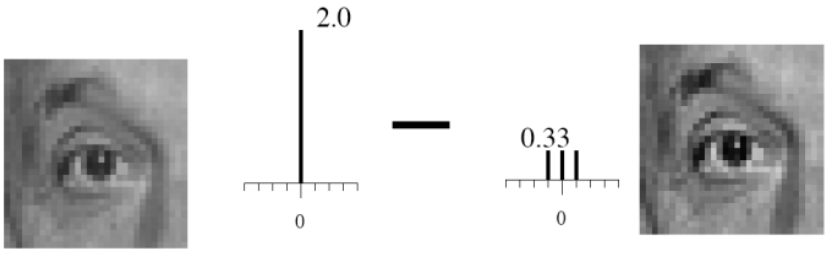
- Apply 2 filters to the image and subtract the partial results;
- Subtract the filters and apply the result to the image;

$$f[m,n] = I \otimes g_1 - I \otimes g_2 = I \otimes (g_1 - g_2)$$



original Filtered (no change)

We already saw blurring, sharpening can be achieved this way:



original Sharpened original

EX. Other example of sharpening → slide;

SUMMARY:

Images may need low-level adjustment such as filtering, in order to enhance image quality (e.g. denoising) or extract useful information (e.g. edges):

- Filtering for enhancement → improve contrast;

- Filtering for smoothing → removes noise;
- Filtering for template matching → detect known patterns;

Convolution is a linear filter and some **basic properties** of **linear systems** are:

- Homogeneity: T[a X] = a T[X] (Apply a linear filter T to the image X and then multiply it by a scalar a);
- Additivity: $T[X_1 + X_2] = T[X_1] + T[X_2]$;
- Superposition: $T[aX_1 + bX_2] = a\,T[X_1] + b\,T[X_2]$ (Combination of homogeneity and additivity);
- Linear systems <=> Superposition (A system is linear iff the superposition applies);

3.1.1 Filters to Reduce Noise (Box and Gaussian Filtering)

Our main aim is to reduce **noise** that can be low (light fluctuations, sensor noise, etc..) or complex (shadows, extraneous objects). Our **assumption is that the pixel's neighborhood contains information about its intensity**. As an assumption, an image I is the result of a signal S with its respective noise N (noise does not depend on the signal). So:

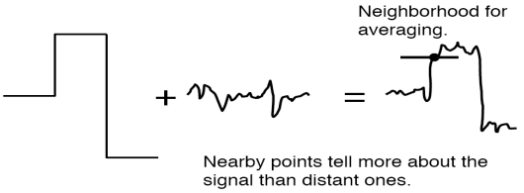
Image I = Signal S + Noise N

In particular, we consider:

- I_i: intensity of i'th pixel;
- $I_i = s_i + n_i$ with $E(n_i) = 0$ (we expect that the noise is 0)
  - s_i deterministic (deterministic = we know exactly the value) ;
  - n_i, n_j independent for $i \neq j$ (the noise in some position is independent from noise in another position);
  - n_i, n_j i.i.d. (independent, identically distributed);

Given the image, if we do the average of a particular area (e.g. the highest area) we recover exactly the value of signal for that particular area.

$$S + N = I$$



Now, if we talk about images and we assume we have noise, a way to recover the signal (and **reduce the noise**) is to apply an **average filter**.
One filter is the **BOX filter** which Replaces each pixel with an average of its neighborhood. It is basically a 3x3 filter with positive entries that sum to 1 (so each entry is about 1/9). Notice that the weights are all equals.

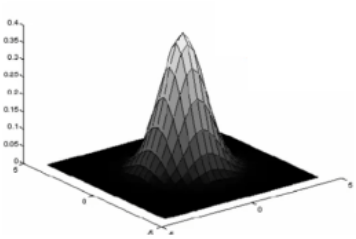
$$\begin{array}{|c|c|c|}\hline \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \hline \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \hline \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \hline \end{array}$$

Box Filter Kernel

Another way to filter is the **gaussian averaging** which is **rotationally symmetric** and weights nearby pixels more than distant ones: this makes sense as "probabilistic" inference.

- the pictures show a smoothing kernel proportional to

$$g(x,y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$



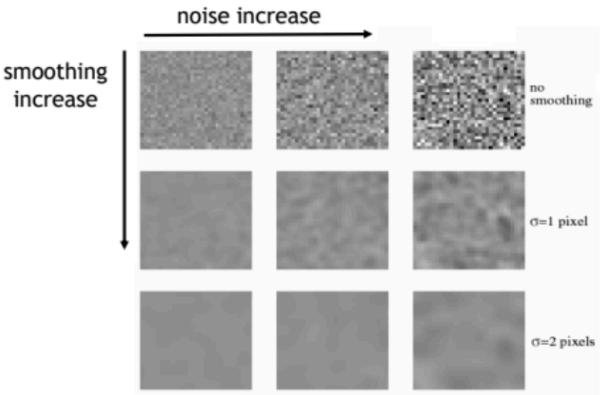
So, there are pixels that contribute differently: given a certain pixel, its immediate neighbor contributes the most and this contribution decreases as we move away from that pixel. This depends on the **σ** which is also called **standard deviation** (σ^2 is the **variance**): if we reduce the sigma (the bell becomes more peaky) we average in a smaller area (so we have a less massive smoothing, we'll keep more detail in the image). Contrarily, with a larger sigma we'll filter more and have a more massive smoothing.

EX. Given an image (let's say a gray square), we can add a different level of noise (see the images on top that have a larger noise starting from the left). In order to recover the original image we need to smooth by applying a gaussian filter. Look that the more noise we have, the larger sigma has to be in order to remove the noise.

Both the **BOX filter and the gaussian filter are separable**, which means we can apply them by first convolving each row with a 1D filter and then convolving each column with a 1D filter. In other words both filters can be expressed as the convolution of an horizontal only row-wise and a vertical only filter.

$$(f_x \otimes f_y) \otimes I = f_x \otimes (f_y \otimes I)$$

Note: remember that convolution is linear (associative and commutative).



EX.

Example: separable BOX filter

$$\begin{array}{|c|c|c|} \hline \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \hline \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \hline \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \hline \end{array} \otimes \begin{array}{|c|} \hline \frac{1}{3} \\ \hline \frac{1}{3} \\ \hline \frac{1}{3} \\ \hline \end{array}$$

$f_x \otimes f_y$ $\quad$ f_x $\quad$ f_y

Example: Separable Gaussian

- Gaussian in x-direction

$$g(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{x^2}{2\sigma^2}\right)$$

- Gaussian in y-direction

$$g(y) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{y^2}{2\sigma^2}\right)$$

- Gaussian in both directions

$$g(x,y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$

$$h(i,j) = f(i,j) * g(i,j) =$$
$$= \sum_{k=1}^{m}\sum_{l=1}^{n} g(k,l) f(i-k, j-l) =$$
$$= \sum_{k=1}^{m}\sum_{l=1}^{n} e^{-\frac{(k^2+l^2)}{2\sigma^2}} f(i-k, j-l) =$$
$$= \sum_{k=1}^{m} e^{-\frac{k^2}{2\sigma^2}} \left[\underbrace{\sum_{l=1}^{n} e^{-\frac{l^2}{2\sigma^2}} f(i-k, j-l)}_{h'}\right] =$$
$$= \sum_{k=1}^{m} e^{-\frac{k^2}{2\sigma^2}} h'(i-k, j)$$

1-D Gaussian horizontally

1-D Gaussian vertically

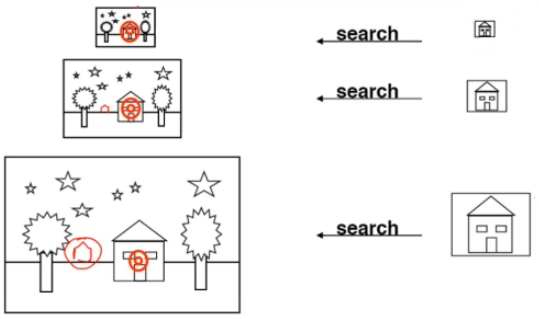
DIM. **Gaussian Separability**: Gaussian separability is an n dimensional Gaussian convolution is equivalent to n 1D Gaussian convolutions. f(i,j) is our image I and g(i,j) is the 2D gaussian function. The output h(i,j) is the convolution between the two. So, we apply the definition of convolution and substitute g(k,l) with the definition of the gaussian. Now, the term k is independent from l so we can take it outside and separate them. We have now a 1D gaussian horizontally. We compute it and then the result is a 1D gaussian vertically. Basically, it is possible to transform the second operation in the convolution between two 1D gaussian (one horizontal and one vertical) so something like $f(i,j) \otimes g_x(i) \otimes g_y(j)$.

How do we apply a 2D gaussian filter?
Just start from the left top of the image and apply the $G_x * G_y$ 2D filter (with coordinates x and y) doing all the possible computation (so multiplications and a sum at the end).
For what concerns the computation, we have $G_x * G_y * I_y * I_x$ steps. Note that if we first apply a G_x 1D gaussian filter and a G_y 1D gaussian filter the steps don't change at all.

3.2 Multi-scale Image Representation

Convolution can also be used for **template matching** (it is a technique in digital image processing for finding small parts of an image which match a template image). We can scan an element (e.g. a house) in an image by applying convolution to the image: basically the output of the convolution will be maximum where the template overlaps that object in the image. Of course, objects can have different scales so we can start by passing a bigger element to detect (e.g. a bigger house) and then we can scale it down to see if there are smaller elements of the same type. In reality there are two approaches:

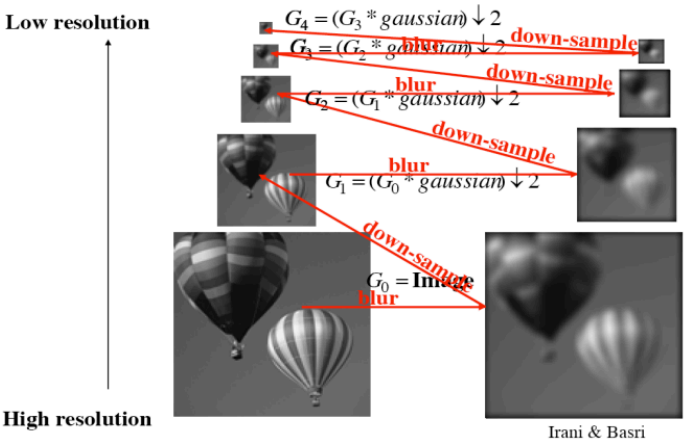
1. **Rescale the pattern** (and keep the image constant);
2. **Rescale the image** (and keep the pattern constant);



So, we can start from a bigger pattern and scale it down until we find the desired objects (if they are in the image). In order to get the multiple scale we need to subsample: we iteratively remove pixels so that we go from a high resolution image to a smaller resolution. To reduce resolution we must:

- **Blur the image**: with a Gaussian filter so that we can keep most important pixels;
- **Downsampling**: it is now possible to resize the image;

Note: It is necessary to blur the image before resizing it to avoid *aliasing*[2]. Blurring the image, we delete the high frequencies in the image and keep only the low frequencies and this is possible because, given a local patch, it retrieves the average of the neighbors. The more we resize, the larger sigma has to be at each step. Also, starting from the original image, we can get a lot more details while scaling it down we'll only keep the most important ones (e.g. the most important edges of the object we're considering).



Some question of interests could be:

- Which information is preserved over "scales": One pixel over an area (the lower frequencies). Of course while choosing the pixel we make the average in that area.
- Which information is lost over "scales": what we lose is definition. The more we scale, the more pixels we lose. At the end, only the most important pixels are preserved (usually the brighter ones, those with higher frequencies).

Note: According to the Fourier transform, a signal (e.g. an image that we have) is a superposition of harmonics sine and cosine that allow it to represent frequencies from low ones to high ones. If we only include the low frequencies that we have the coarse signal whose advantage is that it can be requested with fewer pixels.

3.3 Edge Detection

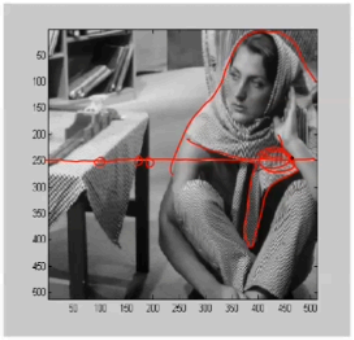
Another important task in CV is **edge detection**. To recognize and detect edges it is necessary to:

1. "Filter" image to find brightness changes;
2. "Fit" lines to the raw measurements;
3. "Project" model into the image and "match" to lines (solving for 3D pose);

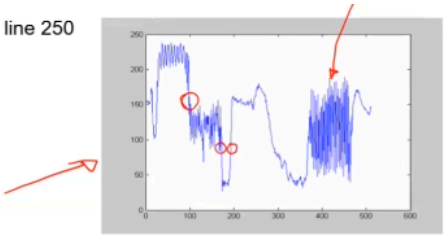
[2] **Aliasing** is an effect that causes different signals to become indistinguishable (or *aliases* of one another) when sampled. It also often refers to the distortion or artifact that results when a signal reconstructed from samples is different from the original continuous signal.

One common idea and approach (in the 80') is the matching of models (wire-frame, geons, generalized cylinders...) to edges and lines. We still have a problem that is extracting edges from these models we've created. It is useful to operate with signals: given an image we can take the signal of a certain line in the image. Edges are usually found when we have a strong curve in the signal (that correspond to a sudden change of color). If there is an area with much noise (that makes difficult to locate edges), it is possible to smooth the image with a Gaussian in order to make the edges more readable (e.g. take a look in the area with noise that corresponds to the scarf area. Edges are difficult to find there. Just smooth the image and we'll get different signals where the edges of the scarf are easier to distinguish).

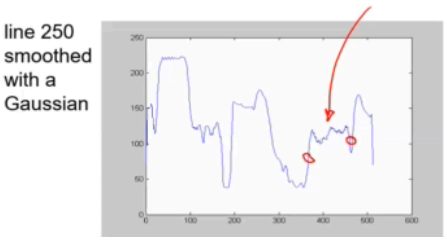
- Barbara Image:
  - entire image



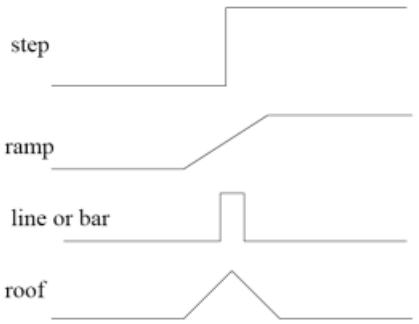
- line 250



- line 250 smoothed with a Gaussian



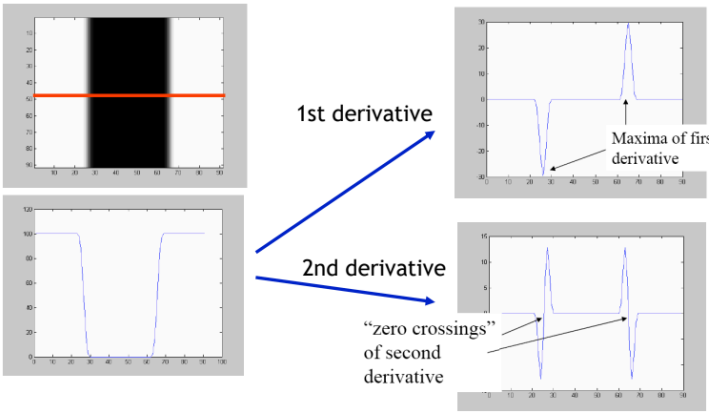
There are different types of idealized **edge types**: Step, Ramp, Line (or Bar) and Roof. Notice that in the real world we'll probably see only ramp and roof edges (not step and line ones).



When making Edge Detection, the main goals should be:

- Good detection: filter responds to edge, not to noise;
- Good localization: there should be a response edge exactly where the edge is and not around the area;
- Single response: one per edge (that is likely to be represented by one pixel);

Note that edges correspond to fast changes where the magnitude of the derivative is large so **if there is an edge and we smooth the image**, then the **edge will still be there**.

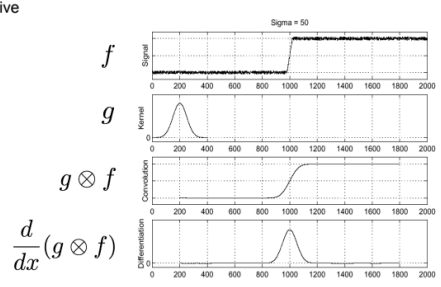


If we take the first derivative of the smoothed image, the max and min points correspond to edges.
If we take the second derivative, edges correspond to "**zero crossings**"[3].
So the derivative is the solution to find the edges in the image. Let's make a recap of what a derivative is:

$$\frac{d}{dx}f(x) = \lim_{h\to 0}\frac{f(x+h)-f(x)}{h} \approx f(x+1)-f(x)$$

This is an easy operation to implement with a convolutional linear filter: image having a function f, just take a point x+1 with coefficient 1 and x with coefficient -1. If we want to make it symmetric we can put a 0 between -1 and 1. This is useful when we have an even number of positions in the filter and we don't want to misalign the first derivative. So we add a 0 so that we have an odd number and the alignment is preserved (between the original image and the differentiation). The zero

- based on 1st derivative:
  - smooth with Gaussian
  - calculate derivative
  - finds its maxima



[3] A **zero-crossing** is a point where the sign of a mathematical function changes (e.g. from positive to negative), represented by an intercept of the axis (zero value) in the graph of the function.

that we add doesn't change the computation. To sum up we can simply implement the first derivative as a linear filter:

- Direct: -1 1;
- Symmetric: -1 0 1;

So, for what concerns the first derivative: we take the signal (which is represented by a function f) and we want to remove any trembling in the signal (where there is a high frequency noise) in order to detect any possible edges. So what we do is smoothing by applying a gaussian kernel. The output is the smoothed signal. After that we apply the derivative from which we can find the edge (it will be the max and the min of the derivative).

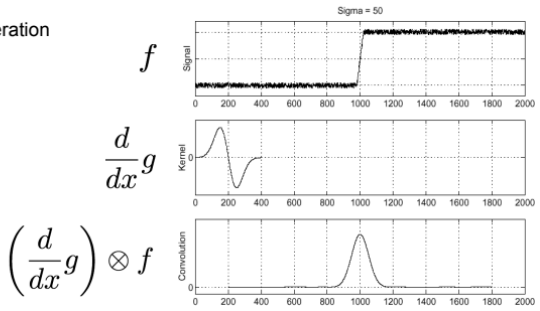
Note: Remember that derivatives as convolution are linear operations so we can save one operation by calculating the derivative of the kernel g and then applying the convolution between its result and the function f. Doing so, we improve the efficiency since we do many fewer multiplications.

The first derivative introduces a **new problem** which is the **amplification of higher frequencies** (e.g. see the frequencies that represent the scarf).
So, how can we actually implement 1D edge detection?
Algorithmically we have to **find a peak in the 1st derivative** but **it should be a local maxima** and **it should be sufficiently large**. What to can do is to use 2 thresholds[4]:

- High threshold to start edge curve (maximum value of gradient should be sufficiently large);
- Low threshold to continue them (in order to bridge "gaps" with lower magnitude);

This operation is called **hysteresis**. So, if we have a strong edge we'll going to keep it but if we have two strong edges and nothing in the middle then it is unlikely that an object does not have a continuous edge between the two (e.g. see the arm of the woman in the example above) and so it makes sense to stem a connection in between using a lower threshold. Note that we will always apply two thresholds and we will always consider points of the contours (of the 2nd threshold) only if we are in close contact with the high threshold (otherwise we will not simply consider them).

- Simplification: $\frac{d}{dx}(g \otimes f) = \left(\frac{d}{dx}g\right) \otimes f$
  - remember: derivative as well as convolution are linear operations
  - saves one operation



What we have seen so far it's the derivative in the x direction but since we have a 2D image we have also to consider the y direction

- in x direction: $\frac{d}{dx}I(x,y) = I_x \approx I \otimes D_x$
- in y direction: $\frac{d}{dy}I(x,y) = I_y \approx I \otimes D_y$

We often approximate the partial derivative with these filters:

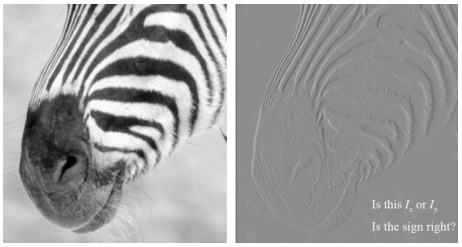
$$D_x = \frac{1}{3}\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \qquad D_y = \frac{1}{3}\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

EX. It is easy to see if we are segmenting the image vertically or horizontally. In the example below we're considering I_x in fact all the **vertical edges are preserved**. The sign is correct in fact if we apply the filter Dx, when mirroring it we obtain [1, 0, -1]. Now, if we make the product we obtain 1*1+0*0+(-1)*(-1) =2 and this is correct in fact the output of the convolution is 0 2 0 (note that 2 is a point of maxima). So detecting the edges along the x axis means the vertical edges will emerge (and the inverse with y axis).

---

[4] Threshold: **thresholding** is the simplest method of segmenting images. The simplest thresholding methods replace each pixel in an image with a black pixel if the image intensity $I_{i,j}$ is less than a fixed value called the threshold T, or a white pixel if the pixel intensity is greater than that threshold. In most methods, the same threshold is applied to all the pixels of an image. However, in some cases, it can be advantageous to apply a different threshold to different parts of the image (like in edge detection), based on the local information of the pixels. This category of methods is called local or adaptive thresholding.

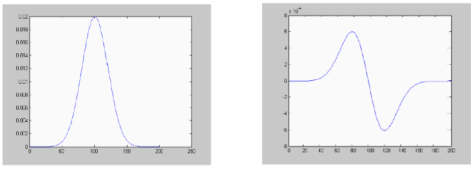
Again, if we have an image and we calculate the first derivative we could have a strong noise. What we can do to reduce it is to convolve the image with the 2D gaussian (1D gaussian if we have a 1D image) in order to blur the image and then we apply the first derivative. Again, notice that we can also apply the first derivative to the 2D gaussian and then apply it to the image in order to improve efficiency and this is granted to the associative property of linear systems.



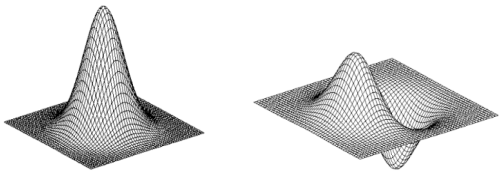
EX. Edge detection over the x and y axes:

derivative in x-direction: $D_x \otimes (G \otimes I) = (D_x \otimes G) \otimes I$

- in 1D:



- in 2D:



What is the gradient?
If we take the derivative along the x in the yellow spot, it will be some value k. If we take the derivative along the y in the same point then we're going to have a 0.
What happens in most of the edges in the image is that they are not going to be x or y aligned but they'll probably have angles. If we compute the two derivatives we'll have two values k_x and k_y. The vector (k_x, k_y) is the **gradient of the vector**. The **gradient direction** is **perpendicular to the edge**. The **gradient magnitude** is proportional to the **strength of the edge**.

So, a **way to detect edges** could be **using the magnitude of the gradient**.
The gradient at each pixel is:

$$\nabla I = \left(I_x, I_y\right) = \left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right)$$

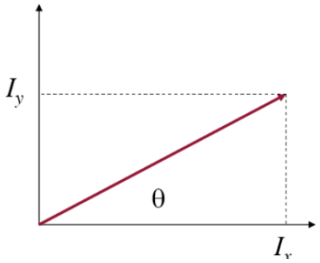
The magnitude (norm) of the gradient is:

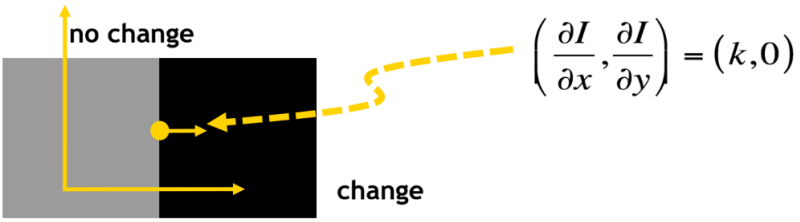
$$\|\nabla I\| = \sqrt{I_x^{\,2} + I_y^{\,2}}$$

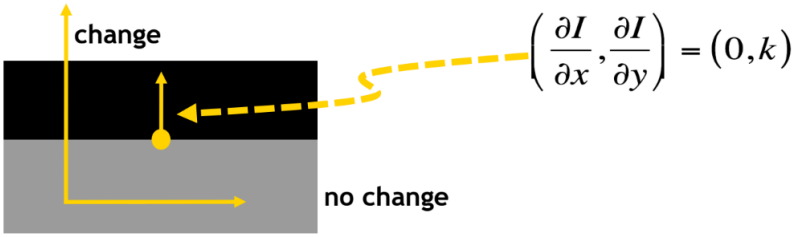
The direction of the gradient is:

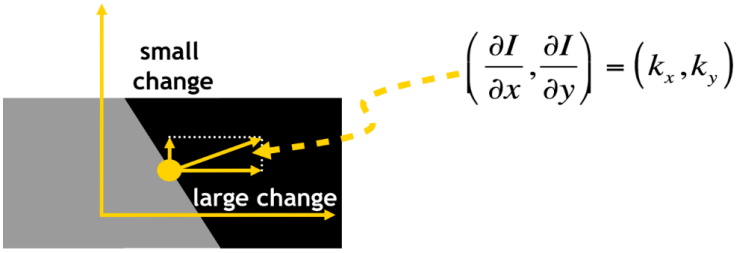
$$\theta = \arctan\left(I_y, I_x\right)$$

So, basically we convolve the image with the gaussian in order to blur the image, then we compute the first derivative along the x and along the y. This gives us the two components k_x and k_y from which we can compute the gradient and the relative magnitude. From the magnitude we get the edges along both directions.



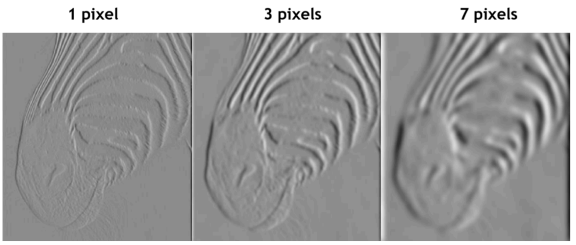


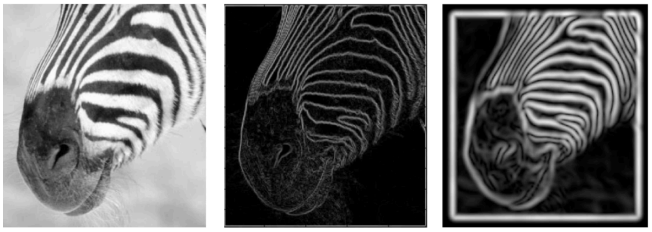




Before continuing, the scale of the smoothing filter affects derivative estimates, and also the semantics of the edges recovered because by increasing smoothing only the strongest edges remain (e.g. if we set σ=7 then we'll focus on larger scale edges).
Now, it is not so straightforward in fact there are 3 major issues:

- The gradient magnitude at different scales is different: which to choose?
- The gradient magnitude is large along a thick trail: how to identify the significant points?
- How to link the relevant points up into curves?

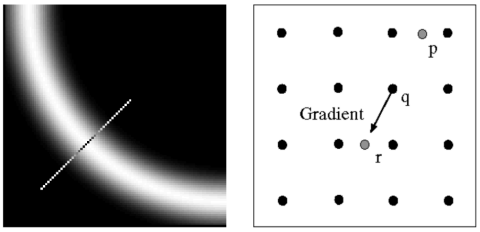


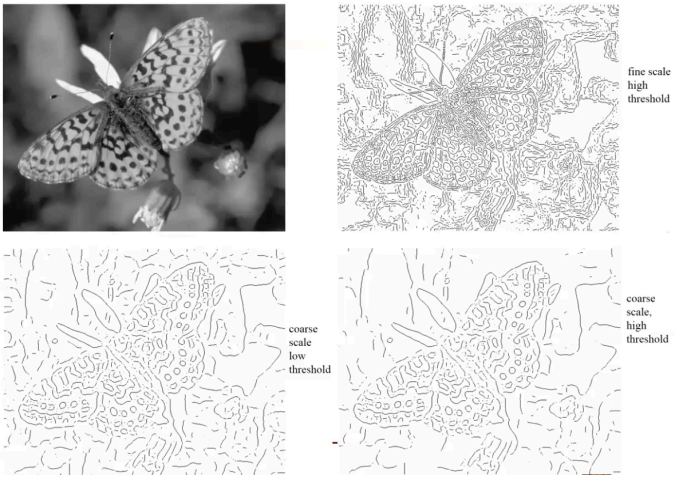


So a procedure has been proposed to make edge detection. An **optimal edge detection** (assuming linear filtering and gaussian noise to be additive) should have good detection, good localization and single response (as we said above) then → an optimal detector is approximately the derivative of a Gaussian. There is not a solution that goes well for every image. There is a tradeoff regarding detection and localization which is that **more smoothing improves detection but hurts localization** (e.g. if we want to remove noise we can smooth using a larger sigma but will hurt localization because trails (contours) will be thicker and we don't want that, the trail should have just 1 pixel).

One of the most important recipe in CV is the **Canny edge detector** that follows these steps:

1. Given the original image;
2. We apply the gaussian filter to smooth the image and remove the noise. Then, we compute the **magnitude of the gradient** in order to locate edges at all possible angles (see steps above);
3. We will consider edges only those that are above a certain **threshold**. So we apply **thresholding** to remain with **strong edges** that are between objects.
4. After that we **thin the thick edges** (since we want edges of 1 pixel) with the **non-maximum suppression** technique. What we do is **check if the pixel is local maximum along gradient direction**. We then choose the largest gradient magnitude along the gradient direction. It requires checking interpolated pixels p and r.
5. Finally we apply a double threshold to determine potential edges (so to distinguish between weak and strong pixels) and hysteresis in order to suppress all edges that are considered weak and not connected to strong edges, and transform the weak pixels into strong ones iif at least one of the neighbor pixels is strong.





EX. Butterfly example: starting from the image of a butterfly, if we apply a fine scale (so small gaussian) and a high threshold we obtain the main edges. If we now apply a coarse scale (so we choose a higher σ) then we obtain the variations that occur over more pixels. We can also lower the threshold.

Let's take another step ahead: we said that we could compute the first derivative and look for edges where we have maximas. We also said that we could compute the second derivative and look for edges when we have "**zero crossings**". It is actually nice to have zero crossings instead of maximas and minimas because in the first derivative when we have to take the point above a certain threshold, we usually end up with a thick line and then we have to make it smaller through non-maximum suppression. In the second derivative, we just have to look for zero crossings that it's just where the image goes from black to white (so we have less computations).
So, let's give the definition of **2nd derivative**:

$$\frac{d^2}{dx^2}f(x) = \lim_{h\to 0}\frac{\frac{d}{dx}f(x+h) - \frac{d}{dx}f(x)}{h} \approx \frac{d}{dx}f(x+1) - \frac{d}{dx}f(x)$$

$$\approx f(x+2) - 2f(x+1) + f(x)$$

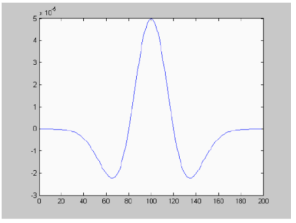
2nd derivative:

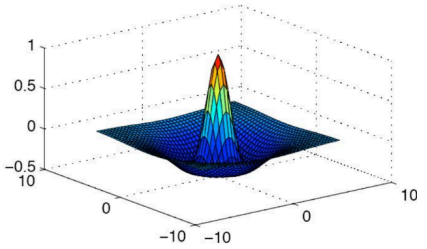
1	−2	1

This means that we can take the 2nd derivative to be a linear convolutional filter for which the kernel values are given by (1 -2 1).
In order to locate the edges we use **The Laplacian** that means applying the 2nd derivative to the x, to the y and sum the results. Again, we don't want to apply the Laplacian directly on the image, we first smooth the image with a gaussian G but, for efficiency reasons, since the Laplacian is a linear filter we can combine it with a Gaussian and then apply the result to the image (again we can do it thanks to the associative property):

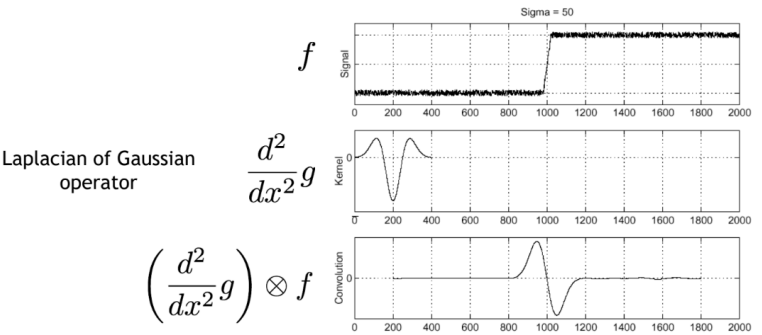
$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

$$\nabla^2\left(G \otimes f\right) = \quad \nabla^2 G \otimes f$$





So, let's now check how **1D edge detection** works:
f is the signal (note that we have a high frequency noise). What we do is calculating the second derivative of a gaussian and convolve it with the signal f. What we obtain is the actual edge that is represented by a **zero crossing**.



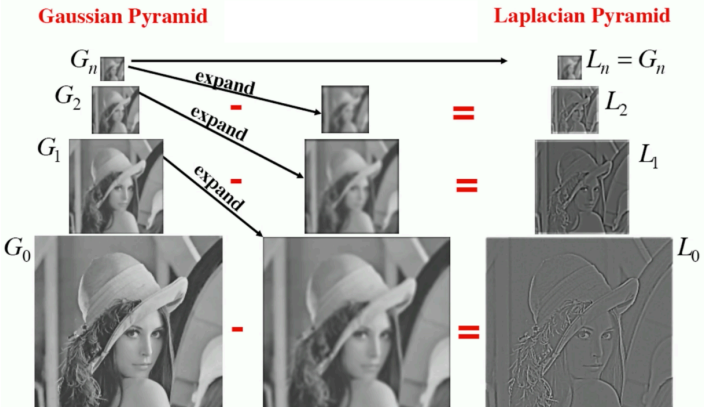
Now, we can approximate the Laplacian by expressing the 2nd derivative of a gaussian of the **difference of gaussians** (DoG). We do that because we want to look for edges at **different scales**. One thing that might be useful to remember is that applying the Gaussian two times is equivalent to applying the Gaussian one time but with a larger kernel. We saw earlier on how to create the Gaussian Pyramid, <u>how do the Laplacian Pyramid work?</u>
The way of constructing the **Laplacian Pyramid** is to subtract the fine scale and coarse scale. So basically, starting from an image G_0, what we do is downsampling the image by applying a Gaussian with a larger kernel getting G_1. Again, from G_1 we can convolve it again with a Gaussian with a larger kernel to get G_2 and so on and so forth. In order to get the Laplacian, we just have to apply this formula:

$$L_i = G_i - \text{expand}(G_{i+1})$$

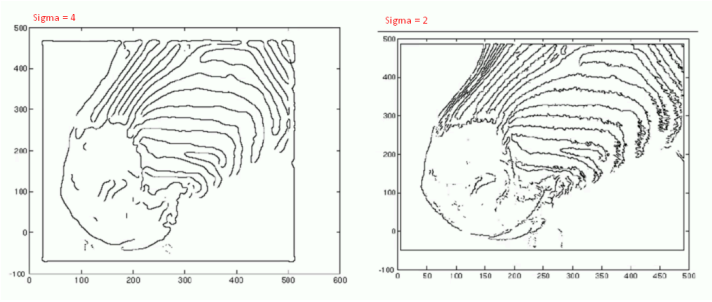
So, get an image and we subtract it with its expanded version.
If we have the Laplacian at scale i the expanded image at scale i+1, we can also retrieve the original image by applying:

$$G_i = L_i + \text{expand}(G_{i+1})$$

EX. In order to get L_0 we subtract G_0 and expand(G_1).



EX. So, these are the edges that we have with a different sigma of the second derivative: remember that before applying the second derivative we have to smooth the image with a Gaussian. If we smooth with a larger sigma we recover the coarse edges. If we lower the laplacian pyramid we recover the small details (also the hairs of the zebra).



RECAP: In order to find edges in and image we can use first derivatives in particular we have to look for maxima and minimas in the signal. Otherwise we can use the second derivatives and edges will be zero crossings. For the first derivative, we have to first smooth the signal so that the computation doesn't increase the noise and after that we convolve the result with the first derivative (note that for the associative property we can convolve gaussian and the derivative so that we obtain smoothing and derivative filter to be applied altogether). Now, we now have an actual filter for the x and the y derivative that can be used to look for vertical and horizontal edges. We want to detect edges that cross all possible directions so we use the Canny edge detector. So given the image, we calculate the norm (=magnitude) of the gradient. Now, given the resulting image, we would like to get rid of parts of the image that don't contain edges so we apply the thresholding. We'll probably get thick edges so we apply the non-maximum suppression in order to make edges thinner (preferably 1 pixel). There is also another step called hysteresis: if we have two near edges and nothing in between, we can use a lower threshold in order to align the two and connect the edges.
Now, it is clear that edge detection with the first derivative takes a lot of steps so it may be smarter to use the second derivative. Again, we have to combine the gaussian filter with the second derivative (not the first derivative as before) resulting in the filter that also smooths the image. We then have to apply the filter to the image by convolution. Now, we apply it in the x and the y and edges will be the zero crossings. If we want to find strong and fine edges in the image, we have to apply the filter with different levels of smoothing (depending on the σ in G) but this is computationally costly especially for larger images. There is a trick which consists in approximating the Laplacian with a difference of Gaussians (DoG). We already know how to get all scales of gaussian smoothing with the gaussian pyramid (we just have to convolve an image with a certain sigma and then down-sampling the image, repeating the process always with a larger sigma). The Laplacian Pyramid is just the difference of gaussians: in particular we have to subtract an image with the expanded version of the image after having applied a Gaussian with a large filter that smoothed and subsampled the image.

3.4 Hough Transform

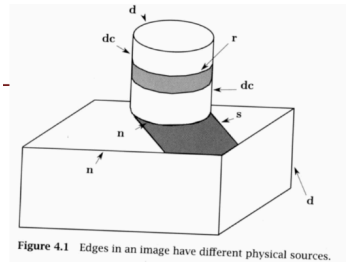


Figure 4.1 Edges in an image have different physical sources.

First of all, let's analyze what it means for finding an edge. What's an edge?

- object-background boundaries (edges (d) and depth discontinuity (dc));
- object-object boundaries (those correspond not necessarily to object boundaries, (n));
- shadows (s);
- discontinuities of object texture (r);
- discontinuities of surface normals (n);

We are interested in every type of edge except for shadows (s) and discontinuities of object textures (r) (e.g. I want to recognize people even if they have different t-shirts, so different textures).
Another problem is the combination between edges and contours: edges are defined as discontinuities in the image. We can assemble them, to obtain corresponding object contours but contours do not necessarily correspond to object boundaries. There is basically no knowledge used how object contours look like, obviously humans use such knowledge to segment objects.
In principle: if we knew which object is in the image it would be much simpler to segment the object.

So we want to detect all the straight lines that can explain all the edge pixels in the image and this is useful because most of the contours are straight lines. Let's say that a line can be represented by:
$y = ax + b$
where:

- We have 2 parameters: a and b that determine all points of a line;
- This corresponds to a transformation: (a,b) -> (x,y): $y = ax + b$;

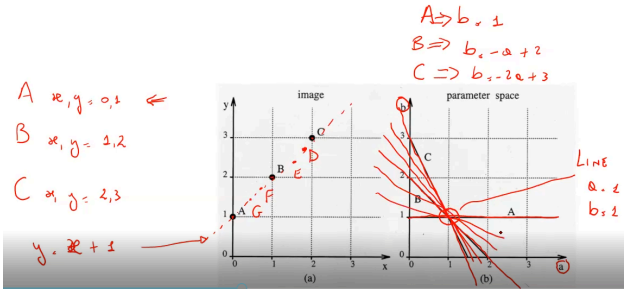
Now, we can make an inverse interpretation so a transformation of (x,y) -> (a,b): b = (-x)a + y (a becomes the independent variable and we get b which is a dependent variable).

What's the usage?

Points for which the magnitude of the first derivative is large lie potentially on a line.

EX. For a particular point (x,y) determine all lines which go through this point:

- the parameters of all those lines are given by: b = (-x)a + y;
- I.e. those lines are given by a line in the parameter space (a,b);



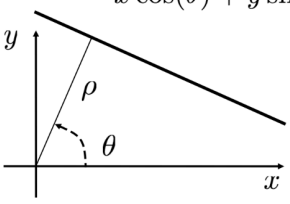
What we want is a procedure to fit a line where points A, B, C pass. So, given the coordinates of a point, just apply the equations and you'll get the points. For example, given A = (0,1) (because x=0, y=1) then b = (-0)a + 1 = 1. Given B = (1,2) then b = -a + 2. The intersection of the 3 lines is the point that corrects to the straight line that passes through A, B and C. That point (let's say I) has coordinates (1, 1) (a=1, b=1). Now, given the equation y=ax+b and substituting the values we obtain y=x+1 which is exactly the line that passes through A, B and C.

So, what's the implementation of Hough Transformation?

- The parameter space (a,b) has to be discretized;
- For each candidate (x,y) for a line, store the line b = (-a) x + y;
- In principle each candidate (x,y) votes for the discretized parameters;
- The maxima in the parameter space (a,b) correspond to lines in the image;

There is a problem with this particular parameterization in fact the parameter "a" can become infinite (for vertical lines) and we can't afford this into the parameter space. So we can avoid this with a different polar representation of points, so for each of the points instead of adding a and b, we parametrize the line as the distance between the line and the origin obtaining an angle Θ. This allows us to represent all possible lines without the problem of infinite values.

$$x\cos(\theta) + y\sin(\theta) = \rho$$



for this parameterization the domain is limited:

- ρ is limited by the size of the image
- and $\theta \in [0, 2\pi]$

Note: the same idea can be made for irregular surfaces (not covered in the course).

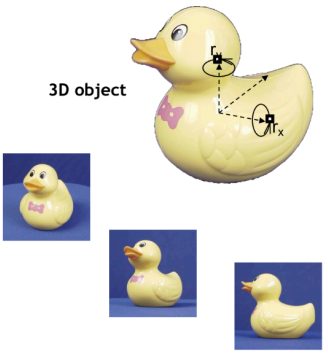
3.5 Object Instance Identification using Color Histograms

To introduce this subject, just remember the difference between:

- Object Identification: recognize your apple, your cup, your dog;
- Object Classification: recognize any apple, any cup, any dog;

Identifying a specific object from a database of different objects can be a trivial operation, because that object can be represented into different forms, shapes, characteristics and so on. In particular there are some **models of variations** that can be distinguished into:

- Viewpoint changes:
  - Translation (position in the image of the object);
  - Image-place rotation (rotation in place of 360° of the object);
  - Scale changes (different scale of the object);
  - Out-of-plane rotation (more difficult rotation that consists in rotating both over the x and the y axis);
- Illumination;
- Cluster;
- Occlusion (We get to see something that we don't know about. In an image it occurs when an object hides a part of another object);
- Noise (It can be a problem when, for example, we have to detect edges);



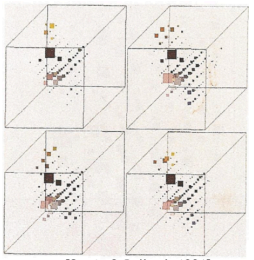
So, in order to identify or recognize objects, we make some assumptions that objects can be represented by a collection of images called **appearances**. For recognition, it is sufficient to just compare the 2D appearances (if we have many appearances of an object then no 3D model is needed).

We need a way of representing objects, in particular each appearance of an object: we can do it with (global) **descriptors** which are no more than vectors. In order to recognize an object, we just have to match it with the (global) descriptors within a dataset. The idea is: If we have a descriptor, we can use it to match an object that has the same id so that it is possible to recognize it. **Models of variations** can be taken care of by:

- Built into the descriptor (e.g. a descriptor can be made invariant to image-plane rotations, translation - so for example the color histogram doesn't change if the object rotate);
- Incorporate in the training data or the recognition process (e.g. viewpoint changes, scale changes, out-of-plane rotation - here the color histogram would change so we must have different images of the same objects in order to recognize a particular instance of an object);
- Robustness of descriptor or recognition process (the recognition process is made matching the descriptors) (e.g. illumination, noise, clutter, partial occlusion - here there are other operations that can be accomplished to recognize an object since having many images of the objects (with for example different illumination) can be computationally expensive);

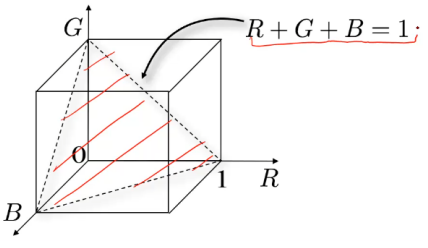
To sum up, we have to build a descriptor that has certain invariances, consider enough data and then assume that the recognition process can deal with invariances.

So, what can we use for recognition?
Color: It stays constant under geometric transformations (e.g. translation, rotation, etc..) and it is defined for each pixel. It is also robust to partial occlusion since, according to the assumption that objects are smooth, even if the portion of the object is covered then the other pixels probably have the same statistics (the covered pixels will be more or less the same as the visible ones). So, the idea is to directly use object colors for identification/recognition (or better: we would like to **use statistics of object colors** using **color histograms** since we would be interested in knowing how many pixels are of a certain degree of a color in the image).
What do we do?
We use **histogram** as the descriptor to describe a particular view of a particular instance. So, given different instances of an object, they're going to have different histograms and we can choose the closest one to the object we want to recognize.
Given R,G,B for each pixel, compute 1D histograms for the R, G and B, as well as for the luminance (E.g. Hist(R) = #(pixels with color R)). Better, rather than have 3 histograms we could build a 3D histogram (E.g. H(R,G,B) = #(pixels with color (R,G,B))). Notice the luminance histogram in no more that a gray-scale histogram (the image pixels are represented by a number in the interval [0, 255] and we want to count how many pixels of the same intensity are in the image).



[Swain & Ballard, 1991]

EX. The image on the left shows how this representation is **robust**: given different instances of the same object (even an occluded one), the histograms are all similar.

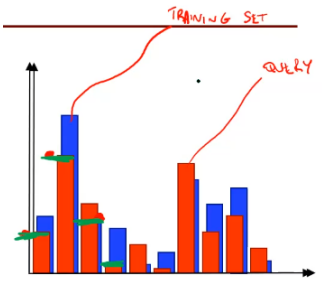
Now, how do we deal with color intensity (illumination)?
One component of the 3D color space is intensity. If a color vector is multiplied by a scalar, the intensity changes, but not the color itself. This means **colors can be normalized by the intensity**. So, we compute the intensity:
I = R+G+B
and we normalize it with these formulas:
r=R/(R+G+B)
g=G/(R+G+B)
b=B/(R+G+B)
An important observation: since r+g+b=1 then only 2 parameters are necessary.
EX. Since r+g+b=1, then one can use r and g and obtain b = 1 - r - g.



What do we do with these histograms?
We take a test image, we compute his histogram and we use it to match the test image with the descriptors of all the known objects of a database. The closest histogram will eventually be the one of the object we're trying to match. Remember that there should be multiple training views per object in order to recognize it: that's because

our object could have different orientation, shape and so on (for example, given a paper duck we would like to have more images with different position or orientation of the duck).

So, we need to quantify the similarity between histograms in order to choose the closest one to our test image. There are some **comparison measures for histograms**:



1. One that turns out to work quite well is the **intersection** among two histograms. So, given a query Q and the training set V, the intersection is computed like this:

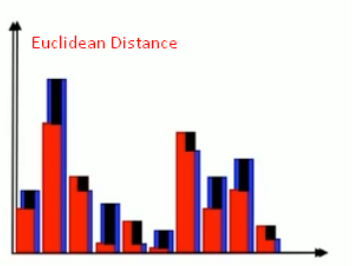
$$\cap(Q,V) = \sum_i \min(q_i, v_i)$$

So, for each bin i in the two histograms just take the two side by side and the minimum between the two. The sum of all those min values will be the intersection and it represents the similarity between the two.
So, the intersection measures the common part of both histograms in a range [0,1] (0 they are completely different while 1 means that they match perfectly). Note that the formula above is valid for normalized histograms: the reasons for normalize a histogram are:

- It can represent some sort of probability that they match;
- If the resolution gets changed, histograms are still comparable;

For **unnormalized histograms**, use the following formula:

$$\cap(Q,V) = \frac{1}{2}\left(\frac{\sum_i \min(q_i, v_i)}{\sum_i q_i} + \frac{\sum_i \min(q_i, v_i)}{\sum_i v_i}\right)$$



2. Another comparison measure is the **Euclidean Distance**:

$$d(Q,V) = \sum_i (q_i - v_i)^2$$

Here we have the opposite situation in fact the closest match will have a distance of 0 while the furthest match will have the highest distance. So the Euclidean Distance focuses on the differences between the histograms and the range is the following: [0,∞]. All cells are weighted equally (it doesn't matter if there are more pixels of a certain color in respect to another one because they are weighted the same and this could be a problem when doing object recognition).

3. Another one is the χ^2 (**Chi-Square)**:

$$\chi^2(Q,V) = \sum_i \frac{(q_i - v_i)^2}{q_i + v_i}$$

This measure has a statistical background in fact we want to test if two distributions are different. The result will be 0 when the two histograms are the same or a larger value if they are different from each other (range is [0,∞] as well as the Euclidean Distance). The difference with the Euclidean distance is that here cells are not weighted equally (the comparison between two histograms depends on the amount of pixels of a certain color that is if there are many pixels of a certain color then those are down weighted while if there are many pixels of a certain color then those are up weighted). So, this method is **more discriminant** than the others.
There may be problems with outliers (therefore assume that each cell contains at least a minimum of samples).

Which measure is best?
It depends on the application. Both Intersection and χ^2 give good performance. Intersection is a bit more robust. χ^2 is a bit more discriminative. Euclidean distance is not robust enough. There also exists other measures.

So, now that we know some comparison measures let's give an **algorithm to make the recognition using histograms**:

1. Build a set of histograms H = {M_1, M_2, M_3, ...} for each known object (more exactly, for each view of each object);
2. Build a histogram T for the test image;
3. Compare T to each histogram $M_k \in H$ (using a suitable comparison measure);
4. Select the object with the best matching score (or reject the test image if no object is similar enough (distance above a threshold t));

Recognition (object identification) with color histograms usually works really well. Some pros and cons:

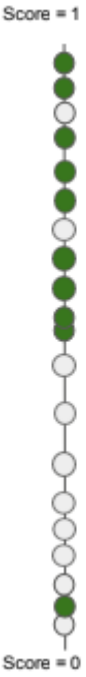
- Pros:
  - Invariant to object translations;
  - Invariant to image rotations;
  - Slowly changing for out-of-plane rotations;
  - No perfect segmentation necessary;
  - Histograms change gradually when part of the object is occluded;
  - Possible to recognize deformable objects (e.g. the pullover example);
- Cons:
  - The pixel colors change with the illumination (e.g. color constancy problem), intensity and spectral composition (illumination color);
  - Not all objects can be identified by their color distribution;

3.6 Performance Evaluation

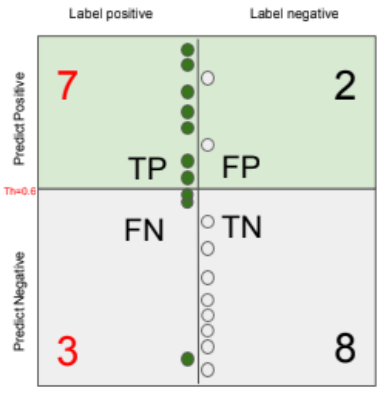
Performance evaluation gives us the tools to understand if a comparison method is better than another. How can we say if method A is better than method B for the same task? We could:

1. Compare a single number - e.g. accuracy (recognition rate - how many time the method actually recognize an object), top-k accuracy;
2. Compare curves - e.g. precision-recall curve, ROC curve;

For what concerns the score-based evaluation, let's start by defining some terminology:
DEF. **The recognition algorithm identifies (classifies) the query object as matching the training image if their similarity is above a threshold t.**



Imagine we have to classify a number of images according to the similarity with the query object. We have a score between 1 (the highest) and 0 (the lowest) that measures the confidence of the fact that the object is in the image. The full spot is where the object is actually in the image while the full spot is where it's not in the image. The decision (if the object is in the image) depends on a certain threshold. We would like to have empty spots at the bottom of the score and full spots at the top.
So, we can split the different cases into 4 quadrants: images can have a positive label (if they actually contain the object) or negative label (otherwise). We can predict positive or negative. Quadrants represent all possible cases. **Quality of model & threshold decide how columns are split into rows. We want diagonals to be "heavy", off diagonals to be "light".**



So, among the images we have:

- **TP**: True Positive (there is the object, we predict that there is - I predict as positive and the label is actually positive);
- **TN**: True Negative (there isn't the object, we predict that there isn't - I predict as negative and the label is actually negative);
- **FP (Type I error)**: False Positive (there isn't the object, we predict that there is - I predict as positive and the label is actually negative);
- **FN (Type II error)**: False Negative (there is the object, we predict that there isn't - I predict as negative and the label is actually positive);

We would like to have a **higher number of TP and TN** while there should be **low FP and FN**.

There are a few performance measures we can use to say how good an algorithm is.
The **overall accuracy** tells us how many correct classifications we do over the set of tests.

The **precision** tells us how precise/accurate the model is out of those predicted positives, how many of them are actual positives (e.g. among images that we say contain the object, how many of those actually contain it, and how precise the model is).
The **recall** (or **specificity**) calculates how many of the actual positives the model capture among the total number of tests labeled as positives (e.g. among images that contain an object, how many of those did we recognize):

$$\text{Overall accuracy} = (TN + TP)/N$$

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \text{True positive rate} = \frac{TP}{TP + FN}$$

Note: We have to choose our performance measure depending on the application. For example, in the example above we have a really high accuracy (99%) so we would say that the system is almost perfect at making predictions. But if the model has to predict if somebody is carrying a virus then we would be more interested in the only one FN (so maybe we should consider other measures like recall). Also, there is a trade-off between precision and recall because typically the one reduces the other and vice versa. Usually increasing the threshold, false positives get reduced and false negatives get increased and this situation causes the precision to increase and the recall to decrease (a specular situation happens if decreasing the threshold). So, again, we should choose our measures carefully depending on the application (e.g. In a medical scenario, we would like to have too high recall because it means having a smaller number of false negatives (that is the worst situation it may happen for a patient because it would mean that some patients who are positive to a certain disease, are termed as falsely negative)).

		Predicted/Classified	
		Negative	Positive
Actual	Negative	998	0
	Positive	1	1

Note: Some observation

- Trivial 100% recall = pull everybody above the threshold;
- Trivial 100% precision = push everybody below the threshold except 1 green on top (Hopefully no gray above it!);
- Striving for good precision with 100% recall = pulling up the lowest green as high as possible in the ranking;
- Striving for good recall with 100% precision = pushing down the top gray as low as possible in the ranking;

False positive rate tells us how many positive result will be given when the true value is negative (e.g. among images that don't contain an object, how many of those we predict they contain it):

$$\text{False positive rate} = \frac{FP}{TN + FP} = \text{1-Specificity}$$

One measure is the **F1-score** is a measure of a test's accuracy and it is a balance between precision and recall. It measures the likelihood of success that is how many times the model is right in average:

$$F_1 = \left(\frac{2}{\text{recall}^{-1} + \text{precision}^{-1}} \right) = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

EX. Given the image above, the results for all metrics are:

Th	TP	TN	FP	FN	Acc	Pr	Recall	Spec	F1
0.5	9	8	2	1	.85	.81	.9	.8	.857

Note: these values **strongly depend on the threshold**. Despite the model, we should choose the best threshold that gives us the best results (e.g. in the last example it should be t=~0.45). In general the number of effective thresholds is equal to the number of examples + 1.

SUM. The recognition algorithm identifies (classifies) the query object as matching the training image if their similarity is above a threshold t. Compare actual outcomes to predicted outcomes using a confusion matrix (classification matrix).

	Predicted = 0	Predicted = 1
Actual = 0	True Negatives (TN)	False Positives (FP)
Actual = 1	False Negatives (FN)	True Positives (TP)

N = number of observations

$$\text{Overall accuracy} = (TN + TP)/N \qquad \text{Overall error rate} = (FP + FN)/N$$

$$\text{False positive rate} = \frac{FP}{TN + FP} = \text{1-Specificity} \qquad \text{Precision} = \frac{TP}{TP + FP}$$

$$\text{True positive rate} = \frac{TP}{TP + FN} = \text{Sensitivity} = \text{Recall}$$

So, we have now understood that **selecting the threshold t is an important operation**:

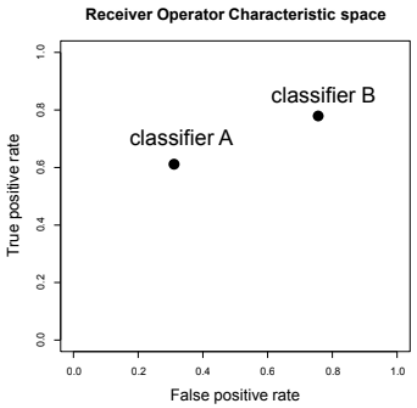
- The lower the t the more query images are classified as matching (More TP but also more FP);
- The higher the t the less query images are classified as matching (More TN but also more FN)

What value should we pick for t?

In order to have a representation of how the performance changes, changing the threshold, we can use the **Receiver Operator Characteristic (ROC)**, a 2D plot where we have the FPR in the x axes and TPR in the y-axes. Remember that:

- TPR: The larger the TPR, the larger the recall of actual true matches (lower threshold t);
- FPR: The larger the FPR, the larger number of false alarms (lower threshold t);

So, when we have a low threshold we're at the top right part of the plot, while with a high threshold we're at the left bottom part of the plot.



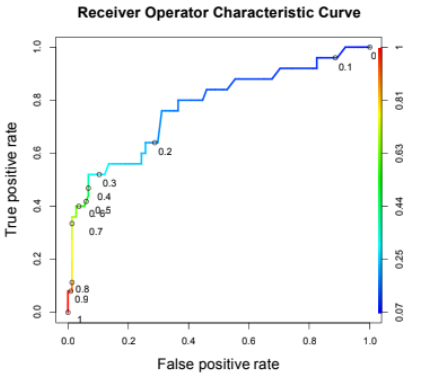
EX. The image on the right is an example of classification of two classifiers A and B with ROC. Notice that classifier A is better than B because even if it has a slightly lower TPR, it also has a lower FPR which is good.

So, we have to choose the **best threshold t** for the **best trade off**

- Cost of failing to identify an object;
- Cost of raising the false alarms;

How can we use the ROC curve to measure the performance of a method?

Given a method, we have to calculate the amount of TPR and FPR varying the threshold from 0 to 1. What we get is a curve (**ROC curve**) that tells us how to operate. Given a curve, we can calculate the **average performance of a method by taking the area under the curve**. The **area under the ROC curve (AUROC)** gives a rough idea on how the method performs independently on the threshold.



How can we compare different methods?

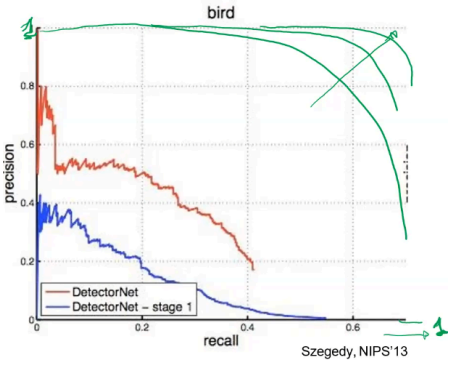
Once we have ROC curves for, let's say, 2 methods then we have to integrate the area under the curve for both methods and compare them. This area is going to be proportional to the number of times we get our decision right.

What is a good AUROC?

- Maximum of 1 (perfect prediction);
- Minimum of 0.5 (because the sum of TPR and FPR must be 1, if it's less then we're taking more wrong decisions then right ones);

Note: an AUROC of 0.5 corresponds to a diagonal curve (so a linear function) and corresponds more or less to random guessing.

Another curve is the **precision-recall curve** which is preferred for detection, where TN's are otherwise undefined (this means that if we have to detect an object in an image we will probably have one or few TP where we find that objects and many TN in all the other points where the object is not present). So, we are interested in TPR (recall) but we're not interested in FPR since it would always be near to 0 (there are many TN so the denominator is big). In general, we would like to have a curve that performs like the green one (so a curve that always has a high precision and recall).



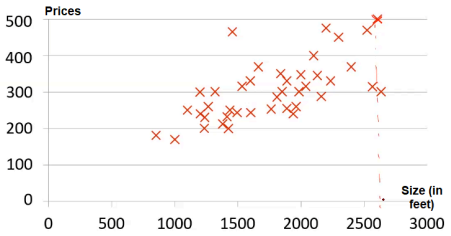
Another way of classifying methods is **confidence** (that the model is correct about a prediction). Basically we reward confident correct answers and heavily penalize the confident wrong answers. There are some measures that take into account the confidence of a method like **Log Loss** and **Brier Score**.

Note: $\hat{y}$ is the ground-truth while y is the prediction.

$$\text{Log Loss} = \frac{1}{N}\sum_{i=1}^{N} -y_i \log \hat{y}_i - (1-y_i)\log(1-\hat{y}_i)$$

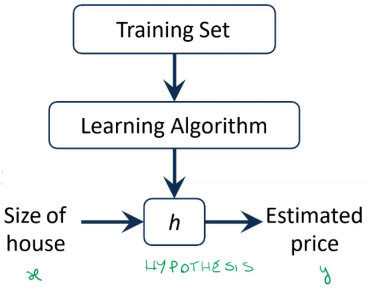
$$\text{Brier Score} = \frac{1}{N}\sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

4. Linear Regression



We can start talking about Linear Regression giving out an example on housing prices in the US. Let's imagine we have a plot like that on the right where we have all the houses sold depending on their price and size. What we want to do is to fit a line that would give the chance to be able to infer what is the cost of a house of a particular size. So basically, we will be able to predict the cost of a house given his size. Notice that we're making predictions on **real values** (e.g. 100, 2500, etc..) so this is a **regression problem**. So regression problems, like the one in the example, predict real-valued output, while **classification problems predict discrete-valued output**.
We're also in the field of **Supervised Learning** in fact we have the "right answer" for each example in data. Supervised machine learning requires labeled input and output data during the training phase of the machine learning lifecycle. This training data is often labeled by a data scientist in the preparation phase, before being used to train and test the model. Once the model has learned the relationship between the input and output data, it can be used to classify new and unseen datasets and predict outcomes. The reason it is called supervised machine learning is because at least part of this approach requires human oversight. The vast majority of available data is unlabelled, raw data. Human interaction is generally required to accurately label data ready for supervised learning. Naturally, this can be a resource intensive process, as large arrays of accurately labeled training data is needed. Supervised machine learning is used to classify unseen data into established categories and forecast trends and future change as a predictive model. A model developed through supervised machine learning will learn to recognise objects and the features that classify them. Predictive models are also often trained with supervised machine learning techniques. By learning patterns between input and output data, supervised machine learning models can predict outcomes from new and unseen data. This could be in forecasting changes in house prices or customer purchase trends. Contrary to Supervised Learning we have **Unsupervised learning** which is the training of models on raw and unlabelled training data. It is often used to identify patterns and trends in raw datasets, or to cluster similar data into a specific number of groups. It's also often an approach used in the early exploratory phase to better understand the datasets. As the name suggests, unsupervised machine learning is more of a hands-off approach compared to supervised machine learning. A human will set model hyperparameters such as the number of cluster points, but the model will process huge arrays of data effectively and without human oversight. Unsupervised machine learning is therefore suited to answer questions about unseen trends and relationships within data itself. But because of less human oversight, extra consideration should be made for the explainability of unsupervised machine learning.
The vast majority of available data is unlabelled, raw data. By grouping data along similar features or analyzing datasets for underlying patterns, unsupervised learning is a powerful tool used to gain insight from this data. In contrast, supervised machine learning can be resource intensive because of the need for labeled data.

Size in feet² (x)	Price ($) in 1000's (y)
2104	460
1416	232
1534	315
852	178
…	…



So, back to our example (remembering that we're in the field of supervised learning) we can give a notation to our training set. Given **m training examples**, we have the **input variable (x)** (also called **feature**) that represents the size of the houses, and the **output variable (y)** (also called **target variable**) that represents the price of the houses that we want to predict. A generic i-th training example can be represented as (x^i, y^i).
In this case we have a **Univariate Linear regression** that is a **Linear Regression with one variable x** but there may also be other features. In that case we talk about **Multivariate Linear Regression**: given n features we can represent as x^i the input (features) of i-th training example and x^i_j the value of the feature j in i-th training example.
EX. Image in the right: $x^2 = [1416, 3, 2, 30]$, $x^2_2 = 3$

Size (feet²)	Number of bedrooms	Number of floors	Age of home (years)	Price ($1000)
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178
…	…	…	…	…

m
n features

Now, given the training set we have to define a **Learning Algorithm** that will provide a **function h** (which stands for **hypothesis**). In the example, our hypothesis is the one that allows us to go from the size of a house to his price (so **h maps from x's to y's**).

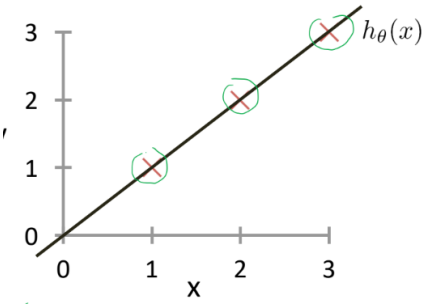
Given the training data, we have to fit a line that represents the training data.
How can we represent the hypotheses h?
We have to choose h parametrized by Θ (which is a vector) where the input is out feature x. The function is going to be equal to $\Theta_0 + \Theta_1(x)$ which are our parameters.
So: $\mathbf{h_\Theta(x) = \Theta_0 + \Theta_1(x)}$.
With multiple features we have:
$\mathbf{h_\Theta(x) = \Theta_0 + \Theta_1(x_1) + \Theta_2(x_2) + .. + \Theta_n(x_n)}$.
For convenience of notation, we define $x_0 = 1$ (for each training example).
Notice that Θ has to be an optimal value that is the one that minimizes the mistakes on the training set.
So we have:
$x = [x_0, x_1, .., x_n] \in R^{n+1}$ (n is the number of features where we've added x_0);
$\Theta = [\Theta_0, \Theta_1, .., \Theta_n] \in R^{n+1}$;
We have as many x's as the examples in out training set as well as the set of Θ's (in reality, the set of Θ is only one for all sets of x's).

How do we choose the parameters Θⁱ's?
Our function h is parametrized by Θ. **We need to choose Θ_0, Θ_1 so that $h_\Theta(x)$ is close to y for our training examples (x, y)**. Given the hypotheses $h_\Theta(x) = \Theta_0 + \Theta_1(x)$ with parameters Θ_0, Θ_1 we have to use the **cost function J**:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

TRAINING EXAMPLES

Our **goal** is to **minimize $J(\Theta_0, \Theta_1)$**.
Note: there is also a simplified case where we don't have the parameter Θ_0 and we just have to minimize $J(\Theta_1)$.
Note: note that the hypothesis is parametrized by Θ but it is a function of x so it gives us a value of the target variable for each of the input, while the cost function is function of Θ and gives us the cost for each set of parameters and this cost is given by considering all the training examples.

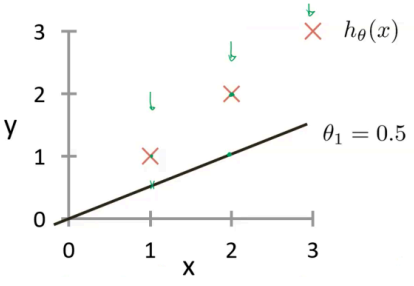


EX. We consider our $\Theta_1 = 1$ and $\Theta_0 = 0$ (so we're in simplified case) and $h_\Theta(x) = x$.
Let's compute the cost of Θ_1:

$$J(\Theta_1) = \frac{1}{2m} \sum^{m} \left(h_\Theta(x^{(i)}) - y^{(i)}\right)^2$$
$$= \frac{1}{2\cdot 3} \left(0^2 + 0^2 + 0^2\right) = 0$$

The computation between parentheses is the discrepancy between hΘ(xi) and yi so 1*1-1 = 0

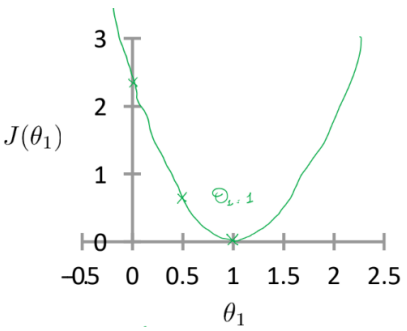
So, $J(\Theta_1=1) = 0$.
We now consider $\Theta_1 = 0.5$ and $\Theta_0 = 0$:



$$J(0.5) = \frac{1}{2\cdot 3} \left(0.5-1\right)^2 + \left(1-2\right)^2 + \left(1.5-3\right)^2$$
$$= \frac{1}{2\cdot 3} \left(\frac{1}{4} + 1 + \frac{9}{4}\right) \simeq 0.58$$

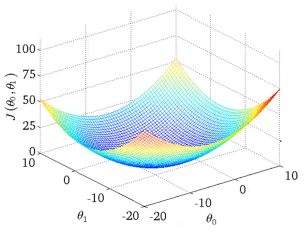
So, $J(\Theta_1=0.5) = 0.58$.
Let's do it again for $\Theta_1 = 0$ and $\Theta_0 = 0$:

$$J(0) = \frac{1}{6}\left(1^2 + 2^2 + 3^3\right) \simeq 2.3$$



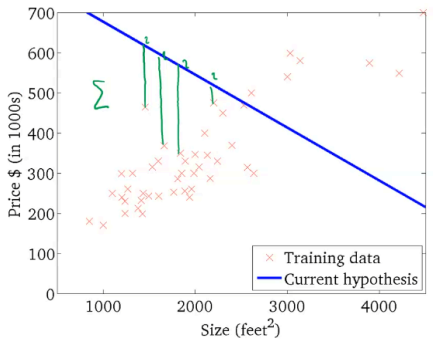
We can now **plot the results** giving the function of the parameter Θ_1 (that is the function $J(\Theta_1)$): remember that the the initial goal was to minimize $J(\Theta_1)$ (supposing that with are in the simplified case and Θ_0=0) so, from the plot, it is clear that we have to choose Θ_1=1 that minimizes the error on training set.

What happen if Θ_0 is not 0 (so we're in the general case)?
In case we have more parameters to consider (so Θ_0, Θ_1), the result of $J(\Theta_0, \Theta_1)$ won't be a parabola (like in the previous example) but a **2D function**. We can represent this 2D function in terms of a **contour plot** that is a line where all the points have an equal value of the function (so the value of J is the same).

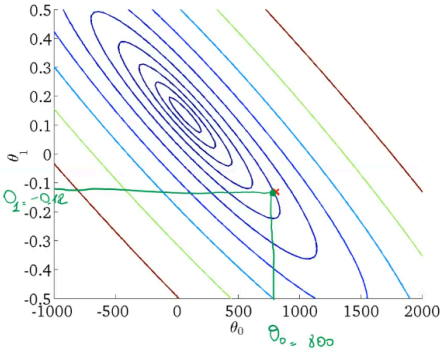


EX. Let's give an example of a general case. We can compute $J(\Theta_0, \Theta_1)$ using the previous formula. So, for example, given Θ_0=800 and Θ_1=-0.12 we'll have an hypotheses of $h_\Theta(x)$=800-0.12(x) that corresponds to the following line. We can represent $J(\Theta_0, \Theta_1)$ in a contour plot just placing the point corresponding to (Θ_0, Θ_1) with the given Θ_0, Θ_1 in the plot.

$h_\theta(x)$ (for fixed θ_0, θ_1, this is a function of x)

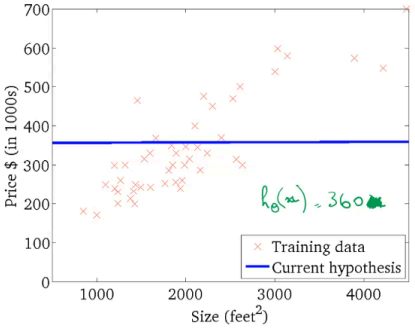


$J(\theta_0, \theta_1)$ (function of the parameters θ_0, θ_1)

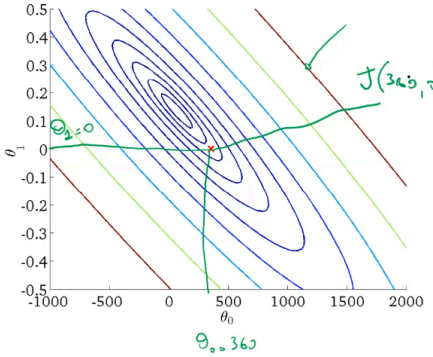


Another example with Θ_0=360 and Θ_1=0:

$h_\theta(x)$ (for fixed θ_0, θ_1, this is a function of x)



$J(\theta_0, \theta_1)$ (function of the parameters θ_0, θ_1)

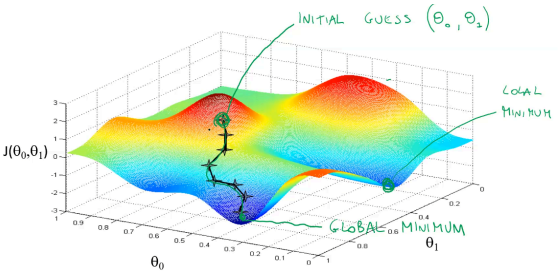


As before, we have to minimize $J(\Theta_0, \Theta_1)$ (which is represented by the same function as before) we have to choose those two parameters in order to end up at the very **minimum of the function J** (that corresponds to the **center of the contour plot**). In order to minimize J → Gradient Descent.

4.1 Gradient Descent

Given a function $J(\Theta_0, \Theta_1)$ with parameters Θ_0, Θ_1 we want to minimize it.
How can we do it?

- Start with some Θ_0, Θ_1;
- Keep changing Θ_0, Θ_1 to reduce $J(\Theta_0, \Theta_1)$ until we hopefully end up at a minimum;



What are we actually doing?
Given an initial guess Θ_0, Θ_1, we have to keep changing those values until we get to a global minimum. Starting from a position, we will follow the slope of the cost function and make steps always in the direction of maximum decrease of the cost function. We may also reach a local minimum and this depends on the initial guess, in fact if we start from a point far from the global minimum we'll probably only be able to reach a local minimum.

The **Gradient Descent Algorithm** is an iterative algorithm that allows us to do so:

$$\text{repeat until convergence } \{$$
$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \quad (\text{for } j = 0 \text{ and } j = 1)$$
$$\}$$

Where α (the so-called **Learning Rate**) is the speed of how much we update Θ.
We need to repeat this step for all parameters Θ until convergence: take the current value of Θ_j and subtract from it α that multiplies the **derivative** in Θ_j of the cost function $J(\Theta_0, \Theta_1)$ (so the cost function computed at Θ_0, Θ_1).
Note: we have to compute this operation for both j=0 and j=1 **at the same time**! This is to avoid Θ_0 to update the result of Θ_1 which is computed later. Their values must be independent and computed simultaneously:

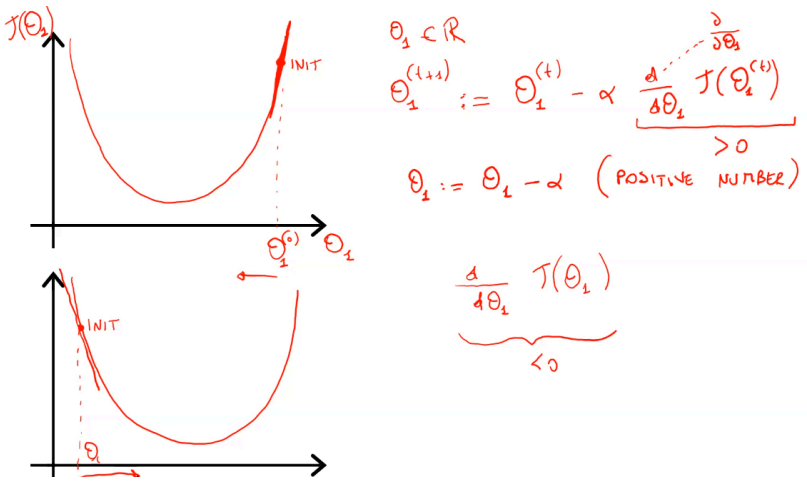
$$\begin{aligned} &\text{temp0} := \theta_0 - \alpha \tfrac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) \\ &\text{temp1} := \theta_1 - \alpha \tfrac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) \\ &\theta_0 := \text{temp0} \\ &\theta_1 := \text{temp1} \end{aligned}$$

OK!

$$\begin{aligned} &\text{temp0} := \theta_0 - \alpha \tfrac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) \\ &\theta_0 := \text{temp0} \\ &\text{temp1} := \theta_1 - \alpha \tfrac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) \\ &\theta_1 := \text{temp1} \end{aligned}$$

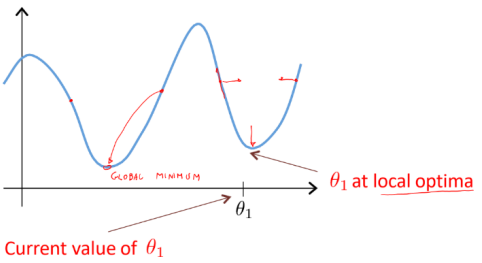
NOT OK!

EX. Let's consider a plot with $\Theta_1 \in \mathbb{R}$ and $J(\Theta_1)$. In this plot we can set up a line that represents the shape of the cost function (so the value the cost function assumes varying Θ_1). Let's also consider the starting point "INIT". How do we calculate $\Theta_1(t+1)$?
$\Theta_1(t+1) := \Theta_1(t) - \alpha * \partial / ((\partial * \Theta_1)) * J(\Theta_1(t))$
Notice that since we are considering one parameter then the partial derivative corresponds to the total derivative. The derivative corresponds to the slope of the line and it is a positive number. For that reason, we're subtracting a positive value from the "INIT" value so we're actually moving to the left (towards the minimum).



Note: if the "INIT" point is on the left side, then the derivative that represents the slope of the line will be negative. Subtracting this value we'll be moving to the right (always towards the minimum).

Now, notice from the gradient descent formula the value of the derivative **depends on the learning rate** (which controls the speed we move in the line). If **α is too small**, gradient descent **can be slow** while if **α is too large**, gradient descent can **overshoot the minimum**. It may fail to converge, or even diverge. Remember that gradient descent can converge to a local minimum, even with the learning rate α fixed. Starting from a point, getting closer and closer to the minimum, the value of the first derivative decreases until we get to the minimum (where the derivative is 0 - it is a straight horizontal line). As we approach a local minimum, gradient descent will automatically take smaller steps. So, there is no need to decrease α over time.



Once we have the algorithm for gradient descent, **let's apply it in the case of linear regression**.

$$h_\theta(x) = \theta_0 + \theta_1 x$$

PARAMETRIZED BY

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

The cost function is equal to the **mean square error** across all training examples. Let's compute it:

$$\frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) = \frac{\partial}{\partial \Theta_j} \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

$$= \frac{\partial}{\partial \Theta_j} \frac{1}{2m} \sum_{i=1}^{m} \left(\Theta_0 + \Theta_1 x^{(i)} - y^{(i)} \right)^2$$

Now we can substitute in the J. We take the derivative of this term with respect to theta-0. So we compute the first derivative paying attention to the fact that the 2 in the esponent comes out and simplifies with the 1/2 out of the parentheses leaving us with 1/m.

Coefficent of theta-zero

$$j = 0 : \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot 1$$

$$j = 1 : \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot x^{(i)}$$

Coefficent of theta-one

So, for the case of gradient descent for **linear regression we have this algorithm**:

$$\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$$

repeat until convergence {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)$$

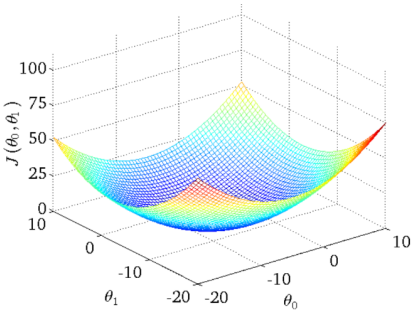
$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) \cdot x^{(i)}$$

}

update θ_0 and θ_1 simultaneously

$$\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$$

Now that we have this algorithm, we actually find the parameters in order to minimize the error. Note that in the case of linear regression, our cost function will be a **bowl-shaped convex function** meaning that if we compute the second derivative (if computable), this will be non-negative (which means that computing the algorithm we will reach the global minimum regardlessly to the starting point).



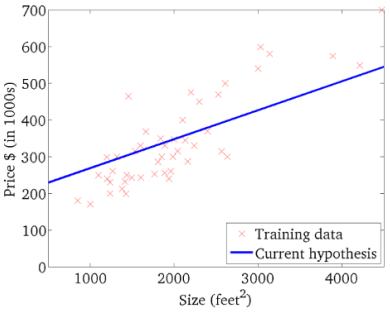
EX.

Starting for a generic point and executing the algorithm, we'll eventually reach the center of the contour plot. Notice that at the beginning, the direction doesn't point towards the center. The gradient of a function always points to the direction of maximum slope which will be orthogonal to the contour line. So the x firstly goes to the point of maximum slope before heading to the center.

In the example above, we're going to have Θ_0=-0.12 and Θ_1=100, so $h_\Theta(x)$=100+(0.12)x that corresponds to the line represented on the left: that will be our hypothesis which is also the best fit.

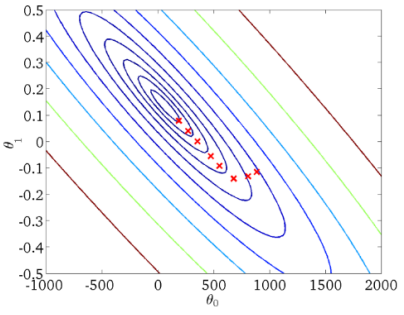
$h_\theta(x)$

(for fixed θ_0, θ_1, this is a function of x)



$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)



Note: the minimum of the cost function is unlikely to be 0 because it would mean that each point exactly fits the line but this almost never happens.

We define this kind of Gradient Descent with "**batch**" in fact **each step of gradient descent uses all the training examples**. So the gradient is the result of considering all dataset samples but this may be very slow.

Can we do something more efficient?
For large training sets, the evaluation of the gradient over all samples may be expensive so **Stochastic** or **"online" gradient descent** approximates the true gradient by a gradient at a single example. How does it work?

Pseudocode:

- Choose an initial vector of parameters $\boldsymbol{\theta_0}$ and learning rate α
- Repeat until convergence:
  - Randomly shuffle examples in the training set
  - For $k = 1,2, \dots m$, do:
    - $\boldsymbol{\theta}^{(i+1)} = \boldsymbol{\theta}^{(i)} - \alpha \nabla J^{(k)}(\boldsymbol{\theta}^{(i)})$

So, the batch gradient takes into account every training example so we compute the gradient of each point and we sum it up. In the stochastic one, we take all the points and we shuffle them. Then we consider one point (a random one between the k ones) and we calculate the gradient of that point, then the go to another point and so on. We'll eventually use every point. At the end, if we don't converge, we can reshuffle the set of points, and start again.
So instead of running a thousand iterations of gradient descent considering all of the samples every time, we'll just sample single points, getting its gradient and proceeding this way with the other points remaining. Clearly, if we consider single points then we don't know what will be the direction of the cost function. We'll zig-zagging in the contour plot until we'll eventually reach the same outcome of the batch gradient method.
This approach is preferable to the batch gradient one but there is another one which is considered to be better and it is called the **mini-batch gradient descent** which is a mixture between the two. What we do is:

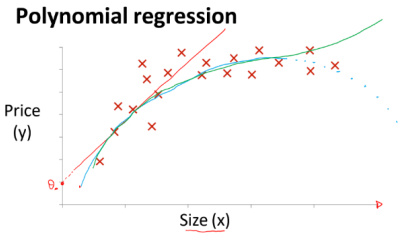
- Shuffle the example set;
- Select mini-batches (that is subsets of the shuffled set);
- We iterate between the mini-batches considering all values;
- If we don't converge then: repeat.

So, in the case of batch gradient descent, the gradient was always pointing towards the orthogonal direction to the contour plot but it was really slow. In the stochastic gradient descent the direction was very noisy. In mini-batch we have something between so it has a smoother convergence than stochastic GD and normally faster than batch GD (thanks to vectorization libraries).

Now, how can we be sure that gradient descent is working correctly?
We said that **we'll eventually converge choosing a proper learning rate α**. Given a generic point in the algorithm, we know that J(Θ) should decrease after each iteration. We can declare convergence if J(Θ) decreases by less than 10^{-3} in one iteration.
If the cost function diverges (or stays still) after each iteration then gradient descent is not working and this means that **the learning rate α is too high**: **to solve just use smaller α.**
For sufficiently small α, J(Θ) should decrease on every iteration but, if **α is too small**, **gradient descent can be slow to converge**.
How do we choose a proper learning rate α?
We start from a small α and, at every iteration, we increase its value (for example tripling it) until we notice that it is too big (and the function diverges).
EX. 0.001 (starting α) → 0.003 → 0.01 → 0.03 → 0.1 → … → 1.

4.2 Features and Polynomial Regression

Imagine we have to predict the cost of a house given its size, then we can choose size as our feature. Look at the plot on the right: if we only consider one feature x then our hypotheses $h_\Theta(x) = \Theta_0 + \Theta_1(x)$ will be represented as a line that actually doesn't fit data too much. We can possibly define another feature that is the squared size of the house: so the hypotheses $h_\Theta(x) = \Theta_0 + \Theta_1(x) + \Theta_2(x^2)$ that is a parabola and it fits better at the beginning but soon leads to a completely wrong hypotheses (see blue line). We can keep on defining other features like x^3 so that our hypotheses becomes $h_\Theta(x) = \Theta_0 + \Theta_1(x) + \Theta_2(x^2) + \Theta_2(x^3)$. The resulting line fits better but, since we add many features, we'll eventually have a lot of data to manage and this might be a problem.



$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3$$
$$= \theta_0 + \theta_1(size) + \theta_2(size)^2 + \theta_3(size)^3$$
$$x_1 = (size)$$
$$x_2 = (size)^2$$
$$x_3 = (size)^3$$

$$\theta_0 + \theta_1 x + \theta_2 x^2$$
$$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$$

Size: 1 - 1000
Size^2: 1 - 1,000,000
Size^3: 1 - 10^9
NEED TO SCALE THE FEATURES

Again, we can define the hypotheses and the cost function (as well as the gradient descent) for a generic number of features:

$$h_\theta(x) = \theta^T x = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

$$J(\theta_0, \theta_1, \ldots, \theta_n) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Repeat {

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

(simultaneously update θ_j for $j = 0, \ldots, n$)

}

Note: If we have too many features, there is the possibility to combine some: for example, if we have two features "frontage" and "depth" we can compute the "area" and use it as a feature instead of the previous two.

Note: Also, make sure that **features are on a similar scale**. For example, if we have two features x_1 = size of the house (0-2000 feet^2) and x_2 = number of bedrooms (1-5) and this can lead to some problems such as the fact that the gradient descent will last longer (it requires a higher number of steps to converge due to the fact that the features don't have the same scale). One possible solution is the normalization of the features so expressing the value of features as values between -1 and 1. Remember that $x_0 = 1$ as a convention in any case so we shouldn't normalize it.

EX. We could replace x_i with $x_i - \mu_i$ to make features have approximately zero mean. So we normalize this way:

- x_1 = (size (feet^2)-1000)/2000 (μ_1=1000 is the average size in feet);
- x_2 = (#bedrooms-2)/4) (μ_2 = 2 is the average number of bedrooms);
- We'll have that $-0.5 \le x_1 \le 0.5$, $-0.5 \le x_2 \le 0.5$

In general if we want to perform mean normalization, we have to replace each feature value (except x_0) with: $\mathbf{x_j = (x_j - \mu_j)/s_j}$ (where s_j is the range computed as max-min, and μ_j is the average value of x_j in the training set).

So **we need to scale the features**. Choosing the right features is something that can make the difference. Some **dangers** of **(Polynomial) Regression** are:

- **Overfitting**: The model is too complex and the residual error is low (or null) but this situation might bring strange deep curves in the cost function. Formally, it is "the production of an analysis that corresponds too closely or exactly to a particular set of data, and may therefore fail to fit to additional data or predict future observations reliably". An overfitted model is a mathematical model that contains more parameters than can be justified by the data. The essence of overfitting is to have unknowingly extracted some of the residual variation (i.e., the noise) as if that variation represented the underlying model structure. The possibility of over-fitting exists because the criterion used for selecting the model is not the same as the criterion used to judge the suitability of a model. For example, a model might be selected by maximizing its performance on some set of training data, and yet its suitability might be determined by its ability to perform well on unseen data; then overfitting occurs when a model begins to "memorize" training data rather than "learning" to generalize from a trend. As an extreme example, if the number of parameters is the same as or greater than the number of observations, then a model can perfectly predict the training data simply by memorizing the data in its entirety. Such a model, though, will typically fail severely when making predictions.

- **Underfitting**: The model is too simple and the residual error is too big. Formally, underfitting occurs when a mathematical model cannot adequately capture the underlying structure of the data. An under-fitted model is a model where some parameters or terms that would appear in a correctly specified model are missing. Under-fitting would occur, for example, when fitting a linear model to non-linear data. Such a model will tend to have poor predictive performance.

4.3 Normal Equation

Let's start defining some notation:
Nabla indicates the gradient (it corresponds to computing the derivative of f along all possible dimensions). In this case we have the gradient of a scalar-valued function of multiple variables. It is defined as a vector of partial derivatives of f according to each variable x_1, x_2, .., x_n. So, f is a function of R^n values that maps into a scalar R ($f:R^n \to R$).

$$\nabla f(p) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(p) \\ \vdots \\ \frac{\partial f}{\partial x_n}(p) \end{bmatrix}$$

NABLA

This allows to define gradient descent in a more formal way: we can define Θ^{t+1} as Θ^t (its previous value) minus the learning rate α multiplied by the gradient of the cost function in Θ. This formula fits the one above in fact $\nabla_\Theta J$ is n+1 dimensional and returns a scalar value that represents the residual error.

$$\Theta := \Theta - \alpha \nabla_\Theta J \qquad \nabla_\Theta J = \begin{bmatrix} \frac{\delta J}{\delta \Theta_0} \\ \vdots \\ \frac{\delta J}{\delta \Theta_n} \end{bmatrix} \in \mathbb{R}^{n+1}$$

Let's now build up some more notation: in general, given any function f that maps from a matrix A of size m*n to a scalar R. We define the **gradient of f(A) with respect to A** as the matrix of all partial derivatives with respect to the terms in A (so A_{11}, .., A_{1n}, .., A_{m1}, .. A_{mn}). The gradient of A with respect to A matches the dimensionality of A that is R^{mxn}.

- For $f: \mathbb{R}^{mxn} \mapsto \mathbb{R}$, define:

$$\nabla_A f(A) = \begin{bmatrix} \frac{\partial}{\partial A_{11}} f & \cdots & \frac{\partial}{\partial A_{1n}} f \\ \vdots & \ddots & \vdots \\ \frac{\partial}{\partial A_{m1}} f & \cdots & \frac{\partial}{\partial A_{mn}} f \end{bmatrix} \qquad f(A) \in \mathbb{R},\ A \in \mathbb{R}^{mxn}$$

GRADIENT OF $f(A)$ W.R.T. A

$\nabla_A f(A) \in \mathbb{R}^{m \times n}$

If A is a square matrix with size n*n then we define its **trace** as the summation of all the terms in the main diagonal array.

Trace:

- If $A \in \mathbb{R}^{nxn}$ $\quad \text{tr}\, A = \sum_{i=1}^{n} A_{ii} \in \mathbb{R}$

Some properties (without proof):

1. tr(AB)=tr(BA);
2. tr(ABC)=tr(CAB)=tr(BCA) (it permutes ciclicly);
3. If f(A)=tr(BA) then ∇_Atr(AB)=B^T (B^T is the transpose of B, ∇_Atr(AB) is the gradient of the trace of AB with respect to A);
4. tr(A)=tr(A^T) (the elements are the same and so the result);
5. If a∈R then tr(a)=a;
6. ∇_Atr(ABA^TC)=$CAB+C^TAB^T$;

What we want to do is rewrite the cost function in a matrix form.
We have m examples (x^1, y^1),..,(x^m, y^m) and n features. So, for each of our training examples we have a vector which is n+1 dimensional (every example has n+1 features, where x_0^i=1 for convention).
Our design matrix is formed by rows with the transposed vector of the m examples. The dimension of this matrix is m*(n+1).
We also have to define a matrix for the target variable y (the one we want to predict) which is just a vector of scalars.

$$x^{(i)} = \begin{bmatrix} x_0^{(i)} \\ x_1^{(i)} \\ x_2^{(i)} \\ \vdots \\ x_n^{(i)} \end{bmatrix} \in \mathbb{R}^{n+1} \qquad \times = \begin{bmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ \vdots \\ (x^{(m)})^T \end{bmatrix} \in \mathbb{R}^{m \times (n+1)}$$

DESIGN MATRIX

$$y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

Now we have all the tools to **rewrite the cost function** as a function of XΘ-y:

$$X\Theta - y = \begin{bmatrix} h(x^{(1)}) - y^{(1)} \\ \vdots \\ h(x^{(m)}) - y^{(m)} \end{bmatrix} \qquad \text{where} \qquad \Theta^T x^{(i)} = \sum_{j=0}^{n} \Theta_j x_j^{(i)}$$

DESIGN MATRIX

X is the design matrix. If we write XΘ-y then we'll end up to all of the differences between the estimate (the hypotheses computed for the specific training example) and the target value. So XΘ-y is actually a vector the of errors that we get. So the cost function can be represented as:

$$\frac{1}{2}(X\Theta - y)^T(X\Theta - y) = \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 = J(\Theta) \qquad \theta \in \mathbb{R}^{n+1} \qquad z^T z = \sum_i z_i^2$$

Note: we simply apply the property on the right and add a ½ in front of the equation. Notice that, in previous formulas, we put 1/m instead of ½ but it doesn't matter since we're interested in finding the minimum of the cost function despite the multiplicative term.
So, we redefined the cost function as the product of vectors and matrices.

Now, let's prove the normal equation by starting from an intuition: imagine that we are in the 1D case (to Θ∈R) and:

$$J(\theta) = a\theta^2 + b\theta + c$$

This equation corresponds to a bowl in the plot since the grade is 2 (so, a parabola with upwards concavity). We want to get the minimum of J in Θ and this happens when the first derivative of J in Θ is equal to 0. We want to **solve it for Θ** (analytically):

$$\frac{\partial}{\partial\Theta} J(\Theta) \overset{SET}{=} 0$$

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \quad \text{attained when} \quad \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = 0$$

Note: We have to find Θ* which is the argmin of J in Θ. What's *arg min*? If we have a function $f(x)=a+x^2$. The minimum of f(x) in x is a (again, the function has an upwards concavity so given "a" a point in the y-axes then "a" will represent its minimum). Now, argmin f(x) in x is 0 in fact it corresponds to the value that x takes when the function is at its minimum.
When minimizing J, what we care about is its argmin that represents the value of Θ that identify the best fit. So we get the argmin of J(Θ) when the gradient of J in respect to Θ is equal to 0.
Let's expand J(Θ):

Start from here:

$$\nabla_\Theta \frac{1}{2}(X\Theta - y)^T(X\Theta - y) \qquad \text{tr } a = a \qquad a^T = a$$

Note that XΘ-y is a scalar so we can apply property 5:

$$= \frac{1}{2}\nabla_\Theta \text{tr}\left(\Theta^T X^T X\Theta - \Theta^T X^T y - y^T X\Theta + y^T y\right)$$

SCALAR

Let's distribute the trace and the gradient operations and apply the nambla operator:

$$= \frac{1}{2}\left[\nabla_\Theta \text{tr}\,\Theta\Theta^T X^T X - \nabla_\Theta \text{tr}\, y^T X\Theta - \nabla_\Theta \text{tr}\, y^T X\Theta\right]$$

Start from defining the gradient of the cost function using the definition stated above. We know that the cost function computes a scalar so we can write the cost function as the trace of itself. Also the trace of the cost function is equal to the cost function itself. So, in the second step we take ½ out of the gradient and multiply it by the gradient of the trace (the cost function is a scalar so we substitute it with its trace) of the expanded quadratic function in the first step. In the third step we simply distribute the trace and the gradient operator (it is possible to do so since they are all linear operators),swapping the order of the elements in the first term, removing the fourth term since it doesn't depend on Θ and its derivative it's 0 and getting the transpose of the second term (it is a scalar so we transpose has the same result). Now, notice that the second and third terms are equal so let's work

on the first one. Notice that we add an I (identity) because we want to get to the recall property (the sixth property):

Recall $\nabla_A \operatorname{tr} ABA^TC = CAB + C^TAB^T$ $\nabla_A \operatorname{tr} AB = B^T$

$\nabla_\Theta \operatorname{tr} \Theta I \Theta^T X^T X = X^T X \Theta I + X^T X \Theta I$

For the second and third terms we obtain (applying the third property):

$\nabla_\Theta \operatorname{tr} y^T X \Theta = X^T y$

Putting all together (notice that we have 2 times $-X^Ty$, we sum $X^TX\Theta$ and simplify the two multiplicative 2 with ½):

$$\nabla_\theta J = \frac{1}{2}\left(X^TX\theta + X^TX\theta - 2X^T\bar{y}\right)$$
$$= X^TX\theta - X^T\bar{y} = 0$$

So, in order to find the minimum of the cost function we must set the gradient to be 0 (just move one term to the right). We end up with the **normal equation**:

$X^TX\theta = X^T\bar{y}$ Normal equation

Inverting (X^TX) we get to a closed form solution that is Θ* (it represents the argmin of J with respect to Θ):

$\theta^* = \left(X^TX\right)^{-1}X^T\bar{y}$

Note: What are the differences between Gradient Descent and Normal Equation?

Gradient Descent	**Normal Equation**
Need to choose α	No need to choose α
Needs many iterations	Don't need to iterate
Works well even when n is large	Need to compute $(X^TX)^{-1}$
	Slow if n is very large

What if (X^TX) is non-invertible?
It could be when:

- We have reductant features (linearly dependent)
  - E.g. x^1 = size in feet^2
    x^2 = size in m^2
    Note that it is easily solvable by removing one of the two features (like x^2).
- We have too many features (e.g. m≤n): in this case a solution might be removing features or using regularization.

EX. We have m=4 training examples and n+1=4 features (where x_0=1 for convention). We can design the X and y matrices where $X \in \mathbb{R}^{m*(n+1)}$ and $y \in \mathbb{R}^m$. We just have to compute the formula below in order to get the optimal value for Θ (which is the one that minimize the cost function).

x_0	Size (feet²) x_1	Number of bedrooms x_2	Number of floors x_3	Age of home (years) x_4	Price ($1000) y
1	2104	5	1	45	460
1	1416	3	2	40	232
1	1534	3	2	30	315
1	852	2	1	36	178

$$X = \begin{bmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \\ 1 & 1534 & 3 & 2 & 30 \\ 1 & 852 & 2 & 1 & 36 \end{bmatrix} \in \mathbb{R}^{m \times (n+1)} \qquad y = \begin{bmatrix} 460 \\ 232 \\ 315 \\ 178 \end{bmatrix} \in \mathbb{R}^m$$

$$\theta = (X^TX)^{-1}X^Ty$$

4.4 Linear Regression and Correlation

Let's make a summary of what we did so far: given a set of known (x,y) points we need to find a function f(x)=ax+b that "best fits" the known points (i.e., f(x) is close to y). We have to use this function to predict y values for new x's. This function can be also used to **test correlation**.
What's correlation?
It's when values track each other (e.g. height and shoe size, grades and entrance exam scores).
On the contrary, what's causation?
It's when one value directly influences another (e.g. education level → starting salary, temperature → cold drink sales).
Given the function f(x) defined before, the better the function fits the points, **the more correlated x and y are**.
So, there will be a residual error and it is meaningful to say how values are correlated.
Correlation applies to linear function only in fact it is a measure of how x's and y's are correlated.
Correlation can be both:

- **Negative**: When one variable goes up, the other goes down (e.g. the more the latitude, the low is the temperature);
- **Positive**: When one variable goes up, the other goes up too (e.g. car weight versus gas mileage);

We already saw how to fit a line with linear regression: given a point and a line, the error for the point is its vertical distance d from the line, and the squared error is d^2. Given a set of points and a line, the sum of squared error (SSE) is the sum of the squared errors for all the points (we previously called it the cost function). The goal is that given a set of points, we have to find the line that minimizes the SSE. We can do it with gradient descent, normal equation of software packaged (e.g. Numpy polyfit);

The correlation is **usually a value r that varies between 1** (when there is maximum positive correlation and points in the line are perfectly aligned so there is no error) **and -1** (when there is maximum negative correlation). If the correlation **is 0 then there is no correlation** between values. **Swapping x and y axes yields the same values** (so correlation doesn't change).This value is usually called **Pearson Coefficient** or **Coefficient Correlation**.
We can also talk about the **coefficient of determination**: there is a value r^2 (r squared) that varies between 0 and 1 and it can be used in the cases where the relations between variables is not linear so it measures fit of any line/curve to a set of points.

SUM. In linear regression we want to find a function that allows us to compute the dependent variable y from the independent variable x, so given some data we have to find a function to estimate the y's. In the correlation, x and y are random variables and we simply measure how much one relates to the other one and what's the common trend (e.g. if one increases, the other decreases).

4.5 Locally-Weighted Regression

First of all let's recall what happens in linear regression during the testing phase: remember that during the training phase our aim is to learn a function $h_\Theta(x^i)$ and measure the discrepancy between the estimate computed by the hypotheses and the ground truth that is given. In particular, we want to find the best parameters Θ to minimize that discrepancy for all the training examples. Our the estimate the Θ, we can use them in the testing phase to estimate new unseen examples x: we just have to make the inner product between the j-th Θ and the j-th Θ feature of the new example.

$$h_\Theta(x) = \sum_{j=0}^{n} \Theta_j x_j = \Theta^T x$$

In the previous sections, we said that choices of features is a really important task in order to avoid overfitting or underfitting. **Locally-weighted regression** (also known as **Loess** or **Lowess**) slightly differs from linear regression we've seen before: in linear regression we have a fixed set of parameters while in the locally-weighted regression (which is a **non-parametric learning algorithm**) parameters grow with the data.
How does it work?
Let's imagine we have a set of training examples. In the case of linear regression we want to fit θ to minimize in the training phase:

$$\frac{1}{2}\sum_{i=1}^{m}\left(y^{(i)} - \theta^T x^{(i)}\right)^2$$

and return $\theta^T x$ in the testing phase. So when we have a new value x, we have to evaluate h the certain x.

In the locally-weighted regression we want to fit θ to minimize:

$$\sum_{i=1}^{m} w^{(i)} \left(y^{(i)} - \theta^T x^{(i)} \right)^2$$

It is easy to notice that the difference with linear regression is the w^i: w^i is the weight that depends on the i-th training example, in fact:

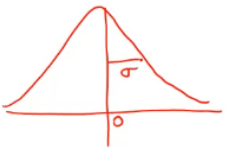
$$w^{(i)} = \exp\left(- \frac{\| x^{(i)} - x \|^2}{2\tau^2} \right)$$

Where τ is the bandwidth, x is a new test example and $x^{(i)}$ is a value given at training time.
If $||x^{(i)}-x||$ is small then $w^{(i)}=1$ while if $||x^{(i)}-x||$ is large then $w^{(i)}\approx 0$.
So, we have a set of training data and we want to get the value of the fit function at a position x. We then consider $w^{(i)}$ that returns a function then tells us how to weight all of the closest neighbors of x. So, imagine we want to obtain a function that fits a new test example x, what we do is fit locally a linear regression and weight locally all the training examples according to **how close they are to the new x** (close points are weighted more than further ones depending on the bandwidth). Everytime we need to add a new x, we fit locally the line that best fits the training date close to our x. If τ is large then we'll consider all of the training set (resulting in normal linear regression), while if τ is smaller then we only consider x's that are close to our new training example.
Note: This is a non parametric technique but this doesn't mean there might be no problem such as overfitting. In fact if τ is large and we have a lot of data it may occur overfitting. Contrary, if τ is small then there may be underfitting.
Note: We use the norm in the $w^{(i)}$ formula since x is normally a vector (except in the case it is only one feature) so we want it to be close to any of the points in the training set.

4.6 Probabilistic Interpretation of Least Squares

From a probabilistic point of view, what does it mean to minimize the cost function?
Let's assume $\mathbf{y^i=\Theta^T x^i+\epsilon^i}$ where $\epsilon^{(i)}$ is the **error** which is used to include all the **unmodelled effects** (such as the random noise). Let's assume that this noise ϵ^i can be modeled by a gaussian distribution which is zero mean and has got a fixed variance σ^2. So:

$$\varepsilon^{(i)} \sim N(0, \sigma^2)$$



What's the probability of ϵ^i?

$$P\left(\varepsilon^{(i)}\right) = \frac{1}{\sqrt{2\pi}\,\sigma} \exp\left(\frac{-(\varepsilon^{(i)})^2}{2\sigma^2} \right)$$

We make an addition assumption that ϵ^i is **independent and identically distributed (i.i.d.)** (so they are all independent and distributed according to a gaussian with zero mean and a σ^2 variance). This allows to get the distribution of y by substituting it from the equation above:

$$P\left(y^{(i)} \mid x^{(i)}; \theta \right) = \frac{1}{\sqrt{2\pi}\,\sigma} \exp - \frac{\left(y^{(i)} - \theta^T x^{(i)} \right)^2}{2\sigma^2}$$

So the probability of $y^{(i)}$ given the example $x^{(i)}$ and the parameters Θ is given by the equation above, y is a random variable which is centered around the value $\Theta^T x^i$ which is the mean of the gaussian. The whole probability is parametrized by Θ.
Making the assumption of the noise which is gaussian distributed actually allows us to write a probability distribution for y with Θ parameters, that is distributed like a gaussian where the mean is $\Theta^T x^i$ and the variance is σ^2:

$$y^{(i)} \mid x^{(i)}; \theta \sim N\left(\theta^T x^{(i)}, \sigma^2 \right)$$

What we want to do next is to consider the **likelihood of the parameters** which is equal to the probability of the training data we're given according to the probability distribution. Given a vector of target values and X the design matrix (remember it is formed by rows with the transposed vector of the m examples), the likelihood is computed by:

$$\mathcal{L}(\Theta) = P(\vec{y} \mid X; \Theta)$$

$$= \prod_{i=1}^{m} P\left(y^{(i)} \mid x^{(i)}; \Theta\right)$$

$$= \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\,\sigma} \exp\left(-\frac{\left(y^{(i)} - \Theta^T x^{(i)}\right)^2}{2\sigma^2}\right)$$

We want to maximize the likelihood of our Θ, and it is rather convenient to consider the **log likelihood**:

$$\ell(\Theta) = \log \mathcal{L}(\Theta)$$

$$= \log \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\,\sigma} \exp(\dots)$$

For the property of logarithm we can rewrite everything like:

$$= \sum_{i=1}^{m} \left[\log \frac{1}{\sqrt{2\pi}\,\sigma} + \log \exp(\dots)\right]$$

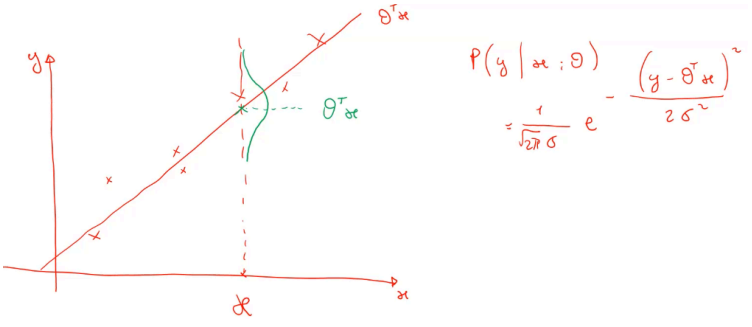
$$= m \log \frac{1}{\sqrt{2\pi}\,\sigma} + \sum_{i=1}^{m} -\frac{\left(y^{(i)} - \Theta^T x^{(i)}\right)^2}{2\sigma^2}$$

In the first row, we bring the product into the logarithm (it becomes a summation over all of the training examples and we also distribute it to the normalizing term and to the exponential). In the second row, since the first term is constant, we can simply multiply it by the number of training examples m. We also simplify the exp and log operations and end up with the second term.

What we actually want to do is maximize the likelihood so we're applying **MLE (Maximum Likelihood Estimation)** that is choosing Θ to maximize L(Θ).
The term on the left is a constant and doesn't depend on Θ so what really matters in the formula is the term inside brackets. Since there is a minus inside of it, then **maximize Θ corresponds to minimizing**:

$$\frac{1}{2} \sum_{i=1}^{m} \left(y^{(i)} - \Theta^T x^{(i)}\right)^2 = J(\Theta)$$

That is our cost function J(Θ).
This means that when we minimize the cost function, we also compute MLE so we maximize the likelihood between parameters assuming that gaussian noise is *i.i.d.*



$$P(y \mid x; \Theta) = \frac{1}{\sqrt{2\pi}\,\sigma} e^{-\frac{\left(y - \Theta^T x\right)^2}{2\sigma^2}}$$

So P(y|x;Θ) is the probability of the getting the y around the mean $\Theta^T x$ of the gaussian around the new test semple and the given parameters Θ.

5. Probability Theory Review

Let's make a recap of probability:

- **Bayes' Rule**: for any events A, B such that P(B) ≠ 0, we define the probability of A given B like:

  $$P(A \mid B) := \frac{P(A \cap B)}{P(B)}$$

  Let's apply conditional probability to obtain Bayes' Rule:

  $$P(B \mid A) = \frac{P(B \cap A)}{P(A)} = \frac{P(A \cap B)}{P(A)}$$
  $$= \boxed{\frac{P(B)P(A \mid B)}{P(A)}}$$

  EX. Given this data, let's compute Bayes' Rule:
  Given P(A)=0.05 where A="patients that are alcoholic" and P(B)=0.10 where B="patients that have liver disease". Now, we know that P(A|B)=0.07 that is patients with liver disease that are alcoholic. What's the percentage of alcoholic people with liver disease?
  P(B|A)= (0.1*0.07)/0.05=0.14 (Applying Bayes' Rule).

  There is also the **conditioned Bayes' Rule**: Given events A, B and C:

  $$P(A \mid B, C) = \frac{P(B \mid A, C)P(A \mid C)}{P(B \mid C)}$$

  Proof: see slides.
- **Chain Rule**: For any n events A_1, ..., A_n, the joint probability can be expressed as a product of conditionals:

  $$P(A_1 \cap A_2 \cap \ldots \cap A_n)$$
  $$= P(A_1)P(A_2 \mid A_1)P(A_3 \mid A_2 \cap A_1)\ldots P(A_n \mid A_{n-1} \cap A_{n-2} \cap \ldots \cap A_1)$$
- **Law of Total Probability**: Let B_1, …, B_n be n disjoint events whose union is the entire sample space then, for any event A:

  $$P(A) = \sum_{i=1}^{n} P(A \cap B_i) = \sum_{i=1}^{n} P(A \mid B_i)P(B_i)$$

  We can then write Bayes' Rule as:

  $$P(B_k \mid A) = \frac{P(B_k)P(A \mid B_k)}{P(A)} = \boxed{\frac{P(B_k)P(A \mid B_k)}{\sum_{i=1}^{n} P(A \mid B_i)P(B_i)}}$$

  EX. Treasure chest B1 holds 100 gold coins. Treasure chest B2 holds 60 gold and 40 silver coins. Choose a treasure chest uniformly at random and pick a coin from that chest uniformly at random. If the coin is gold, then what is the probability that you chose chest B1?
  $P(B_1)=P(B_2)=0.5$ (Choosing from B1 or B2 is a disjoint event)
  $P(G|B_1)=1$
  $P(G|B_2)=0.6$
  $P(B_1|G)=[P(B_1)P(G|B_1)]/[P(B_1)P(G|B_1)+P(B_2)P(G|B_2)] = (0.5*1)/(0.5*1+0.5*0.6)=0.625$
- **Independence**: Events A, B are independent if:

  $$P(AB) = P(A)P(B)$$

  We denote this as A ⊥ B. From this, we know that if A ⊥ B:

  $$P(A \mid B) = \frac{P(A \cap B)}{P(B)} = \frac{P(A)P(B)}{P(B)} = P(A)$$

  Note: If two events are independent, observing one event does not change the probability that the other event occurs.
  In general: events A_1, ..., A_n are mutually independent if

  $$P(\bigcap_{i \in S} A_i) = \prod_{i \in S} P(A_i)$$

  for any subset S ⊆ {1, ..., n}.

- **Random Variables**: A random variable (RV) X maps outcomes to real values. X takes on values in Val(X) ⊆ R. X = k is the event that random variable X takes on value k. With Discrete Random variables, Val(X) is a set and P(X = k) can be nonzero. With Continuous Random Variables, Val(X) is a range, P(X = k) = 0 for all k and P(a ≤ X ≤ b) can be nonzero.
- **Probability Mass Function (PMF)**: Given a discrete RV X, a PMF maps values of X to probabilities:

  $$p_X(x) := P(X = x)$$

  This expresses the probability that the random variable X takes value x. For a valid PMF:

  $$\sum_{x \in Val(x)} p_X(x) = 1$$

  If we take the summation for x that varies on the entire Val(X) of $p_X(x)$ the result is 1.
- **Cumulative Distribution Function (CDF)**: A CDF maps a continuous RV to a probability (i.e. R → [0,1]):

  $$F_X(x) := P(X \leq x)$$

  A CDF must fulfill the following:
  - $\lim_{x \to -\infty} F_X(x) = 0$
  - $\lim_{x \to \infty} F_X(x) = 1$
  - If a ≤ b, then $F_X(a) \leq F_X(b)$ (i.e. CDF must be nondecreasing)
  - Also note: $P(a \leq X \leq b) = F_X(b) - F_X(a)$.
- **Probability Density Function (PDF)**: PDF of a continuous RV is simply the derivative of the CDF.

  $$f_X(x) := \frac{dF_X(x)}{dx} \quad P(a \leq X \leq b) = F_X(b) - F_X(a) = \int_a^b f_X(x)dx$$

  A valid PDF must be such that:
  - For all real numbers x , $f_X(x) \geq 0$.
  - $\int_{-\infty}^{\infty} f_X(x)dx = 1$
- **Expectation**: Let g be an arbitrary real-valued function.
  If X is a discrete RV with PMF p_X:

  $$\mathbb{E}[g(X)] := \sum_{x \in Val(X)} g(x)p_X(x)$$

  If X is a continuous RV with PDF f_X:

  $$\mathbb{E}[g(X)] := \int_{-\infty}^{\infty} g(x)f_X(x)dx$$

  Intuitively, expectation is a weighted average of the values of g (x), weighted by the probability of x.
  For any constant a ∈ R and arbitrary real function f:
  - E[a] = a
  - E[af(X)] = aE[f(X)]

  Given n real-valued functions $f_1(X)$, ..., $f_n(X)$:

  $$\mathbb{E}[\sum_{i=1}^{n} f_i(X)] = \sum_{i=1}^{n} \mathbb{E}[f_i(X)]$$

- **Variance**: The variance of a RV X measures how concentrated the distribution of X is around its mean. Var(X) is the expected deviation of X from E[X]. For any constant a ∈ R, real-valued function f(X):
  - Var [a] = 0
  - $Var[af(X)] = a^2Var[f(X)]$
- **Joint and Marginal Distributions (and Multiple RVs)**:
  - Joint PMF for discrete RV's X, Y (for discrete RV's X_1, ...,X_n):

    $$p_{XY}(x, y) = P(X = x, Y = y) \qquad p(x_1, ..., x_n) = P(X_1 = x_1, ..., X_n = x_n)$$

    $$\Sigma_{x \in Val(X)} \Sigma_{y \in Val(Y)} p_{XY}(x, y) = 1 \qquad \Sigma_{x_1} \Sigma_{x_2} ... \Sigma_{x_n} p(x_1, ..., x_n) = 1$$

Distribution	PDF or PMF	Mean	Variance
$Bernoulli(p)$	$\begin{cases} p, & \text{if } x = 1 \\ 1-p, & \text{if } x = 0. \end{cases}$	p	$p(1-p)$
$Binomial(n, p)$	$\binom{n}{k} p^k (1-p)^{n-k}$ for $k = 0, 1, ..., n$	np	$np(1-p)$
$Geometric(p)$	$p(1-p)^{k-1}$ for $k = 1, 2, ...$	$\frac{1}{p}$	$\frac{1-p}{p^2}$
$Poisson(\lambda)$	$\frac{e^{-\lambda}\lambda^k}{k!}$ for $k = 0, 1, ...$	λ	λ
$Uniform(a, b)$	$\frac{1}{b-a}$ for all $x \in (a, b)$	$\frac{a+b}{2}$	$\frac{(b-a)^2}{12}$
$Gaussian(\mu, \sigma^2)$	$\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$ for all $x \in (-\infty, \infty)$	μ	σ^2
$Exponential(\lambda)$	$\lambda e^{-\lambda x}$ for all $x \geq 0, \lambda \geq 0$	$\frac{1}{\lambda}$	$\frac{1}{\lambda^2}$

- Marginal PMF of X , given joint PMF of X, Y (of X_1, given joint PMF of X_1, ..., X_n):

$$p_X(x) = \sum_y p_{XY}(x,y) \quad p_{X_1}(x_1) = \sum_{x_2} \dots \sum_{x_n} p(x_1, \dots, x_n)$$

- Joint PDF for continuous X, Y (for continuous RV's X_1, ...,X_n):

$$f_{XY}(x,y) = \frac{\delta^2 F_{XY}(x,y)}{\delta x \delta y} \quad f(x_1, \dots, x_n) = \frac{\delta^n F(x_1, \dots x_n)}{\delta x_1 \delta x_2 .. \delta x_n}$$

- Marginal PDF of X, given joint PDF of X, Y (of X_1, given joint PDF of X_1,..., X_n):

$$f_X(x) = \int_{-\infty}^{\infty} f_{XY}(x,y)dy \quad f_{X_1}(x_1) = \int_{x_2} \dots \int_{x_n} f(x_1, \dots, x_n)dx_2 \dots dx_n$$

- **Expectation for multiple random variables**: Given two RV's X, Y and a function g: $R^2 \rightarrow R$ of X, Y:
  - For discrete X, Y:

$$\mathbb{E}[g(X,Y)] := \sum_{x \in Val(x)} \sum_{y \in Val(y)} g(x,y) p_{XY}(x,y)$$

  - For continuous X, Y:

$$\mathbb{E}[g(X,Y)] := \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} g(x,y) f_{XY}(x,y) dxdy$$

    These definitions can be extended to multiple random variables in the same way as in the previous slide. For example, for n continuous RV's X_1, .., X_n and function g : $R_n \rightarrow R$:

$$\mathbb{E}[g(X)] = \int \int \dots \int g(x_1, \dots, x_n) f_{X_1, \dots, X_n}(x_1, \dots, x_n) dx_1, \dots, dx_n$$

- **Conditional distributions for RVs**: Works the same way with RV 's as with events:
  - For discrete X, Y:

$$p_{Y|X}(y|x) = \frac{p_{XY}(x,y)}{p_X(x)}$$

  - For continuous X, Y:

$$f_{Y|X}(y|x) = \frac{f_{XY}(x,y)}{f_X(x)}$$

  - In general, for continuous X_1, ...,X_n:

$$f_{X_1|X_2,\dots,X_n}(x_1|x_2, \dots, x_n) = \frac{f_{X_1,X_2,\dots,X_n}(x_1, x_2, \dots, x_n)}{f_{X_2,\dots,X_n}(x_2, \dots, x_n)}$$

- **Bayes' Rule for RVs**: Also works the same way for RV 's as with events:
  - For discrete X, Y:

$$p_{Y|X}(y|x) = \frac{p_{X|Y}(x|y)p_Y(y)}{\sum_{y' \in Val(Y)} p_{X|Y}(x|y')p_Y(y')}$$

  - For continuous X, Y:

$$f_{Y|X}(y|x) = \frac{f_{X|Y}(x|y)f_Y(y)}{\int_{-\infty}^{\infty} f_{X|Y}(x|y')f_Y(y')dy'}$$

- **Chain Rule for RVs**: Also works the same way as with events:

$$f(x_1, x_2, \dots, x_n) = f(x_1)f(x_2|x_1)\dots f(x_n|x_1, x_2, \dots, x_{n-1})$$
$$= f(x_1)\prod_{i=2}^{n} f(x_i|x_1, \dots, x_{i-1})$$

- **Independence for RVs**: For $X \perp Y$ to hold, it must be that $F_{XY}(x, y) = F_X(x)F_Y(y)$ **FOR ALL VALUES** of x, y. Since $f_{Y|X}(y|x) = f_Y(y)$ if $X \perp Y$, chain rule for mutually independent X_1, ..., X_n is:

$$f(x_1, \dots, x_n) = f(x_1)f(x_2)\dots f(x_n) = \prod_{i=1}^{n} f(x_i)$$

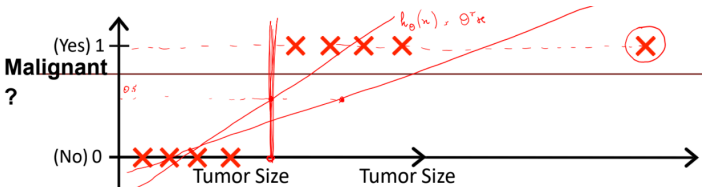
  Note: Very important assumption for a Naive Bayes classifier!

6. Classification with Logistic Regression, Newton's Method for Optimization, Generalized Linear Models

6.1 Logistic Regression

Earlier on we saw Linear Regression that is effectively a regression problem while Logistic Regression is a **classification problem** that is about **predicting discrete-valued output**. A classification problem requires that examples be classified into one of two or more classes (usually discrete). A problem with two classes is often called a two-class or **binary classification** problem (Logistic Regression) while a problem with more than two classes is often called a **multi-class classification** problem. Again we have a certain number of features x and we use them to predict a value y that is one of a set of classes (only two in the case of Logistic Regression).

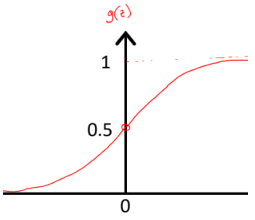
EX. In this example we classify the possibility for a tumor to be malignant. The only feature we have is the tumor size. The threshold classifier is set to 0.5 so, if the prediction $h_\Theta(x) \geq 0.5$ then we predict y=1 (the tumor is malignant), otherwise y=0.



Notice that in linear regression $h_\Theta(x)$ could be larger than 1 and smaller than 0 while in logistic regression it is a value between 0 and 1 so: $0 \leq h_\Theta(x) \leq 1$.

So, concerning the **Classification with Logistic Regression**, we want out probability given by $h_\Theta(x)$ to be between 0 and 1. We want the probability beyond a certain value so saturate with 1, and the probability above a certain value to saturate with 1. One way to do so is by using the Sigmoid (or Logistic) Function that has the following behavior:

$$g(z) = \frac{1}{1+e^{-z}}$$



Notice that it corresponds to a trend similar to the curve on the right. So instead of considering $\Theta^T x$, we're going to consider $h_\Theta(x)=g(\Theta^T x)$ in order to saturate the hypotheses either to 0 or to 1.
So, using the sigmoid function the hypotheses can be written as:

$$h_\theta(x) = \frac{1}{1+e^{-\Theta^T x}}$$

$h_\Theta(x)$ is the estimated probability that y = 1 on input x.

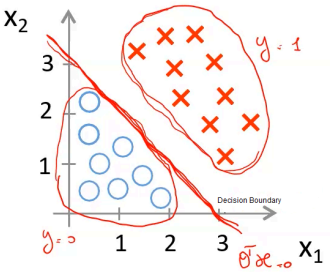
$$h_\theta(x) = P(y=1 \mid x;\theta)$$

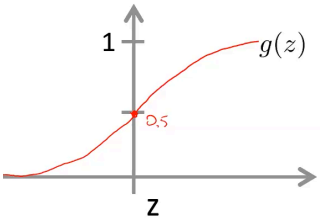
Also notice that:

$$P(y=0|x;\theta) + P(y=1|x;\theta) = 1$$
$$P(y=0|x;\theta) = 1 - P(y=1|x;\theta) \longrightarrow P(y=0 \mid x;\theta) = 1 - h_\theta(x)$$

EX. If $x = [x_0, x_1] = [1, \text{tumorSize}]$, $h_\Theta(x) = g(\Theta^T x) = 0.7$ tell patient that 70% chance of tumor being malignant.

<u>How do we define the decision boundary?</u>
We suppose predict y=1 if $h_\Theta(x)=g(\Theta^T x) \geq 0.5$ and this happens when $z \geq 0$ while we predict y=0 if $h_\Theta(x)=g(\Theta^T x)<0.5$ and this happens when z<0. So in this case the decision boundary is equal to 0 that's when the hypotheses $h_\Theta(x)$ is equal to 0.5.

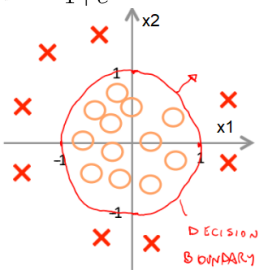




Now, let's assume we have this hypotheses:
$h_\Theta(x) = g(\Theta_0+\Theta_1 x_1+\Theta_2 x_2)$

Assuming that $\Theta_0=3$, $\Theta_1=1$, $\Theta_2=1$ (we'll see how to choose the parameters), then $\Theta^T x=-3+x_1+x_2$. So, we predict y=1 if $z=\Theta^T x=-3+x_1+x_2\geq 0$ and we predict y=0 if $-3+x_1+x_2<0$. This means out decision boundary (that corresponds to a line) is given by $\Theta^T x=-3+x_1+x_2=0$.

$$h_\theta(x) = g(\theta^T x)$$
$$g(z) = \frac{1}{1+e^{-z}}$$



Notice that we may also have **non-linear decision boundaries**. Assume we have $h_\Theta(x)=g(\Theta_0+\Theta_1x_1+\Theta_2x_2+\Theta_3x^2_1+\Theta_4x^2_2)$ with parameters $\Theta_0=-1$, $\Theta_1=0$, $\Theta_2=0$, $\Theta_3=1$, $\Theta_4=1$. As before, we we predict y=1 if $z=\Theta^T x=-1+x^2_1+x^2_2\geq 0$ (the same idea apply to y=1, just as before). So the decision boundary is given by $\Theta^T x=-1+x^2_1+x^2_2=0$.
Some more complex boundaries can be made just by adding more polynomial features.

<u>How do we learn Θ's?</u>
What we already saw for the case of Linear Regression, also applies to Logistic Regression. So, assume we have a training set where each training example has this shape (x^i, y^i). As for the features, we assume that we have n features $x=[x_0, x_1, .., x_n]\in R^{n+1}$ where $x_0=1$ by convention. We want to choose a set of parameters $\Theta=[\Theta_0, \Theta_1, .., \Theta_n]\in R^{n+1}$ that minimizes the error.

In order to do this, we must recall the probabilistic interpretation of the hypothesis $P(y=1|x;\Theta)$.
We saw that the probability of predicting y=1 and y=0 is given by:

$$P(y=1 \mid x;\theta) = h_\theta(x) \qquad P(y=0 \mid x;\theta) = 1-h_\theta(x)$$

Starting from these two line, we can write the probability of y as:

$$P(y \mid x;\Theta) = h_\Theta(x)^y \left(1-h_\Theta(x)\right)^{1-y}$$

Notice that if y=1 then the second term becomes 0 and we remain with the first one (the inverse for y=0).
We can now write the **Likelihood** of the parameters which is also the probability of our training data:

$$L(\Theta) = P(\vec{y} \mid X;\Theta) = \prod_{i=1}^{m} P\left(y^{(i)} \mid x^{(i)};\Theta\right) = \prod_{i=1}^{m} h_\Theta(x^{(i)})^{y^{(i)}} \left(1-h_\Theta(x^{(i)})\right)^{1-y^{(i)}}$$

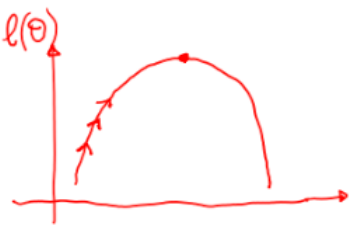
The probability of the target vector (with all the possible label for the training data), given the design matrix (with all the features) and the parameters Θ is equal to the product of i that goes to m (all training data) of the probability of each target value given the features of the sample i parametrized by Θ that is, again, equal to the general function we saw before applied to each training example.
It is convenient to consider the log likelihood to have a sum (instead the product) and to take the exponential to the front. So, the **Log Likelihood** is:

$$\ell(\Theta) = \sum_{i=1}^{m} y^{(i)} \log h_\Theta(x^{(i)}) + \left(1-y^{(i)}\right) \log\left(1-h_\Theta(x^{(i)})\right)$$

Note: in Linear Regression we could switch from the LogLikelihood to the cost function. We had a minus in front of the formula, so maximizing the likelihood corresponded to minimizing the cost function. This means that when computing the gradient we went to the negative gradient direction.

In Logistic Regression we assume that we use the **Gradient Ascent**, to get to the maximum of the Log Likelihood. The formula is the same as in Linear Regression with the difference that here we go to the **positive gradient direction**:

$$\Theta_j := \Theta_j + \alpha \frac{\partial}{\partial \Theta_j} \ell(\Theta)$$



Where the derivative of the LogLikelihood in Θ_J is:

$$\sum_{i=1}^{m} \left(y^{(i)} - h_\Theta(x^{(i)})\right) x_j^{(i)}$$

The final formula is:

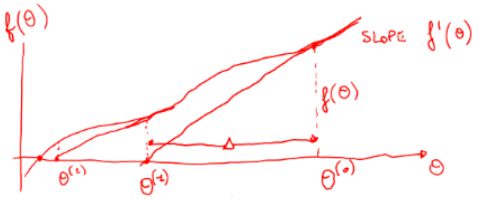
$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)}$$

Where $h_\Theta(x)=g(\Theta^T x)$ (in Linear Regression it was simply "$\Theta^T x$"!).

6.2 Newton's Method for Optimization

There are many optimization techniques we can use (just think of gradient descent) and, of course, each technique has its advantages and disadvantages. Now we see how **Newton's Method** works.
Let's assume we have a function f, we have to find Θ such as f(Θ)=0. Remember that in logistic regression we wants to find the likelihood of the parameters Θ in which we want the gradient to be equal to 0. So, Newton's Method is actually equivalent to finding the 0 of the first derivative of the log likelihood. So **our goal is to maximize l(Θ) so that l'(Θ)=0** (the first derivative is equal to 0).
So, assume we are in a certain starting parameter value Θ_0 and we take the slope of the curve at the relative position and if we follow the slope we'll find the next position of Θ (which we call Θ_1). Here we compute the new value of f (as well as its derivative) and use the tangent to derive Θ_2. The first line has equation y=ax where a is the first derivative f' of f at Θ_0. So we can write it as:

$$f'(\theta^{(0)}) = \frac{f(\theta^{(0)})}{\Delta} \qquad \Delta = \frac{f(\theta^{(0)})}{f'(\theta^{(0)})}$$



And we know notice that:

$$\theta^{(1)} = \theta^{(0)} - \Delta$$

So, what we want to do is updating Θ by following the first derivative of f. So we can define recursively the new position of Θ as the last position of Θ minus a certain Δ which is ration between f and f' computed at the last position of Θ. The general updating function is the following:

$$\theta^{(t+1)} := \theta^{(t)} - \frac{f(\theta^{(t)})}{f'(\theta^{(t)})}$$

Now, f(Θ) is no more that the first derivative of the log likelihood l'(Θ) so we can rewrite the Newton's Method algorithm in terms of the log likelihood by substituting into the delta, so:

$$f(\theta) = l'(\theta)$$

$$\theta^{(t+1)} := \theta^{(t)} - \frac{l'(\theta^{(t)})}{l''(\theta^{(t)})}$$

← Second derivative computed at $\Theta^{(t)}$

Now we need to redefine this formulation in the case **Θ is a vector**:
The only difference is that the second derivative becomes the Hessian to the power of -1 times the gradient in Θ of the log likelihood (that is a vector n+1 dimensional). The Hessian is defined as the matrix of second derivatives with respect to Θ_i and Θ_j (the matrix has dimension $R^{(n+1)*(n+1)}$).

$$\theta^{(t+1)} := \theta^{(t)} - H^{-1} \underbrace{\nabla_\theta l}_{\text{VECTOR } \mathbb{R}^{n+1}} \qquad \text{HESSIAN } \underbrace{H}_{\text{MATRIX } \mathbb{R}^{(n+1)\times(n+1)}}: \quad H_{ij} = \frac{\partial^2 l}{\partial\theta_i \partial\theta_j}$$

Newton's Method is usually faster than Gradient Descent, in fact it benefits from **quadratic convergence** (if we are in an area around the optimum, at each iteration we'll get squares of the error, e.g. 0.1 → 0.01 → 0.0001). The problem is the inverse of the hessian that has complexity $O(n^3)$ so it is not smart to use it if n (the number of features) is large.

6.3 Exponential Family of Distributions

This topic is important because it helps to turn out there is a general formulation for the optimization techniques, in particular in the next chapter (Generalized Linear Models) we'll get tools to solve for any distribution of the exponential family.

First of all, let's remember that when optimizing the parameters we were talking about the probability of y given x parametrized by Θ. The target value y can be both a real number distributed according to a Gaussian Distribution (in the case of Linear Regression) and we solved it with Least Squares, and a discrete variable (that takes value from 0, 1) assumed to have a Bernulli distribution (in the case of Logistic Regression).

$P(y \mid x; \Theta)$

$y \in \mathbb{R}$: GAUSSIAN → LEAST SQUARES

$y \in \{0, 1\}$: BERNOULLI → LOGISTIC REGRESSION

Now, we talked about the Bernoulli distribution parameterized by Φ, where Φ is the probability of y of taking label 1:

$\text{Bernoulli}(\phi) \quad P(y=1; \phi) = \phi$

And the normal distribution with mean μ variance σ^2:

$\mathcal{N}(\mu, \sigma^2)$

Both of them belong to the exponential family of distributions, so let's write a general formulation of it:

$$p(y; \eta) = b(y) \exp\left(\eta^T T(y) - a(\eta)\right)$$

Where η is the natural parameter, T(y) is the sufficient statistic that contains all the information that a single sample can contribution towards all the unknown parameters Θ (in most cases T(y)=y), b(y) is the base measure and a(η) is the log-partition function (it is a normalizing vector that usually allows to integrate to 1). Now, it is useful to notice that y is a scalar as well as b(y). In our case, also η and T(y) are scalars (despite in the general case they are usually vectors). If we consider them to be in a product relation then η and T(y) match.

Now, let's use this stuff to rewrite the two distributions by defining the values of a, b and T.

Let's start with the **Bernoulli distribution**:

$$P(y, \eta) = b(y) \exp(\eta^T T(y) - a(\eta))$$

We already saw that we can represent the probability of y like this:

$\text{Ber}(\phi) \quad P(y=1; \phi) = \phi \quad P(y; \phi) = \phi^y (1-\phi)^{1-y}$

We want to rewrite the P(y, Φ) to match the general form of the exponential family of distributions.

$$P(y; \phi) = \phi^y (1-\phi)^{1-y}$$

$$1 = \exp\left(\log \phi^y (1-\phi)^{1-y}\right)$$

$$2 = \exp\left(y \log \phi + (1-y) \log(1-\phi)\right)$$

$$3 = \exp\left(\left(\log \frac{\phi}{1-\phi}\right) y + \log(1-\phi)\right)$$

$\eta \quad T(y) \quad -a(\eta)$

So, first of all let's apply the exponential (1°) and bring the exponents to the front of the expression (notice that we split the multiplication as a summation of logs) (2°). Then we multiply (1-y) for log(1-Φ) getting log(1-Φ) -y(log(1-Φ)) and apply the property of the logarithm (in the case of a subtraction) for the two logarithms with the same argument getting (3°). Now, y corresponds to T(y), the first logarithm is the natural parameter η, the second logarithm is -a(η) (we add a minus to stick with the general formula) and b(y)=1 (since we have no other elements in front of the exponential).

If we now compute Φ as a function of η then we get the sigmoid function σ(η):

$$\eta = \log \frac{\phi}{1-\phi} \implies \phi = \frac{1}{1+e^{-\eta}} = \sigma(\eta)$$

$T(y) = y$

$b(y) = 1$

Also a(η) can be written as a function of η so:

$a(\eta) = -\log(1-\phi) = \log(1+e^{\eta})$

Let's see it again for the case of the **Gaussian Distribution**:

$$P(y,\eta) = b(y)\exp(\eta^T T(y) - a(\eta))$$ $\mathcal{N}(\mu, \sigma^2)$ ← Normal Distribution case

Let's put the variance σ^2 in order to simplify the derivation. Now, let's consider the equation for the normal distribution:

$$\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}(y-\mu)^2\right)$$

We can separate the part depending on y and the one depending on μ.

$$= \underbrace{\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}y^2\right)}_{b(y)} \exp\left(\mu y - \overbrace{\frac{1}{2}\mu^2}^{a(\eta)}\right)$$

$\eta = \mu$ $T(y) = y$

After this separation, everything in the front part of the equation is b(y) and, given T(y)=y and η=μ then we also have a(η). Note that we can substitute η in a(η):

$$a(\eta) = \frac{1}{2}\mu^2 = \frac{1}{2}\eta^2$$

Some common properties of the exponential family of distribution are:

- MLE (Maximum Likelihood Estimate) with respect to η is concave, so the negative log likelihood (NLL) is convex;
- The expected value y parametrized by η the natural parameter η is equal to the first derivative in η of the log partition function a(η).
- The variance of y parametrized by η is equal to the second derivative in η of the log partition function a(η).

$$E[y;\eta] = \frac{\partial}{\partial \eta} a(\eta)$$

$$\text{VAR}[y;\eta] = \frac{\partial^2}{\partial \eta^2} a(\eta)$$

6.4 Generalized Linear Models

Generalized Linear Models (GLM) give us the recipe to deal with inference and training of any of the distributions that come from the exponential family of distributions. First of all let's write some assumptions we'll use to define the generalized linear models:

1. We assume the distribution of y given x parametrized by Θ is from the exponential family of distributions.
2. Given x, the goal of the GLM is to output the expected value of the sufficient statistic given x. Remember that in most of the cases T(y)=y. What we actually want is that the hypothesis is the expected value.
3. We want to have a linear relation between η and $\Theta^T x$. In a more general case, so η is a vector of dimensionality k, we'll have to consider each η_i.

1) $y \mid x;\Theta \sim \text{ExpFamily}(\eta)$

2) GIVEN x, GOAL IS TO OUPUT $E[T(y)\mid x]$
WANT $h_\Theta(x) = E[T(y)\mid x]$

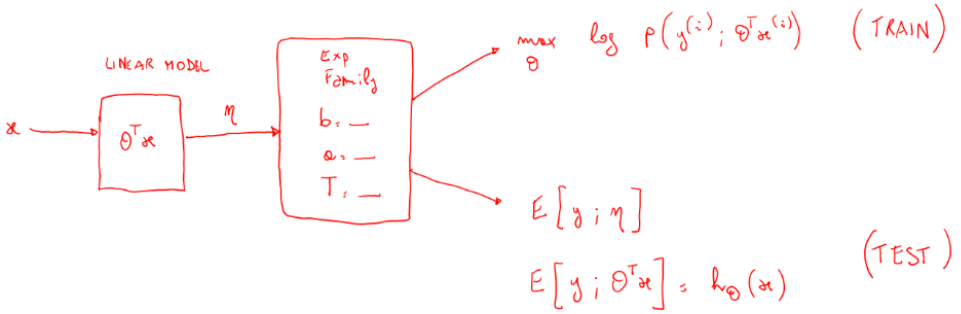
3) $\eta = \Theta^T x$
$\left[\eta_i = \Theta_i^T x \quad \text{if } \eta \in \mathbb{R}^k\right]$

$h_\Theta(x) = p(y=1 \mid x;\Theta)$
$T(y) = y$
$E[y\mid x;\Theta] =$
$0 \cdot p(y=0\mid x;\Theta) + 1 \cdot p(y=1\mid x;\Theta)$
$\left(E[x] = \sum x p(x)\right)$
$= p(y=1 \mid x;\Theta)$

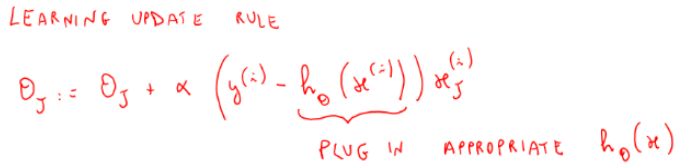
Let's now check that the interpretation we have of the hypotheses in the case of Logistic Regression, holds in the case of assumption 2. We already know that the hypotheses can we written as the probability of having y=1 given x parametrized by Θ. We also assume that T(y)=y. The expected value of y given x parametrized by Θ is given by the summation

of 0 times the probability of y=0 given x and 1 times the probability of y=1 given x, parametrized by Θ (so the summation of x*p(x)) that corresponds to the hypotheses.

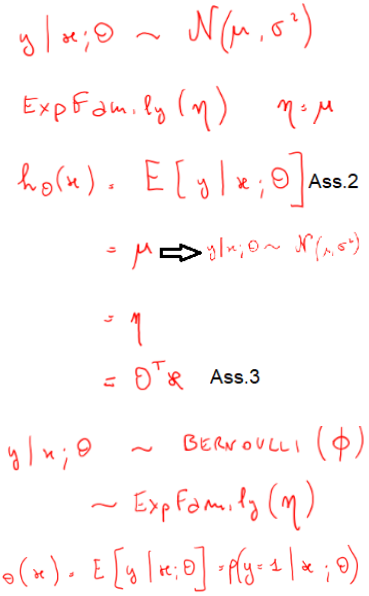
Let's now see a general formulation: we have the data x and we want to have a linear relation between the data and the natural parameter η. We need to assume that η comes from the exponential family of distributions, and for each distribution we have a function for a, b and T (as seen before). During training we want to maximize in Θ the log likelihood, so the log of the probability of $y^{(i)}$ parametrized by our η which is $\Theta^T x^{(i)}$. During testing we want to know for a specific sample what is the expected value of y given η (so $\Theta^T x$) that is what we have computed as the hypotheses.



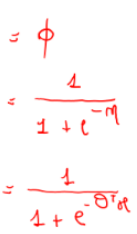
We have one unified learning update rule (e.g. gradient descent), we just have to plug in the appropriate hypotheses depending on the application (e.g. $\Theta^T x$ for linear regression or $g(\Theta^T x)$ for logistic regression):

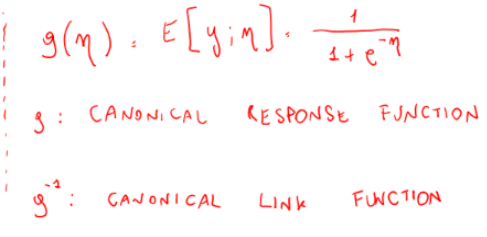


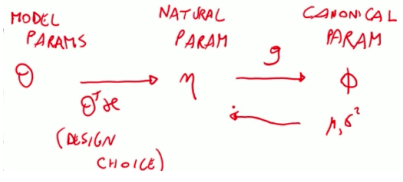
Now, we have to write down the generalized linear model for the Ordinary Least Squares and for the Logistic Regression.
Let's start with the **Ordinary Least Squares**:
We have a "y" (the response variable) that is continuous. Again, we model y given x parametrized by Θ as a gaussian with mean μ and variance σ^2. The Gaussian belongs to the exponential family and in the Gaussian distribution the natural parameter η=μ.
The hypothesis is the expected value of y given x parametrized by Θ (note that this is assumption 2). The expected value is μ (from the fact that y is distributed like a gaussian) and this is directly equal to η (we said before that η=μ). By assumption 3 we know that $\eta=\Theta^T x$.



Now, let's see it for the case of the **Logistic Regression**:
We have a "y" which is discrete (so it takes value 0, 1). We model y given x parametrized by Θ as a Bernoulli distribution. The Bernoulli distribution belongs to the exponential family and the natural parameter η=Φ. For fixed x and Θ, the algorithm outputs the hypotheses which is the expected value of y given x parametrized by Θ, that is again the probability of having y=1 given x parametrized by Θ (in the case of a Bernoulli distribution) and this is equal to Φ. We saw that η is equal to the sigmoid function in which we can plug in the relation between the natural parameter and the features.







The g function is what relates the natural parameter η and the Φ, in fact g(η) is basically the expected value of y given η that is the sigmoid function. "g" is called a canonical response function and its inverse g^{-1} is called canonical link function.

SUM. We have seen 3 parametrizations: Θ is the model parameters and from Θ we have seen the natural parameter η to be equal to $\Theta^T x$ (this is design choice). If we apply the canonical response function g, then we get the canonical parameter (Φ for Bernoulli, (μ, σ^2) for Gaussian, λ for Poisson). In the case of Gaussian there is a linear relation between the x's and the y. In the GLM we allow for modeling cases. There isn't such a linear relation between the features and the output variable but, rather, there is a linear function between the link function and the features (so between Θ and η).

6.5 Multi-Class Classification

In **Multi-Class Classification** we can have different outcomes. For example we may have a weather prediction model in which the possible outcomes are 4 (sunny, cloudy, rain, snow). Let's try to formalize it:
The response vector y takes k different values and for each i-th response variable we have a parameter Φ that tells us what it is the probability that y takes label i (**Generalized Bernoulli Distribution**). So, we can write this probability as:

$$p(y = i) = \phi_i$$

Actually, we don't need all the k parameters in fact noticing that the probability sums to 1 then Φ_K is:

$$\phi_k = 1 - (\phi_1 + \dots + \phi_{k-1})$$

← So we need k-1 parameters (not k).

Now, let's try to write the **Multinomial Distribution as one of the Exponential Family**: first of all let's notice that y∈{1, 2, .., k}. We have T(y) here is not equal to y but is a k-1 dimension vector of this form:

$$T(1) = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad T(2) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \dots, \quad T(k-1) = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \in \mathbb{R}^{k-1} \qquad T(k) = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

So, for T[i] we have that the i-th element is equal to 1 and the others to 0. This representation is called "**One hot vector**" and represents the point in which we have 1 where the label actually is. Let's also define T(K) that's the vector with all 0's. We can write a general formula of T(y) at position i that is basically the indicator function y=1.

$$T(y)_i = 1\{y = i\}$$

An example might be: $T(1)_1 = 1\{1=1\} = 1$ (since in the first position of the first vector the value is 1), while $T(1)_2 = 1\{1=2\} = 0$ (since y≠i so in the second position we have a 0).
Our goal is to compute the expected value of $T(y)_i$ that is the probability of y=i represented as Φ_i:

$$E[T(y)_i] = T(1)_i\, p(y=1) + \dots + T(i)_i\, p(y=i) + \dots + T(k)_i\, p(y=k) = p(y=i) = \phi_i$$

So, all terms are 0 except for the one in which $T(i)_i$ and we remain only with p(y=i) that is the probability we want.
Now, let's use the indicator function to write down the general equation for P(y):

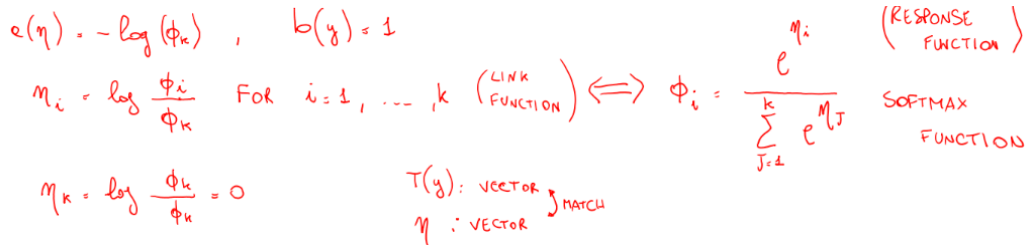
$$\begin{aligned} p(y) &= \phi_1^{1\{y=1\}} \cdot \phi_2^{1\{y=2\}} \cdots \phi_k^{1\{y=k\}} \\ &= \phi_1^{T(y)_1} \cdot \phi_2^{T(y)_2} \cdots \phi_k^{1 - \sum_{j=1}^{k-1} T(y)_j} \\ &= b(y) \exp\left(\eta^T T(y) - a(\eta)\right) \end{aligned}$$

$$= \dots = \exp\left[T(y)_1 \log\frac{\phi_1}{\phi_k} + \dots + T(y)_{k-1} \log\frac{\phi_{k-1}}{\phi_k} + \log\phi_k\right]$$

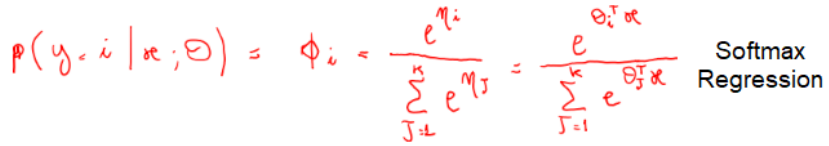
$\exp(\log(\cdot))$

P(y) can be written as the Φ_i when y=i and we can rewrite all of that using $T(y)_i$. Notice that for the case of Φ_K we represent it as a combination of all the others so 1 minus the summation from j=1 to k-1 of $T(y)_j$.We rewrite again all of this, as the exponential of a log (like in the Bernoulli distribution) and, substituting the terms (a(η), b(y) and η) in the expression, we can write it as the exponential family of distribution. η_i is our link function and we can solve Φ_i by finding the **softmax function** that is e to the ηi divided by the sum of j=1 to k of e to the η_j (so this is

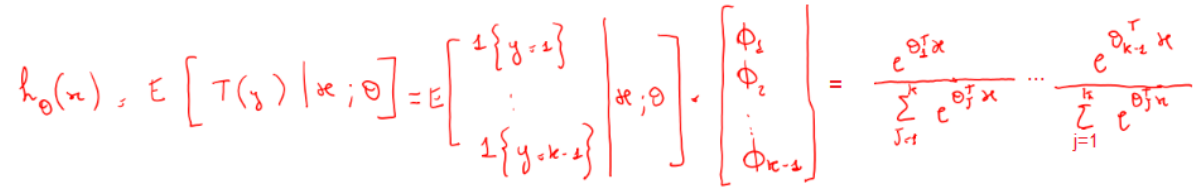
the response function). We can also set up another natural parameter η_K that is equal to 0. Notice that T(y) and η are vectors and both vectors match.



Now, we can **apply the generalized linear model and get the response function**. For assumption 3 we have that $\eta_i=\Theta^T x$ (this apply for i=1, .., k-1). So, we a vector of parameters $\Theta=[\Theta_1, .., \Theta_{k-1}]$ with dimensionality n+1 (one for feature plus the bias). We also define $\Theta_k=0$ and this matches $\eta_k=\Theta_k^T x=0$ (defined before). Now, we want to write the probability of having y=i given x parametrized by Θ as Φ_i that is basically the **softmax function** and switch to the Θ (because η and Θ corresponds by the linear relation):

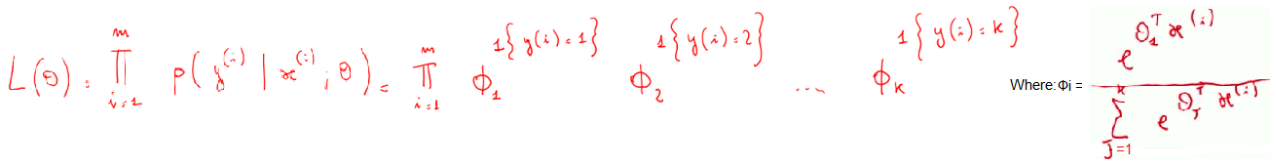


From assumption 2 we have that the hypothesis corresponds to the expected value of having T(y) given x parametrized by Θ and T(y) can be written as set of indicator functions from y=1 up to y=k-1 given x parametrized by $\Theta=\Theta_1, .., \Theta_{k-1}$. As written right above this can be again represented as the softmax function:



So, given a training example we can get the hypothesis that is the probability that it belongs to one of the k classes. We can plug it into the Gradient Descent algorithm for the case of the Softmax Regression.

An equivalent way of learn the θ's for Softmax Regression could be using the likelihood so given $y\in\{1, .., k\}$ and the set of training data $(x^{(1)}, y^{(1)}), .., (x^{(m)}, y^{(m)})$ and we write the likelihood as:



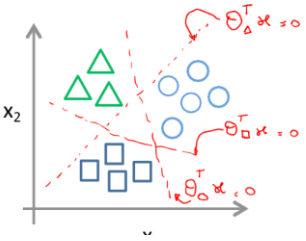
We take the log likelihood of it and try to maximize it.

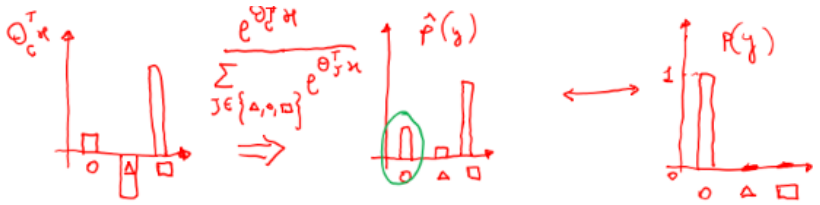
It is possible to interpret Softmax Regression as **One-vs-All** that means learning c classifiers where the hypothesis for each class is given by:

$$h_\Theta^{(c)}(x) = P(y = c|x; \Theta) \qquad (c = 1,2,3)$$

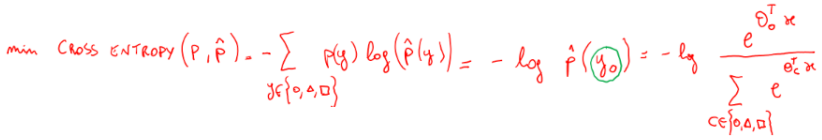
So it equivalent to train a logistic regression classifier $h_\Theta^i(x)$ for each class i to predict the probability that y=i. When making a prediction on a new input x, we pick the class i that maximizes the hypothesis $h_\Theta^i(x)$.

Now we see the relationship between Softmax Regression and Cross Entropy Minimization: assume we have a feature vector $x^{(i)}\in R^{n+1}$ and $y^{(i)}\in\{0, 1\}^K$ (so $y^{(i)}$ is a one-hot vector). The class parameter $\Theta_{CLASS}\in R^{n+1}$ where CLASS are the classes $C\in\{\Delta, \square, o\}$. $\Theta_\Delta^T x=0$ corresponds to a decision boundary between the class Δ and all the rest (see the example with all the geometrical forms on the right). So $\Theta_C^T x$ will have a value for each of the class. We can compute the probability of each class by applying the softmax regression function. Next, we compare it with the ground-truth that is a one hot vector.





Doing MLE is equivalent to **minimizing the cross entropy** between the ground truth and the estimation and it can be computed as minus the summation of y given the possible classes of p(y) multiplied by log(p-hat(y)).
It is possible to simplify this formula by simply considering -log(p-hat(y_o)) (notice that for △,□ p(y) is 0 so we only remain with log(p-hat(y_o) thanks to the multiplication) and this is the **cost**. This corresponds to the softmax function that computes the cost for o.



7. Image Classification with Linear Classifiers

Image classification is a core task in computer vision. Let's assume to have a given set of discrete labels (e.g. dog, cat, truck, etc..), image classification is about assigning a label or class to an entire image assuming to have only one class for each image. An image is represented by a grid of RGB numbers, so assuming to have a n*m image the dimension of the grid is n*m*3 (where 3 is the RGB channels). Image classification is not an easy task in fact we have some challenges:

- **Viewpoint variation**: All pixels in the image change when the camera moves.
- **Background Clutter**: It means that the image contains a large number of things, making it difficult for the observer to locate the desired object (the images have a lot of noise).
- **Illumination**: The system should be able to cope with variations in illumination as well. So, if we offer our image classification system a picture of the same item with varying brightness levels (Illumination), the system should be able to assign them the same label.
- **Deformation**: The object could appear in a different shape or position with respect to the usual ones.
- **Occlusion**: The object may be partially occluded by another object making it difficult to identify.
- **Intraclass variation**: Variations within the same class (e.g. relatives, breeds, etc..).

So we have to come up with an **image classifier** that allows us to identify a particular object in the image. There are several approaches.

7.1 Machine Learning: Data-Driven Approach

A first approach could be building a classifier using a **data driven approach**, so: we **collect a dataset of images and labels** (we have labels so we already know what's the object in the image), we use **machine learning to train a classifier** and then, we can **evaluate the classifier on new images**.
One example of classifier is the **Nearest Neighbor**: we algorithm is pretty simple because during the training phase it simply memorizes the training data and during the testing phase it compute the distance between the test image and all train images to find the closest one so that it can **predict the label of the nearest image**.
The **distance matrix** used is usually the **L1 distance** in which we compute the absolute difference between test image and the training image and then, the summations of the resulting pixels is made (so the output is a scalar). Nearest Neighbor algorithm is not good enough in fact it has an O(1) complexity for training and an O(n) complexity for prediction (where n is the number of example images) but we want **classifiers that are fast at prediction** (**slow for training is ok**).
Another algorithm is the **K-Nearest Neighbors (KNN)** in which instead of copying labels from nearest neighbors, take the majority **vote from K closest points**. The algorithm is basically the same as the previous one. Some distance metrics are the L1 (Manhattan) distance and the **L2 (Euclidean) distance**.

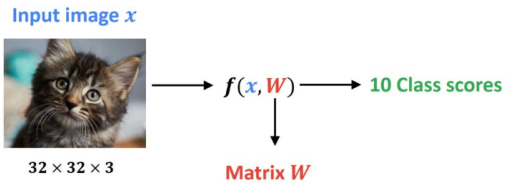
L1 Distance:
$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

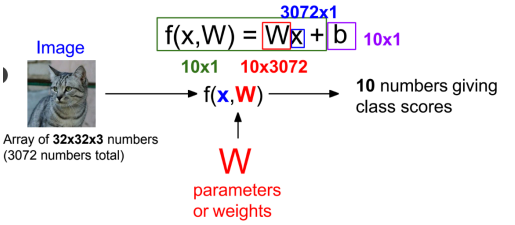
L2 Distance
$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$

Distance metric and K are **hyperparameters** that are chosen using the validation set.

7.2 Linear Classification: Parametric Approach

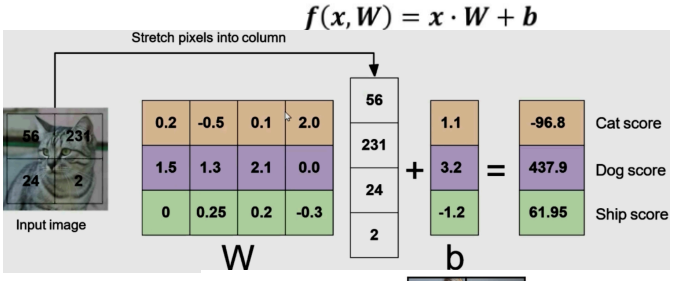
Let's assume we take a certain image x with dimension 32x32x3 from the CIFAR-10 dataset. We have a function f that takes an image x and a set of weights W as input. The image x is a **vectorized image** so we simply take each pixel in the image and put it in a 1 dimension vector (so the vector size will be 3071x1). W is a set of c **weights** where c are the number of possible classes (so assuming to have 10 classes the vector size will be 3072x10). The relation between the classifier and the output score is linear so we get **Wx** (in our case the vector will be 10x1) in which we end up with the **class scores**. The **class with the higher score** is the one the image belongs to.



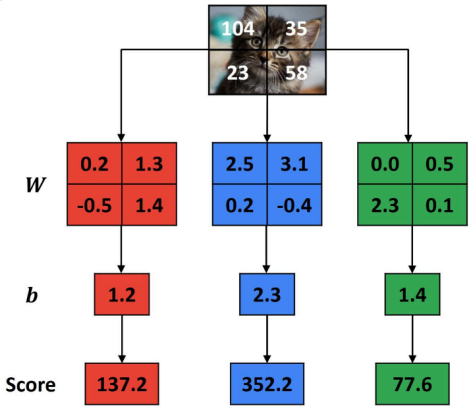


Notice that we also have a **bias vector** b that will be used to **adjust the hyperplanes** that this multiplication produces. You can think of the b as an intercept term. For example, if x is of dimension 2, we will have a plane into space. We can then shift this plane, up and down to adjust it and have optimal classification. However, in this case, b will only act as a parameter that will assist these hyperplanes. The vector b has dimension 1xc where c is again the number of classes (in our case c=10).

Notice in the example on the left that the final classification is wrong as this is due to the given set of weights and the bias vector. This is due to the fact that at the beginning we start with a **random initialization** (so we set random values for W and b) that gives us more or less the same scores for all classes. So we have to **adjust W and b** so that we may do a proper classification.



So, from a parametric point of view, we learned that score is the result of the inner product between a vectorized image x and one of the rows of a matrix W. We know that for inner products, there is a property: if there is a high correlation between two vectors (or vectorized images), the inner product will be large. On the other hand, if this correlation is low the inner product will be small. We can regard one of these vectors as a Template Vector. We can have one template vector for each class. Then, if the similarity between the Template Vector and some other vector is high, that vector belongs to that class. On the other hand, if the similarity is low, we probably will not get the vector that belongs to that class. Now, rather than stretching an image into a column vector, we can also reshape the rows of the matrix W into the same shape as the image. By doing that for each class, we will get the following tiny images of 2x2. pixels. However, these tiny images can now have a better intuition because these are very similar to Template vectors. So, we multiply x and W element by element and add the bias b, if the score is relatively high that would mean that this image shape is present in our image. This is the **Algebraic Viewpoint**.

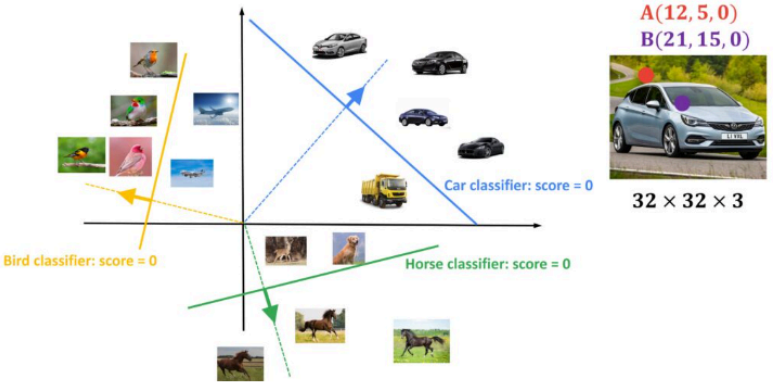


For our first exercise, we will train some Linear Classifiers and plot learned weights from the matrix W. It would be interesting to visualize what we can get as the output. So, for each class, we will get a template image that should represent a certain property. For instance, if we work with the CIFAR10 dataset, W will be the template matrix for a certain class from this dataset like a car, airplane, or ship. Here's an illustration of interpreting all Linear Classifiers in **Visual Viewpoint**.

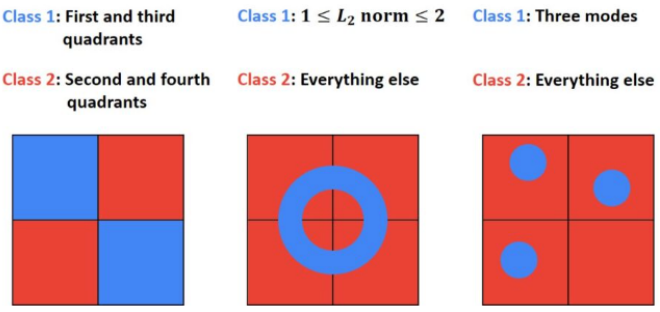
Another interesting interpretation of a Linear Classifier is called a **Geometric Viewpoint**. Let's say that we have one image and we want to calculate the score for three classes: car, horse, and bird. To illustrate this point of view, let us just pick one pixel from an image. That pixel will have some coordinates of x=12, y=5, channel=0. Then we will draw a plot where the x axis is the value of the pixel and the y axis is the value of the classifier when the pixel changes while we keep all other pixels in the image fixed. We will change the values of this individual

pixel from 0 to 255 (for color images in the CIFAR10 dataset we are changing the values of just one RGB channel) and multiply each value with three different weights template matrix W. Once we do this, we will get certain lines that represent how these scores are changing in a linear manner.
Another experiment that we can perform is to incorporate multiple pixels simultaneously. So, now instead of one pixel, we can pick two pixels. Then, we will again change the values of the first pixel and second from 0 to 255. Let us call them x_1 and x_2 pixels. We will plot x_1 and x_2 and we will calculate the score for each of these three classes (car, horse, and bird). Note that, as we change the values of two pixels 256 times, we will get 256x256 different score values. So, for each of these combinations, we will get a grid. We will go to a one-pixel increment and we will get an image of 256x256 pixels. We can think of this image as some score image for each of these classes.

EX. Now let's illustrate this. In the following image, we can see a plot where along the x axis we have values of the first pixel, and along the y axis we have values of the second pixel. Along the third axis z, we will have the values of the classifier. It is difficult to draw this third axis in 2D so try to imagine that you are looking at this graph from a bird's perspective. Now, to represent how scores are changing in 3D we will get planes instead of lines. These planes will intersect and form certain lines in this pixel space. These lines show all points in the space where a score will be equal to 0. Now, because this is a linear example, there is a direction in this pixel space along which the score will increase linearly. This direction is orthogonal to this line. Then, the learned car template will be somewhere along this line that is orthogonal to the score. In the example below, we can see three different lines for three different classes.



To visualize the correlation between these three planes let's have a look at the **3D interpretation** of the previous example. Now, these three planes will intersect one horizontal plane at z=0 and form three lines. Along these lines, we will have all points in the space where a score is equal to 0. Then, we can see that everything below these lines will have a negative score value and everything above these lines will have a positive score value. The highest score values will be located in the highlighted areas of these planes. Also, notice three red lines where planes intersect with each other. These lines are called decision boundary lines and they separate our classes into three regions in space.



Another interesting thing is that we see some **hard cases** for a Linear Classifier. The idea is that two-dimensional pixel space is colored, with red and blue corresponding to different categories that we want the classifier to recognize. These are examples of three cases that are completely impossible for the linear classifier to recognize. On the left, we can see the case where the first and the third quadrants have one category, and the second and fourth quadrants have the other category. If you think about it, there's no way that we can draw a single hyperplane that can divide the red and the blue categories. Another case that is very interesting is the case on the right where we can see three modes. Here, in the blue category, there are maybe three distinct regions in pixel space, that correspond to possibly different visual appearances of the category we wish we want to recognize. Again if we have these different regions in pixel space, corresponding to a single category, there's no way for a single line to perfectly carve up the red and the blue regions.

So far we have defined a (linear) score function f(x,W) = Wx + b but we still need to find a way to **quantify how good W is**. So, we need to define a **loss function** that quantifies our unhappiness with the scores across the training data. Given a dataset of n examples (x_i, y_i), the loss over the dataset is a average of loss over examples:

$$L = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i)$$

We want to do MLE (Maximum Likelihood Estimation) meaning we want to maximize the probability of our data so it is convenient to interpret raw classifier scores as **probabilities**. So given s=f(x_i; W) the computed score of the i-th example, we can compute the probability of each class using the **Softmax Function**:

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

The loss for the i-th training example is computed as:

$$L_i = -\log P(Y = y_i|X = x_i)$$

Maximizing the probability of the training data is equivalent to comparing with **Cross Entropy** the probabilities for each class we get with the one-hot encoded vector of the ground truth.

Cross Entropy

$$H(P, Q) = H(p) + D_{KL}(P||Q)$$

EX.

$s = f(x_i; W)$

Probabilities must be >= 0

$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$ Softmax Function

Probabilities must sum to 1

$L_i = -\log P(Y = y_i|X = x_i)$

cat	**3.2**	exp	**24.5**	normalize	**0.13**	compare **1.00**
car	5.1		164.0		0.87	0.00
frog	-1.7		0.18		0.00	0.00
	Unnormalized log-probabilities / logits		unnormalized probabilities		probabilities	Correct probs

Cross Entropy $H(P, Q) = H(p) + D_{KL}(P||Q)$

So, in out case the target output and the output of the network are:

$$P^t(y) = \begin{cases} 1 & y = y_i \\ 0 & y \neq y_i \end{cases} \qquad Q(y|x_i) = P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

The **Cross Entropy Loss** for the image x_i is computed as:

$$L_i = L(x_i) = -\sum_y P^t(y) \log Q(y|x_i) = -1 \cdot \log Q(y_i|x_i) = -\log P(Y = y_i|X = x_i) = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

Note: The minimum loss we get is 0 while it has no upper bounds (it can eventually reach max infinity).

Note: A **Regularization** can be applied to prevent the model from doing too well on training data. This is basically done to reduce the chance of overfitting and:

- Express preferences over weights;
- Make the model simple so it works on test data;
- Improve optimization by adding curvature;

$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W)$$

Full Loss:

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + R(W)$$

Notice that λ is a hyperparameter. Some example of regularization functions are:

L2 regularization: $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2): $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

There are also more complex formulas (e.g. Dropout, Batch normalization, Stochastic depth, etc..).

How do we find the best set of weights W?

Just as in the other algorithms, we have to make some optimization to reduce the loss. In multiple dimensions, the **gradient** is the vector of (partial derivatives) along each dimension. The slope in any direction is the **dot product** of the direction with the gradient. The direction of steepest descent is the **negative gradient**. So far we have followed an analytical approach but it's not always easy, especially if considering more layers. In **Neural Networks** we usually use some other ways to compute the gradient where **Backpropagation** is the most efficient one.

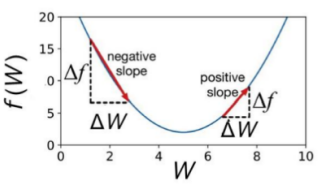
$$\frac{f(W)}{dW} = \lim_{h \to 0} \frac{f(W + h) - f(W)}{h}$$

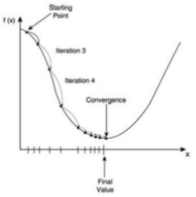
current W:	**W + h** (second dim)**:**	**gradient dW:**
[0.34,	[0.34,	[-2.5,
-1.11,	-1.11 + **0.0001**,	**0.6**,
0.78,	0.78,	?,
0.12,	0.12,	?,
0.55,	0.55,	
2.81,	2.81,	
loss 1.25347	**loss 1.25353**	(1.25353 - 1.25347)/0.0001 = 0.6

First of all let's see another **numerical approach** to compute the gradient that simply makes use of the definition of first derivative. So, given an increment h we compute the value of the function at f(W+h) and subtract it for f(W), then we finally divide it by h. Now, notice that W is an n dimensional vector so as the h so, for example, if we want to increment the 2nd dimension, we must have something like H = [0, h, 0, 0, ..]. This method is **slow** because there's the need to iterate over all dimensions. Also, it is an **approximation** because we are computing the lim for h→0 but we are bounded by the numerical approximation we can have. This is useful in order to check the implementation when using the **analytical gradient** (this is called **gradient check**). The problem with this approach is that it is **error-prone** meaning that if we are in a large NN with a lot of layers, if we change some parameters then everything we have done need to be thrown away and rewrite the gradient again: that's why we need a better approach to compute the gradient.
Just to make a summary, the gradient is computed this way:

$$W_{t+1} = W_t + a_t d_t \qquad d_t = -\nabla f(W_t)$$

We move in the opposite direction of the gradient. In case we have more dimensions, remember that the gradient always points to the direction which is orthogonal to the contour line before ending up to the minimum.





Again we could optimize Gradient Descent using stochastic GD or mini batch GD. To summarize:

- Assume Loss to be:

  $$L(W) = \frac{1}{n}\sum_i^n L_i(W)$$

  With n the number of training samples and L_i the loss for training sample x_i.
- **Stochastic Gradient Descent**: Randomly choose one training sample x_i and update weights based on loss $L_i(W)$. Remember that in this case we have no guarantee that the line will be orthogonal to the direction of the contour plot but we'll eventually end up to the same outcome as batch gradient GD.

  $$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W)$$

  $$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

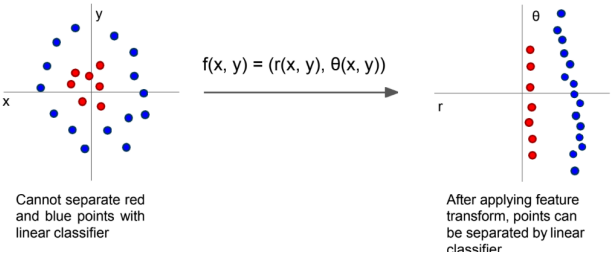
- **Mini-batch training**: process a subset of training samples M ⊂ {1, ... , n} and update weights based on:

  $$L_M(W) = \frac{1}{|M|}\sum_{i\in M} L_i(W)$$

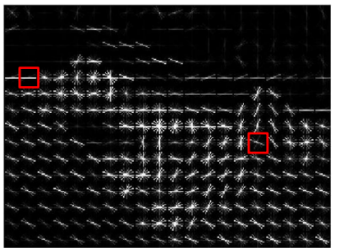
  This approach is faster than batch GD and the convergence is smoother than the stochastic method.
- **Batch training**: Process all training samples and update weights based on:

  $$L(W) = \frac{1}{n}\sum_i^n L_i(W)$$

7.3 Image Features



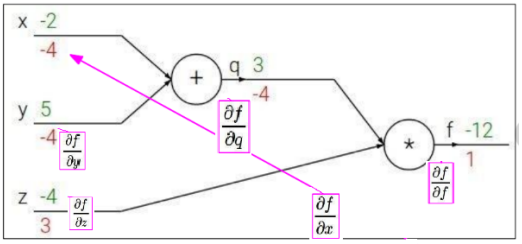
So far we talked about features as the elements that allow us to make a prediction. In the case of an image, the features are the **raw pixels** so each (R, G, B) for each pixel in the image. Thus, in order to increase the performance, instead of using pixels we can pre-compute some features such as the **histograms or colors**. This is useful because we may have cases in which pixels are not linearly separable so we couldn't use a linear classifier (remember that linear classifiers try to set a hyperplane that separates the two classes). If we concatenate the features to the distance from the origin, they might become separable.
A **more recent approach** consists in considering **Histogram of Oriented Gradients (HoG)** in which we consider the first derivative of all patches across a number of fixed directions. The whiter the resulting bars, the larger the derivative value. So, for each path we compute the derivatives and build an histogram in which each bin represents an orientation. The color of the bar is proportional to the size of the histogram. This is better than a normal histogram of colors.



Another approach is extracting random patches from the image and clustering them to form a codebook of “visual words”. We could represent the content of an image by considering how many instances of those codebooks were contained in the image. This is another sophistication we won’t see in detail.
Notice that we can concatenate all the methods described above. Now, this approach of engineering features has come to a dead end, so we could improve performance more than this but at the end we'll never get to the level of performance the machine can actually devise for itself if it was free to choose the features. So, instead of extracting features by hand, we’ll see a method that uses Neural Networks to figure out the features and also the specific machine learning method to use.

7.4 Neural Networks and Backpropagation

As said before, analytic gradient (as well as the numerical gradient) is not good for the Neural Network so many reasons (e.g. too many matrix calculus, if we change the loss we need to re-derive from scratch and it is also not feasible for very complex models) so we have to devise a better way to compute the gradient which is **Backpropagation**.



Let’s start by looking at an example. Let’s assume we have a function f(x, y, z) = (x + y)z. We have that x=-2, y=5, z=-4, then if we compute the function with these values, the result is the loss. We need to be able to compute the gradient of f with respect to all parameters in order to change their values so that the value of f changes.So, we want:
We notice that:

$$\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$$

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

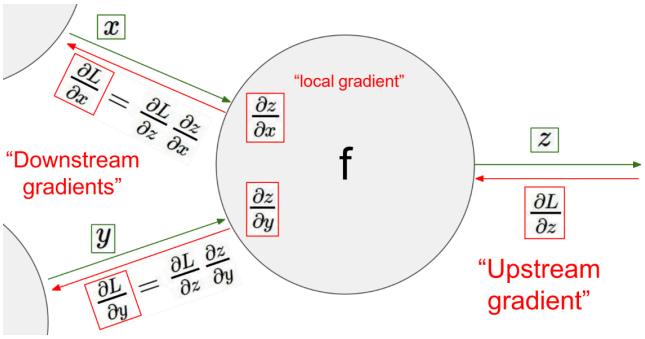
Chain rule:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y}$$

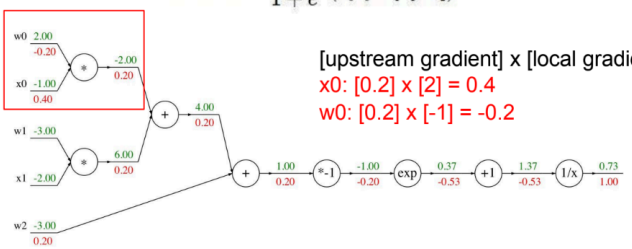
Upstream gradient — Local gradient

So, we start from the derivative of f, and we backpropagate until we compute all derivatives and, finally, we compute the new loss. When computing the derivative of f with respect to y, we can use the general case which consists in using the **chain rule** by making the multiplication between the **upstream gradient** and the **local gradient**. So in this case we will have the product of what comes from upstream (-4) and the local gradient (1), the result is in fact -4. The same is applied to the derivative of f with respect to x.

Let’s try to give a general formulation: we have a certain node of the shape on the right. At that node the **local gradients** are the derivatives of z with respect to x and with respect to y. Then we have some **upstream gradient** that comes all the way up from deep neural networks and this is the derivative of the loss L with respect to z (that’s all we need to know for backpropagation). For what concerns the derivative of L with respect to x, it will simply be computed as the product of the local gradient and the upstream gradient (the same for y). So, the derivative of L with respect to x, will become the upstream gradient for the next layer and so on and so forth.



Another example: $f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$



[upstream gradient] x [local gradient]
x0: [0.2] x [2] = 0.4
w0: [0.2] x [-1] = -0.2

EX. Let’s assume we have some values for the x’s and the w’s. We want to change the weights to decrease the loss so we only care of computing the derivatives of the loss with respect to the w’s. In order to do this computation, we have to know the derivative for each of the elements. So in the first node we have that the upstream gradient is 1 and the local gradient is computed as the

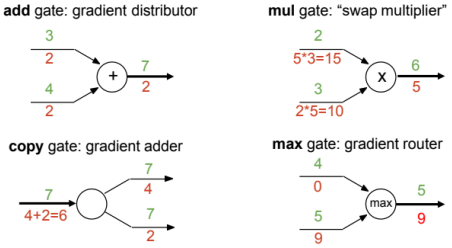
$f(x) = e^x$	$\rightarrow$	$\frac{df}{dx} = e^x$	$f(x) = \frac{1}{x}$	$\rightarrow$	$\frac{df}{dx} = -1/x^2$
$f_a(x) = ax$	$\rightarrow$	$\frac{df}{dx} = a$	$f_c(x) = c + x$	$\rightarrow$	$\frac{df}{dx} = 1$

derivative of f with respect to x (f(x) us 1/x) that is $-1/x^2$, so since x=1.37 then the local gradient is $-1/1.37^2$=-0.53. Let's make the product between the local gradient and the upstream gradient and the result is always -0.53 (this becomes the upstream gradient for the next layer). So again, the gradient of the f(x)=x+c is just 1 so the result is -0.53 (upstream gradient)*1=-0.53, and so on. Notice that at a certain point we have a block (the fork): in that case we'll compute the result by simply using the same upstream gradient. So, in the case of w_2 the local gradient is 1, the upstream gradient is 0.20 and the product is still 0.20. In the last square, we have to multiply the upstream gradient 0.20 with the local gradient -1 (that is the derivative of with respect to f(x)=ax that is a) of the other node getting 0.20*(-1)=-0.2. These two last examples are the so-called **gates**, there are several and each of them has a certain behavior.

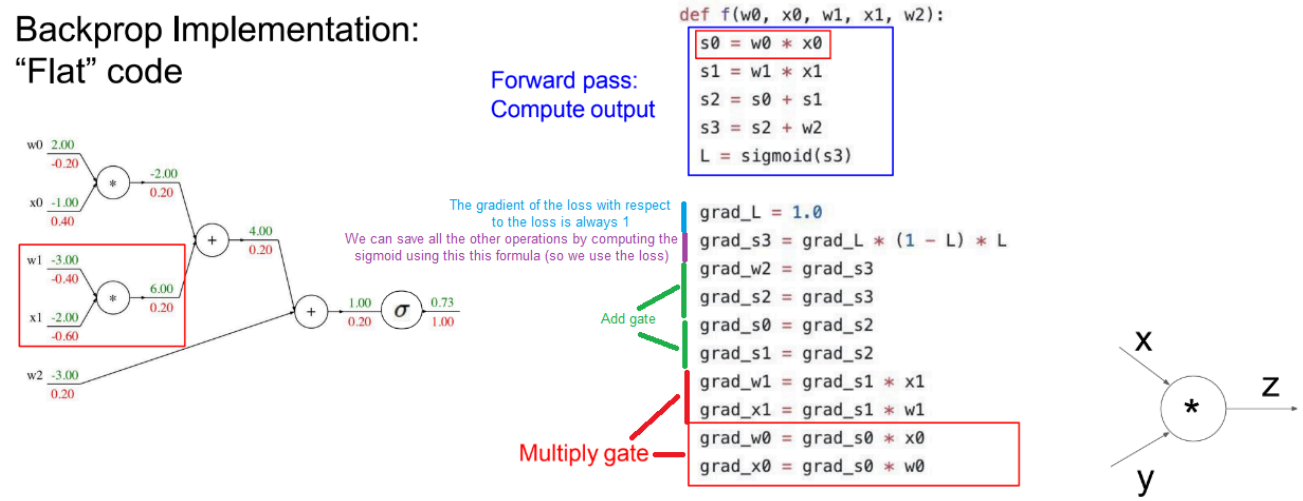
Note: Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!
Note: Notice that in order to do backpropagation, we only need the latest upstream gradient number computed but we need all the local gradients. Having these values and knowing the instructions to do all the calculations, we have everything to perform backpropagation during training.

So, we have some **gates** and the computation performed is pretty much always the same:

- Add gate (+): It is a gradient distributor, we take the upstream gradient and we copy it to all of the inputs;
- Mul gate (*): It is a swap multiplier, we take the upstream gradient and we multiply it times the other of the two inputs;
- Copy gate ( ): It is a gradient adder, we copy the information by summing them up;
- Max gate (max): It is a gradient router, we backpropagate the value where the maximum was coming from (notice that we can't say anything about the minimum so it is set at 0 when backpropagating).



EX. Let's again see the previous example but this time we given an implementation of the flat code:



So algorithmically we execute all of the nodes in a topological order and once we get the loss, we take them in a reverse order, take the upstream gradient, multiply it by the local gradient and go backward. So we recursively backpropagate the gradient across all of the nodes by following the local gradients. Again we need to save some values for use in backward. In the example on the right we need to save x and y that will be used in the backpropagation phase to compute their gradient (in that case we'll use the gradient of z computed at execution time).

```
class Multiply(torch.autograd.Function):
  @staticmethod
  def forward(ctx, x, y):
    ctx.save_for_backward(x, y)
    z = x * y
    return z
  @staticmethod
  def backward(ctx, grad_z):
    x, y = ctx.saved_tensors
    grad_x = y * grad_z   # dz/dx * dL/dz
    grad_y = x * grad_z   # dz/dy * dL/dz
    return grad_x, grad_y
```

Need to cache some values for use in backward

Upstream gradient

Multiply upstream and local gradients

So far we have seen we can have derivatives for scalars, so given two scalars x, y ∈ R, the derivative of y with respect to x tells us how much y changes if x is incremented a little bit.

What happens if we introduce vectors? We have two cases:

- If $x \in R^N$, $y \in R$ (x is a n dimensional vector) then the derivative of y with respect to all the different x is the **gradient** (so it tells us how much y changes if all x are incremented a little bit).

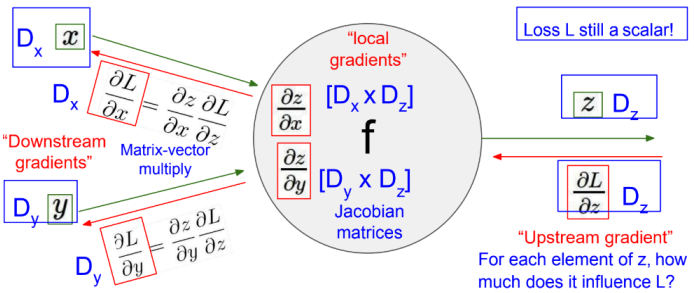
$$\frac{\partial y}{\partial x} \in \mathbb{R}^N \quad \left(\frac{\partial y}{\partial x}\right)_n = \frac{\partial y}{\partial x_n}$$

- If $x \in R^N$, $y \in R^M$ (they are both vectors) then the derivative of each of the y with respect to each of the the x is the **jacobian** that is a matrix with all the possible derivatives (so it tells us how much each y changes if each x is incremented a little bit).

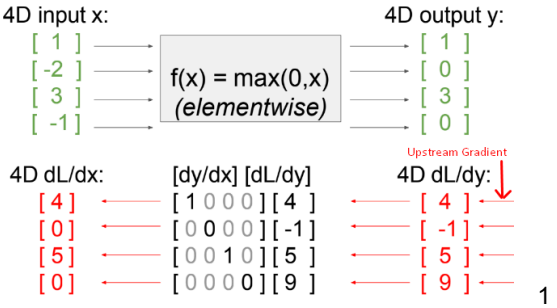
$$\frac{\partial y}{\partial x} \in \mathbb{R}^{N \times M} \quad \left(\frac{\partial y}{\partial x}\right)_{n,m} = \frac{\partial y_m}{\partial x_n}$$

How does backpropagation work with Vectors?

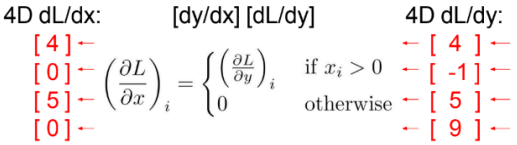
Let's assume we have a vector in x (D_x dimensional), a vector in y (D_y dimensional) and then there is a function of vector that outputs z (D_z dimensional). The derivative of L with respect to z is D_z dimensional since L is a scalar and we have that the derivative tells us how L changes when all z changes a little bit. For what concerns the **local gradients** (e.g. the derivative of z with respect to x), they are **jacobian matrices** since both elements are vectors and we can have multiple derivatives in the matrix. The chain rule still works because the local gradient has dimensionality D_x*D_z and when we multiply it by D_z then the result is D_x (remember the property of matrices).



EX. Let's see an example of how it works: We have a 4D input x, a function f(x) = max(0, x) and the output is a 4D output y. We perform backpropagation so we compute the gradient of L with respect to y using the upstream gradient (so assume to have one) and the output is the one we see. Now, for the chain rule we need the derivative of y with respect to x, which is a matrix in which all elements are on the main diagonal and we have the i-th element is or 0 depending if the element was positive or negative. We multiply it by the derivative of L with respect to y (that we have) and we come up with the output.

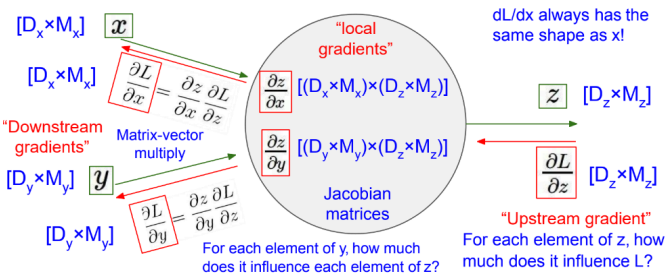


The **problem** when working with matrices is memory, saving the Jacobian matrices can be expensive so we have to think of something smarter. Notice in the previous example that **Jacobian is sparse** so **off-diagonal entries are always zero**! For this reason we can simply use the **implicit multiplication**.

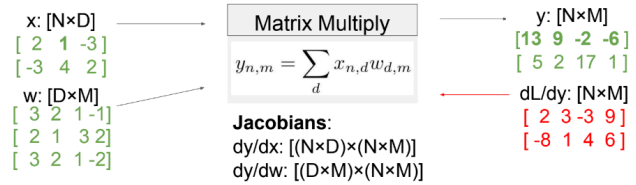


How does backpropagation work with Matrices?

Let's assume we have a matrix x (D_x*M_x dimensional), a vector in y (D_y*M_y dimensional) and then there is a function of matrices that outputs z (D_z*M_z dimensional). For the same reason as before, the gradient of the loss with respect to z is D_z*M_z dimensional. Again, the chain rule always works: assuming to compute the gradient of L with respect to x, we compute the local gradient that has dimensionality (D_x*M_x)*(D_z*M_z) and then we multiply it the upstream gradient getting a D_x*M_x dimensional matrix. The **problem is dimensionality**!



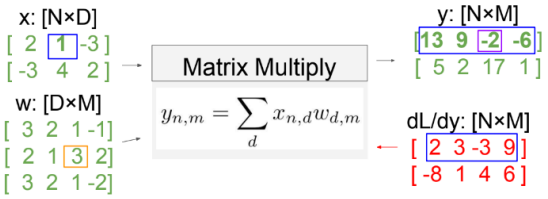
EX. Let's see an example of how it works: We have a N*D input x and a D*M input y, a function that does matrix multiplication and the output is a N*M dimensional y. We perform backpropagation so we compute the gradient of L with respect to y using the upstream gradient (so again assume to have one) and the output is N*M dimensional



again. Now, for the chain rule we need the derivative of y with respect to x and the derivative of y with respect to w that are two Jacobians and **take too much memory**.

Can we do better?

First of all we notice that an element $x_{n,d}$ affects the entire row y_n. Now, remember that the derivative of L with respect to x is the product of the derivative of L with respect to y (upstream gradient) and the derivative of y with respect to x (the local gradient). Now, we should consider both M and N according to the chain rule but we may not consider N because $y_{n,m}$ is only influenced row-wise (so we cut out the off-diagonal terms). Now, let's also notice that $x_{n,d}$ affects $y_{n,m}$ by $w_{d,m}$ meaning that the local gradient ends up being $w_{d,m}$ (We don't need to compute the Jacobians anymore). So:



$$\frac{\partial L}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} \frac{\partial y_{n,m}}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} w_{d,m}$$

Finally:

[N×D] [N×M] [M×D]

$$\frac{\partial L}{\partial x} = \left(\frac{\partial L}{\partial y}\right) w^T$$

[D×M] [D×N] [N×M]

$$\frac{\partial L}{\partial w} = x^T \left(\frac{\partial L}{\partial y}\right)$$

Summary

- **backpropagation** = recursive application of the chain rule along a computational graph to compute the gradients of all inputs/parameters/intermediates
- implementations maintain a graph structure, where the nodes implement the **forward**() / **backward**() API
- **forward**: compute result of an operation and save any intermediates needed for gradient computation in memory
- **backward**: apply the chain rule to compute the gradient of the loss function with respect to the inputs

---

So far, we are considering an x which is an image, and a set of parameters W. We have a score function f = Wx that computes a score for each class and finally we classify the image with a class. This can be considered as a **1-layer of a bigger Neural Network application**.

$$f = Wx$$
$$x \in \mathbb{R}^D, W \in \mathbb{R}^{C \times D}$$

Now, we could take the output of the first product W_1x, we apply some non-linearity that is simply the max between 0 and the output W_1x and we multiply the result for W_2: this is **2-layer Neural Network**. This is also called "fully-connected networks" in fact in our case, each pixel corresponds to the output.

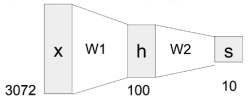
$$f = W_2 \max(0, W_1 x)$$
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

So given D the number of pixels in the image, the number of classes here is given by H*C. W_1 is the set of parameters that map from D to H.

We may also have other layers, in the case of 3-layers we have:

$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H_1 \times D}, W_2 \in \mathbb{R}^{H_2 \times H_1}, W_3 \in \mathbb{R}^{C \times H_2}$$

So we map H_1 dimension with D, H_2 dimension with H_1 and C with H_2.



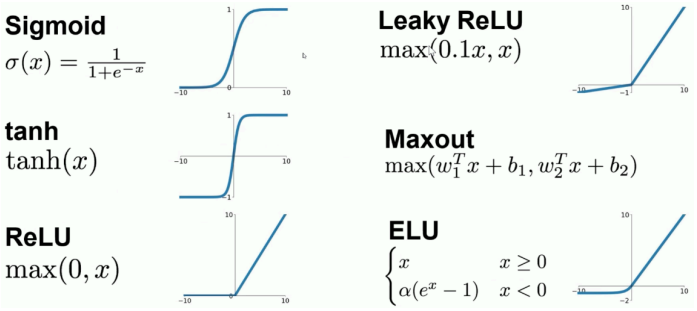
EX. Taking again the first examples we've done when introducing the topic, we start from 3072 that are all colors for all pixels in a CIFAR-10 image, we map it with W_1 getting 100 and finally, since we have 10 classes, we map it with C getting 10.

The function **max(0, z)** is called the **activation function**. This is necessary because we have to reduce the dimensionality at each step. If we build a neural network without one we end up having a linear classifier again:

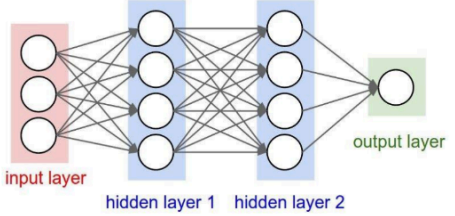
$$f = W_2 W_1 x \qquad W_3 = W_2 W_1 \in \mathbb{R}^{C \times H}, f = W_3 x$$

There are few **activation functions** (**non-linearity**) we can consider in a Neural Network:

Usually **ReLU** is the ideal choice for most of the cases.



For what concerns the architectures of a Neural Network, we have an **input layer**, an **output layer**, and a set of **hidden layers**. We consider them **fully connected layers**: starting from Wx, we have that x is D dimensional and W maps from D to H_1. The matrix multiplication means that each of the H_1 new dimensions depends on all of the input dimensionalities.

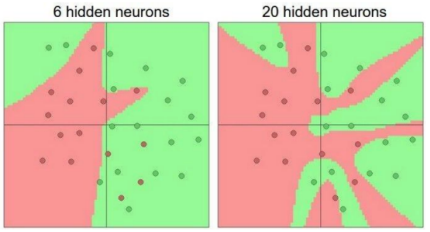



```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

Note: If we want to code how a Neural Network works, assuming the sigmoid has the activation function, we simply have to multiply the input of the previous step with the given set of parameters and add the bias. Then we apply the sigmoid function. Finally, when computed the hidden layers, we compute the output neuron. This is how the forward pass works. The backward pass is a little more tricky.
We start from the loss loss that is the square of the predicted values and the ground truth. We compute gradients with respect to w_1 and w_2 so that we can change them and arrive to hopefully decrease the loss. At the end we change w_1 and w_2 depending on the gradients and a learning rate.

```
import numpy as np
from numpy.random import randn

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = randn(N, D_in), randn(N, D_out)
w1, w2 = randn(D_in, H), randn(H, D_out)

for t in range(2000):
  h = 1 / (1 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
  print(t, loss)

  grad_y_pred = 2.0 * (y_pred - y)
  grad_w2 = h.T.dot(grad_y_pred)
  grad_h = grad_y_pred.dot(w2.T)
  grad_w1 = x.T.dot(grad_h * h * (1 - h))

  w1 -= 1e-4 * grad_w1
  w2 -= 1e-4 * grad_w2
```

What happens if we set more capacity?
If we add more neurons (hidden layers) to the Neural Network, we are basically adding more parameters, so the boundaries that divide the different samples into classes become less regular and more complex so that they can fit the training data perfectly (notice that all the training data is correctly classified and the error is 0). As we already know this isn't necessarily a good thing because we don't have enough generalization, so we are **overfitting**. We would like to have a large number of neurons so a high number of parameters avoiding the complex boundaries and we can achieve it by applying **regularization** (in general, the more the regularization by choosing a larger λ, the smoother are the boundaries).

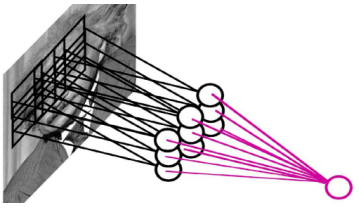


So if we **want to define a new layer** we just need the **forward pass** and the **backpropagation error** but we have to **watch out for overfitting**. If we have a small loss on the training set but a high loss on the validation set then this means we have few generalization power meaning that we are overfitting (notice that training set and validation set should have approximately the same trend).
Deep Learning is based on the availability of large dataset, massive parallel compute power and advances in machine learning over the years. The traditional approach consisted in making feature extraction often hand crafted and fixed but then apply them on a ML algorithm for classification. Instead, in Neural Networks we often talk about an **end-to-end system** meaning that the network learns the features and the classifier all together as seen before. So, again we have a number of fully connected layers with some parametrization, and after each layer we have some results W plus some non-linearity (the activation function), this is the forward pass. At the end we'll compute the error and starting from that we'll propagate the error backwards to update the parameters.
So the main ideas of Deep Learning are:

- We perform feature extraction by learning them (across many layers);
- Efficient and trainable systems by differentiable building blocks;
- Composition of deep architectures via non-linear modules (by using the activation function);
- "End-to-End" training: no differentiation between feature extraction and classification. The parameters of the classification model are learnt the same way as the features are leant;

7.5 Convolutional Neural Networks

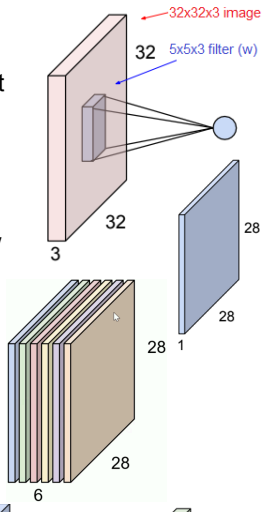
Now, let's assume we have to detect an edge in a 1000x1000 image. If we want to test whether there is an edge in each position then we'll need 1 million hidden units meaning about 10^{12} parameters (that is a huge number). Now, the edge is a local information so we may consider filters that are smaller (e.g. 10x10) so if that's the case we'll only need 100 million parameters (because we remove the stuff we're not going to use). So, in order to **understand an image** we can just **look at the local information** (**locality**). Now, another step can be done by noticing that **statistics are similar at different locations** of the image (e.g. if we look at a vertical edge, it has the same parameters anywhere we find it, so the vertical edge is the same everywhere). So we can **share the same parameters across different locations** by using **multiple filters** (**stationarity**). In this case, using 10x10 100 filters, we'll need only 10k parameters.
So, the standard Neural Network applied to images scales quadratically with the size of the input and does not leverage stationarity. The solution is to **connect each hidden unit to a small patch** of the input and **share the weight across the hidden unit**: this is called a **convolutional network**.



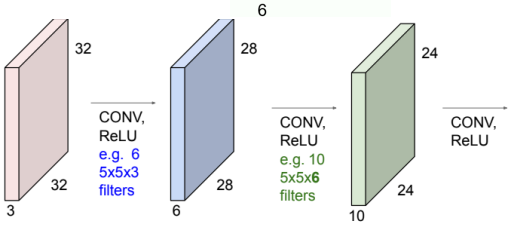
Moreover, by **pooling** (e.g. max or average) filter responses at different locations we gain robustness to the exact spatial location of features. So we have a feature that represents

a larger area (it reduces the resolution), it reduces the overall image reducing the activations (making the image more computationally feasible) and it grows some robustness to the exact location where the feature is coming from (e.g. If we have to look for a vertical edge in the image, and we apply pooling, it doesn't matter if the edge was more on the left or on the right since at the end we apply the max operation on all features we are considering and we understand where it is).

So let's see how CNN works more in detail: let's assume we have a 32x32x3 image. The input will be the stretched 3072x1 image to compute the product with the W getting a 10x3072 vector and finally we compute the activation that outputs the score for the 10 classes. Again, this is a **fully connected layer** since each number computed by the activation are connected to all of the values R, G, B in the pixels (so each element in the activation is the result of taking a dot product between a row of W and the input, a 3072-dimensional dot product).
In CNN we **don't flatten the image into a vector**, so let's assume we have a 32x32x3 image. We have a 5x5x3 filter and we **convolve** it with the image by sliding it over the image spatially computing the dot products. Notice that the depth between the image and the filter always matches, so in the kernel we also have a value for all R, G, B, so it is like an operation with colors in which we know what value to give to each of the R, G, B channels.
For each position we put the filter on, **we get a value** which is the result of taking a dot product between the filter and a small 5x5x3 chunk of the image (i.e. 5*5*3 = 75-dimensional dot product + bias): $\mathbf{w^T+b}$
Doing the convolution over all spatial locations between the input image and the filter, we get to a new 28x28x1 image called the **activation map**. Notice that we lose one position in all corners due to the application of the filter, if we want to keep the same size of the initial image we can apply **padding** by adding a corner of 0 around the image.
Now, the activation function we have computed can be used to detect something in the image (e.g. vertical edges). If we want to detect more things at once (e.g. we also want to detect horizontal edges) we can compute other activation maps. So we add other channels as the number of filters we consider each of size 28x28x1 (e.g. if we consider 6 activation maps we'll get to a 28x28x6 image).

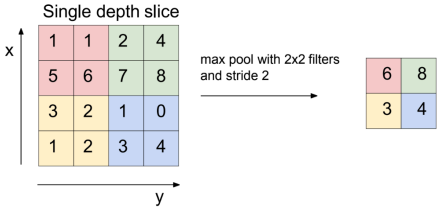


Out of the first convolution layer we can get to an activation map that looks like an image with the only difference that we lose the RGB channels for k channels where k is the number of convolutional kernels (e.g. 6 in the last example). We can define a new convolutional layer and so on and so forth (notice that we must use filters with a depth as much as the depth of the activation map, e.g. 6).



Note: In a fully connected layer, if we had got two fully connected layers, we had to put some non-linearity in the middle otherwise we would end up with only one layer. The same problem happens in CNN due to the associativity of the convolutions: Given x an image and w_1, w_2 two kernels, we may apply $[x(w_1)]w_2$ (so we could apply the first kernel to the image and then the result to the second image) but also $(w_1w_2)x$ (so we multiply the kernels getting a new kernel w_3 and then apply it to the image getting a single layer). So we **apply ReLU** that returns a 0 if we have a negative value, otherwise the actual value.
Note: If we choose a larger filter (let's say a filter as big as the input image) we are going to have the operation equivalent to a fully connected one. In that case we'll get a 1x1 activation map and it is the same as having a 1D layer fully connected. The output is a function of all pixels in the image. If we consider smaller patches, the output will be a function of a smaller number of pixels (the ones corresponding to the patch) but if we consider multiple layers, the activation later on in the network will be a function of a larger path in the input image.
Note: We usually slide the filter of one pixel but it is possible to change this value (usually it is possible to do so by changing the **stride** value). In the case stride = 2 we'll get an activation map which is half the original image. If we lower the resolution and if we have a lot of layers, the complexity is a little lower.

To make the **representations smaller** and **more manageable** we can apply a **pooling layer** that operates over each activation map independently (so apply it in each channel independently, getting the same number of channels in the new downsampled image). One example is **max pooling**: let's assume we have a certain activation map, if we consider 2x2 filters and stride 2 when pooling, for each of the blocks we consider the maximum. The effect is that we get another



activation map that is half of the size. Pooling is an alternative to convolution with strides.

SUM. Convolutional Neural Networks (CNN) are a stack of convolutions, pooling and fully connected layers. There is a trend on using smaller filters and deeper architectures. Also the trend is towards getting rid of pooling and fully connected layers and using just convolution. The historically architectures looked like [(CONV-RELU)*N-POOL?]*M-(FC-RELU)*K,SOFTMAX where N is usually up to ~5, M is large, 0 <= K <= 2 but recent advances such as ResNet/GoogLeNet have challenged this paradigm.

8. Probability Theory Review 2

Let's make a recap of probability:

- **Covariance:** it measures how much one RV's value tends to move with another RV's value. For RV's X, Y:

  $$Cov[X, Y] := \mathrm{E}[(X - \mathrm{E}[X])(Y - \mathrm{E}[Y])] = \mathrm{E}[XY] - \mathrm{E}[X]\mathrm{E}[Y]$$

  If Cov [X, Y] < 0, then X and Y are negatively correlated, If Cov [X, Y] > 0, then X and Y are positively correlated, If Cov [X, Y] = 0, then X and Y are uncorrelated.
  If $X \perp Y$, then E[XY] = E[X]E[Y]. Thus, Cov[X, Y] = E[XY] − E[X]E[Y] = 0 (this is unidirectional: Cov [X, Y] = 0 does not imply $X \perp Y$).
- **Variance of two variables:** Var[X + Y] = Var[X] + Var[Y] + 2Cov[X, Y] (i.e. if $X \perp Y$, Var[X + Y] = Var[X] + Var[Y]). A **special case** is the following: Cov[X, X] = E[XX] − E[X]E[X] = Var[X].
- **Covariance Matrices**: For a random vector $X \in R^n$, we define its covariance matrix Σ as the nxn matrix whose ij-th entry contains the covariance between X_i and X_j:

  $$\Sigma = \begin{bmatrix} Cov[X_1, X_1] & \dots & Cov[X_1, X_n] \\ \vdots & \ddots & \vdots \\ Cov[X_n, X_1] & \dots & Cov[X_n, X_n] \end{bmatrix}$$

  Applying linearity of expectation and the fact that $Cov[X_i, X_j] = E[(X_i - E[X_i])(X_j - E[X_j])]$, we obtain $\Sigma = E[(X - E[X])(X - E[X])^T]$. Some properties are:
  - Σ is symmetric and PSD (all the eigenvalues need to be non negative);
  - If $X_i \perp X_j$ for all i,j, then $\Sigma = \mathrm{diag}(\mathrm{Var}[X_1],...,\mathrm{Var}[X_n])$;
- **Multivariate Gaussian**: The multivariate Gaussian $X \sim N(\mu, \Sigma)$, $X \in R^n$:

  $$p(x; \mu, \Sigma) = \frac{1}{det(\Sigma)^{\frac{1}{2}}(2\pi)^{\frac{n}{2}}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$$

  The univariate Gaussian $X \sim N(\mu, \sigma^2)$, $X \in R$ is just the special case of the multivariate Gaussian when n = 1:

  $$p(x; \mu, \sigma^2) = \frac{1}{\sigma(2\pi)^{\frac{1}{2}}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right)$$

  Notice that if $\Sigma \in R^{1\times1}$, then $\Sigma = \mathrm{Var}[X_1] = \sigma^2$, and so
  - $\Sigma^{-1} = 1/\sigma^2$;
  - $\det(\Sigma)^{½} = \sigma$;
- **Some Nice Properties of MV Gaussians**:
  - Marginals and conditionals of a joint Gaussian are Gaussian.
  - A d-dimensional Gaussian $X \in N(\mu, \Sigma = \mathrm{diag}(\sigma^2_1, .., \sigma^2_n))$ is equivalent to a collection of independent Gaussians $X_i \in N(\mu_i, \sigma^2_i)$. This results in isocontours aligned with the coordinate axes.
  - In general, the isocontours of a MV Gaussian are n-dimensional ellipsoids with principal axes in the directions of the eigenvectors of covariance matrix Σ (remember, Σ is PSD, so all n eigenvalues are non-negative). The axes' relative lengths depend on the eigenvalues of Σ.

Note: some visualizations of MV Gaussians → slide.

9. Generative Algorithms, Gaussian Discriminant Analysis, Naive Bayes

9.1 Generative Learning Algorithms

Let's assume we have a training dataset in which we have two possible classes (e.g. malignant/benign cell). Our goal is to separate these two classes and predict correctly if the tumor cell is malignant or benign. What we have seen so far with the previous classification algorithms (e.g. Logistic Regression, Neural Networks) are the **discriminative models** that directly learns the probability of y given x, or estimates the class of the given sample x given a function f(x) that is our hypothesis.
A **generative model**'s main goal is to learn the probability of x given y (where x are the features and y the class) and the probability of the class y (the **class prior**):

$$\text{LEARNS} \quad p(x \mid y) \qquad p(y)$$

Instead of directly computing a hyperplane that allows us to get a decision of 0 or 1, here we **try to model what the features look like** in the case of positive or negative class. In order to do so, it is possible to use the **Bayes Rule**:

$$p(y=1 \mid x) = \frac{p(x \mid y=1)\,p(y=1)}{p(x)} \qquad p(x) = p(x \mid y=1)\,p(y=1) + p(x \mid y=0)\,p(y=0)$$

So, we can apply the rule and compute the probability of y=1 given x is going to be the decision, and in order to take it we need the normalizing factor (at the denominator) that is the probability of x equivalent to computing the probability of x in the case of the two classes.

9.2 Gaussian Discriminant Analysis

Now, let's get into one generative model that is the **Gaussian Discriminant Analysis (GDA)**. We suppose to have $x \in R^d$ to be d-dimensional and we assume the **probability of x given y is gaussian**. Notice that here the convection $\mathbf{x_0=1}$ that allowed us to make vector products **is dropped**.

A random variable $z \in R^d$ is distributed like a gaussian with mean $\vec{\mu} \in R^d$ and variance $\Sigma \in R^{dxd}$ (it is a square matrix). The **probability density function of z** can be written as:

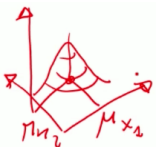
$$\text{pdf:} \quad p(z) = \frac{1}{(2\pi)^{\frac{d}{2}} |\Sigma|^{\frac{1}{2}}} \exp\left(-\frac{1}{2}(z-\mu)^T \Sigma^{-1} (z-\mu)\right) \qquad z \sim N(\vec{\mu}, \vec{\Sigma})$$

Each element in μ_i in μ is the expected value of z_i and each value Σ_{ij} in the covariance matrix Σ is the expected value $z_i z_j$ minus the expected value of the two:

$$\mu_i = E[z_i] \qquad \Sigma_{ij} = E[z_i z_j] - E[z_i]E[z_j]$$

Like seen previously, we can rewrite the probability of z like the vector $\vec{\mu}$ and the covariance matrix of z is equal to:

$$E[z] = \vec{\mu}$$
$$\text{Cov}(z) = E\left[(z-\mu)(z-\mu)^T\right] = E(zz^T) - E[z]E[z]^T$$



In the case of GDA, we will assume to **model the features of each of the 2 classes with 2 gaussians**. We can write 2 (bidimensional) multivariate gaussian for the case of x given y=0 and y=1 (d is the number of features):

$$p(x|y=0) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_0)^T \Sigma^{-1}(x-\mu_0)\right)$$

$$p(x|y=1) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_1)^T \Sigma^{-1}(x-\mu_1)\right)$$

The parameters of the model are μ0, μ1∈R^d, Σ∈R^{dxd} (the covariance matrix) and Φ∈[0, 1] (the classes).
The probability of y will be given by:

$$p(y) = \phi^y (1-\phi)^{1-y}$$

Now, remember that we want to maximize the likelihood that, in the case of the discriminative models, was computed like:

$$L(\theta) = \prod_{i=1}^{m} p(y^{(i)}|x^{(i)};\theta) \qquad \text{TRAINING SET} : \left\{x^{(i)}, y^{(i)}\right\}_{i=1}^{m}$$

In the case of the Generative Model, we model x and y jointly and so, instead of writing the likelihood as the product of the probability of y, we'll write the **joint likelihood** of all parameters. It is equal to the product of the joint distribution of x^i and y^i parametrized to the parameters that can be rewritten like:

$$L(\phi,\mu_0,\mu_1,\Sigma) = \prod_{i=1}^{m} p(x^{(i)}, y^{(i)};\phi,\mu_0,\mu_1,\Sigma) = \prod_{i=1}^{m} p(x^{(i)}|y^{(i)})\, p(y^{(i)})$$

We want to perform **Maximum Likelihood Estimation (MLE)** so we try to maximize the log likelihood:

$$\max_{\phi,\mu_0,\mu_1,\Sigma} \ell(\phi,\mu_0,\mu_1,\Sigma) = \log L(\dots)$$

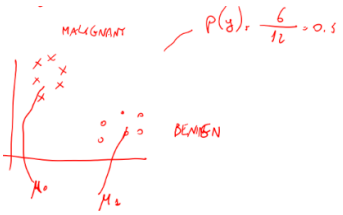
What comes out from the MLE is that Φ is ratio of the sample that has class y=1 (note that 1 is the **indicator function**).

$$\phi = \frac{\sum_{i=1}^{m} y^{(i)}}{m} = \frac{\sum_{i=1}^{m} 1\{y^{(i)}=1\}}{m}$$

$$1\{\text{true}\} = 1$$
$$1\{\text{FALSE}\} = 0$$

The **mean** for the features for a specific class are given by:

$$\mu_0 = \frac{\sum_{i=1}^{m} 1\{y^{(i)}=0\}\, x^{(i)}}{\sum_{i=1}^{m} 1\{y^{(i)}=0\}} \qquad \mu_1 = \frac{\sum_{i=1}^{m} 1\{y^{(i)}=1\}\, x^{(i)}}{\sum_{i=1}^{m} 1\{y^{(i)}=1\}}$$



EX. μ_0 is the mean of the features that are from class 0.
The **covariance matrix Σ** is computed considering all the samples:

$$\Sigma = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \mu_{y^{(i)}}\right)\left(x^{(i)} - \mu_{y^{(i)}}\right)^T$$

Now, in order to make a **prediction with GDA**, we need to consider the posterior meaning the probability of y given x, in particular the class that maximizes the probability. We can then rewrite it using Bayes' Rule:

$$\arg\max_y p(y|x) = \arg\max_y \frac{p(x|y)\,p(y)}{p(x)} = \arg\max_y p(x|y)\,p(y)$$

In the last step, since we want to maximize P(y) and P(x) doesn't depend on y, we can delete it.
If P(y) is **uniform** (meaning that **P(y=1)=P(y=0)**) then the formula reduces to:

$$\arg\max_y p(x|y)$$

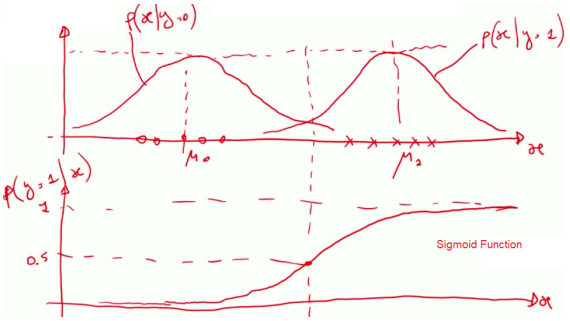
Now, let's see which is the relation between GDA and Logistic Regression: first of all, for fixed μ0, μ1, Σ, Φ we want to plot the probability of y=1 given x parametrized by μ0, μ1, Σ, Φ as a function of x:

$$p(y=1|x;\phi,\mu_0,\mu_1,\Sigma) \quad \text{AS A FN. OF } x$$

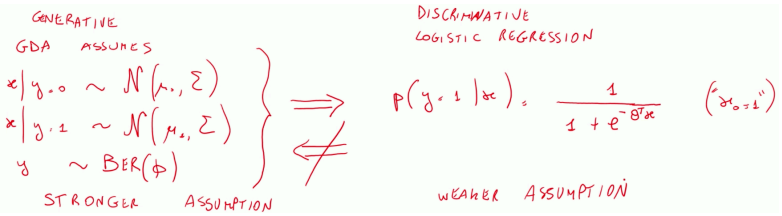
This is the result of applying Bayes' Rule that is:

$$\frac{p(x|y=1;\mu_1,\Sigma)\cdot p(y=1;\phi)}{p(x;\mu_0,\mu_1,\Sigma,\phi)} \rightarrow p(x|y=0;\mu_0,\Sigma)\,p(y=0;\phi)+p(x|y=1;\mu_1,\Sigma)\,p(y=1;\phi)$$

EX. Let's assume we have 2 classes: x is class 1 and o is class 0 and they are distributed according to the picture. So, we have 5 samples for each class (10 in total). The probability of y=1 is P(y=1)=5/10=0.5. We have 2 gaussians that model the features of the 2 classes. They are using the same variance so the shape of the two gaussians is the same, just shifted depending on the μ value of the class. So, the 2 gaussians have the same max value. Now, given the two gaussians, we want to know the probability of y=1 given x: we can notice that at the beginning the probability is low because the gaussian for the negative class is much larger than the gaussian for the positive class. At some point the behavior will switch since the gaussian for the positive class will become predominant towards the left area. The point where the two gaussians cross is where we have the same probability is y (that is 0.5). The curve that represents the probability of y=1 given x is modeled like a **sigmoid function**.



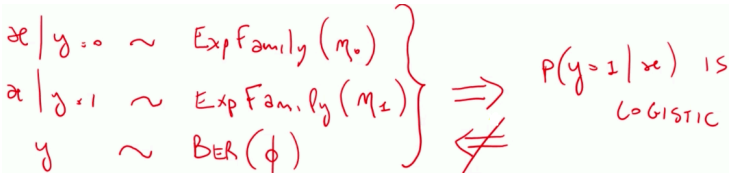
So, GDA assumes that the probability of y=0 and the probability of Y=1 are two gaussians and the probability of y is distributed like a Bernoulli: this means the probability of y=1 given x is actually a sigmoid function that can be reconducted to the discriminative case of the logistic regression where the probability of y=1 given x is equal to $1/(1+e^{-\Theta^T x})$ (remember that in this case $x_0=1$).



The opposite is not true, so if we start with logistic regression and we model the probability of y=1 given x with a sigmoid function, we won't end up with the GDA assumption.

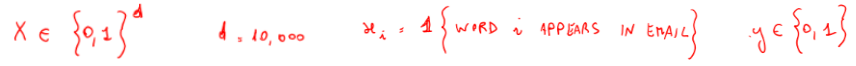
Note: If we assume the features are gaussian we should expect that adopting the GDA we can use fewer data and get a better result. Otherwise we are better off with a weaker assumption and stick to logistic regression.

Note: The same assumptions made so far are valid when the features are distributed like any distribution from the exponential family of distribution (so if the features are distributed like a Poisson, Gaussian and so on..):

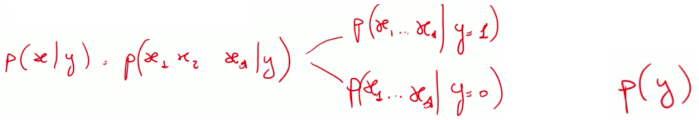


9.3 Naïve Bayes

Let's assume we want to build a spam filter. We have a feature vector x that corresponds to the english dictionary. If one of the words appears in the document, then there is a 1 in the corresponding point of the vector x (otherwise a 0). So x is d dimensional vector that can take values 0 and 1 and x_i can be represented by an indicator function. The y is 0 or 1 (the document is spam or non spam).



Assuming to use a **generative model**, we want to model the probability of x given y, that in the general case is equal to the **joint probability** of x_1, x_2,.. ,x_d given y. We need to model the probability of the features in the case of y=1 (so a spam email) and y=0 so a non-spam email. We also need to model the prior P(y).



The possible values x can take are $2^d=2^{10,000}$ (because each x can take values 0 or 1). We call the model we will be using the **Multivariate Bernoulli Event Model** (Multivariate because the input is given by multiple random variables, Bernoulli because it is a binary model). In the general case the number of parameters is:

$$\#\ \text{PARAMETERS}: 2\cdot(2^d-1)+1$$

So, 2^d because we need to have a probability value for each of the possible states for the feature vector, -1 because they integrate to 1. Then multiply by 2 because we need to model the possible state in the case of the two different distributions we consider (spam email or non-spam email)+1. Since there are a lot of parameters, the **Naive Bayes assumption** comes to the rescue:
Let's assume the **x_i's are conditionally independent given y** (so the probability of x_i is independent from the probability of x_j), firstly we apply the chain rule:

$$P(x_1 \dots x_{10000} | y) = P(x_1|y)\cdot P(x_2|x_1,y)\cdot \dots \cdot P(x_{10000}|x_{9999}\dots x_1, y)$$

Out of the assumption, we have that:

$$\overset{(\text{ASSUME})}{=} P(x_1|y)\,P(x_2|y)\dots P(x_{10,000}|y) = \prod_{i=1}^{d} P(x_i|y)$$

So, in Naive Bayes the parameters are going to be:

$$\phi_{j|y=1} = P(x_j=1|y=1) \quad \phi_{j|y=0} = P(x_j=1|y=0) \quad \phi_y = P(y=1)$$

So we have that Φ_j given y=1 is the probability of x_j=1 given y=1 and Φ_j given y=0 is the probability of x_j=1 given y=0. We also have Φ_y that is the parameter for the prior y that is the probability of y=1.
The Joint Likelihood is (n are the training samples):

$$\mathcal{L}(\phi_y, \phi_{j|y}) = \prod_{i=1}^{n} P(x^{(i)}, y^{(i)}; \phi_y, \phi_{j|y})$$

So the likelihood is a function of Φ_y and $\Phi_{j|y}$ that is the product for all training samples of the joint probability of x^i and y^i (the feature and the label) given Φ_y and $\Phi_{j|y}$. In this case of the Naive Bayes assumption we consider 2d+1 parameters (this is more manageable). Once we have the likelihood is always to maximize it. So:

$$\phi_y = \frac{\sum_{i=1}^{n} 1\{y^{(i)}=1\}}{n}$$

EX. The emails that were spam over the total number of emails.
And:

$$\phi_{j|y=1} = \frac{\sum_{i=1}^{n} 1\{x_j^{(i)}=1, y^{(i)}=1\}}{\sum_{i=1}^{n} 1\{y^{(i)}=1\}} \qquad \phi_{j|y=0} = \frac{\sum_{i=1}^{n} 1\{x_j^{(i)}=1, y^{(i)}=0\}}{\sum_{i=1}^{n} 1\{y^{(i)}=0\}}$$

EX. The emails that contain the word j and were spam over the total number of emails classified as spam. The emails that contain the word j and weren't spam over the total number of emails classified as non-spam.
To **decide whether an email is spam or not spam** at **prediction time**, we have apply Bayes Rule and compute the posterior:

$$P(y=1|x) = \frac{P(x|y=1)\,P(y=1)}{P(x|y=1)\,P(y=1) + P(x|y=0)\,P(y=0)}$$

Now, let's see how we can apply it in **image recognition** (e.g. digit recognition). Let's suppose we have a black and white image that represents a digit. Each pixel represents the random variable $X_1, .., X_n \in \{0, 1\}$ that can be both 0 or 1 (black and white). The target variable, in our case, is $Y \in \{5, 6\}$ (so we want to predict if the digit in the image is a 5 or a 6). We want to predict the class of the image given the features, in particular the class that gives the maximum probability:

$$\arg\max_Y P(Y|X_1,\dots,X_n)$$

$$P(Y|X_1,\dots,X_n) = \frac{P(X_1,\dots,X_n|Y)P(Y)}{P(X_1,\dots,X_n)}$$

Likelihood — Prior — Normalization Constant

We apply Bayes Rule so that we can split the posterior probability into the likelihood and the prior and then there is a normalization constant which is used to normalize the probability.
We can expand the last formula in the case of digit recognition:

$$P(Y=5|X_1,\ldots,X_n) = \frac{P(X_1,\ldots,X_n|Y=5)P(Y=5)}{P(X_1,\ldots,X_n|Y=5)P(Y=5)+P(X_1,\ldots,X_n|Y=6)P(Y=6)}$$
$$P(Y=6|X_1,\ldots,X_n) = \frac{P(X_1,\ldots,X_n|Y=6)P(Y=6)}{P(X_1,\ldots,X_n|Y=5)P(Y=5)+P(X_1,\ldots,X_n|Y=6)P(Y=6)}$$

To classify, we'll simply **compute these two probabilities** and **predict based on which one is greater** (notice that we could simply compare the numerators to make the prediction and choose the greatest).
For the Bayes classifier, we need to "learn" two functions, the **likelihood** and the **prior**. For the prior we only need 1 parameter for the case of digit recognition because we just need to decide whether it is the class, and since it is a Bernoulli Distribution the probability will simply tell which is the class.
Assuming to have a 30x30 pixels image, in the general case (without the Naive Bayes assumption) we need 2^{900}-1 parameters for each class so it is $2(2^{900}-1)$ for the likelihood. In total (prior and likelihood) we need $2(2^{900}-1)+1$ parameters (a lot!).
So, again, we can take the **Naive Bayes assumption** into account and assume that all features are **conditionally independent given the class label Y** (e.g. all pixels in the image are independent):

$$P(X_1,\ldots,X_n|Y) = \prod_{i=1}^{n} P(X_i|Y)$$

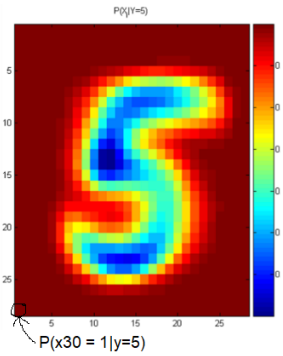
With this assumption we only need 2n parameters for the likelihood (+1 for the prior).
Now that we've decided to use a Naïve Bayes classifier, we need to **train it with some data**. So:

- We need to estimate P(Y=v) as the fraction of records with Y=v:

$$P(Y=v) = \frac{Count(Y=v)}{\#\ records}$$

- Estimate $P(X_i=u|Y=v)$ as the fraction of records with Y=v for which X_i=u:

$$P(X_i=u|Y=v) = \frac{Count(X_i=u \wedge Y=v)}{Count(Y=v)}$$



So, for each pixel in the image X_i we count which is the probability that the pixel assume one value for each of the class (e.g. the number of times the pixel x_{30} assumes the value 1, so it is a white pixel, given class Y=5 divided by the times the image is classified with Y=5. Notice that in the image we have pixels in which the probability of being white is larger, that is the way a digit is recognized).

For the **testing** we need to compute the probability of each image and compare it in the case of having one of the different classes so the posterior. For example: for each pixel in the previous image we compute the product for each pixel being equal to the value it assumes given Y=5. Then we normalize the product by summing each product computed for each of the classes we have.
If we do it for all classes, we'll get a probability for each class that represents the confidence that the image is that specific class. We have to select the image with the highest probability.

$$P(x_1=0|y=5)\,P(x_2=0|y=5)\,P(x_{55}=1|y=5)\ldots P(y=5)$$
$$* + P(x_1=0|y=6)\,P(x_2=0|y=6)\ldots$$

Laplace Smoothing is a way to address the issue where we have **no observation on a certain feature** (e.g. if we take the vocabulary example, we have a situation in which we never saw a document with a certain word).
Let's assume we model some documents using the Multivariate Bernoulli Event Model with the Naive Bayes assumption that each feature X_i is conditionally independent. We have a feature x_{8000} that has never appeared in any document in each class we are considering (so the document is spam y=1 or not spam y=0). If we compute the probability of y=1 given x so:

$$P(y=1|x) = \frac{P(x|y=1)\,P(y=1)}{P(x|y=1)P(y=1)+P(x|y=0)P(y=0)}$$

$$P(x|y=1) = \prod_{j=1}^{10,000} P(x_j|y=1)$$

According to Naive Bayes, the P(x|y=1) is the product for j that varies from 1 to d=10,000 (vocabulary size) of the probability of x_j given y=1. For the case x_{8000} we have that the probability is 0 and since we have a product, the whole product will be 0!

EX. Assuming Lazio Team has never won in the previous 5 competitions we need to predict if it will win against Roma in the next competition. We need to compute the probability of y=1 (it will win) that is given by the number of times it has already won (0) over the time it has won plus the times it hasn't won (0+5). The result is 0 as before! A way to solve might be assuming we have at least a 1 both for the positive and negative cases, meaning:

$$p(y=1) = \frac{0+1}{0+1+5+1} = \frac{1}{7} = \frac{\#\text{"1"s} + 1}{\#\text{"1"s} + 1 + \#\text{"0"s} + 1}$$

More generally:

$$\text{IF } y \in \{1, \dots, k\} \qquad P(y=j) = \frac{\left(\sum_{i=1}^{n} 1\{y^{(i)} = j\}\right) + 1}{n + k}$$

So we have to count how many times y=j, we add a 1 and we divide everything for every possible value of y plus the number of classes (k in the general case).
So, we may need to write the **Naive Bayes parameters considering the Laplace smoothing**:

$$\phi_{j|y=1} = \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1, y^{(i)} = 1\} + 1}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\} + 2} = p(x_j = 1 | y = 1)$$

Note: In the denominator we sum 2 because we only have 2 classes.

Up to this moment, we were considering the case in which X_i had a Bernoulli distribution so it could take one of 2 values. In case we have features that can take k possible value (so they are not binary), we talk about **Multinomial Event Model**. One problem with the previous representation is that **words were not counted** so we didn't know how many times a feature occurred **losing some information** (e.g. in the email example, we had a vector of words and each words could assume a value 1 or 0 depending on the fact the word is in the document or not). So, let's first of all define a new representation. The new feature vector x we'll be considering will be d_i dimensional where d_i is the length of the email. So, we can represent the email with a vector like this:

$$x = \begin{bmatrix} 1600 \\ 1600 \\ 800 \\ \vdots \\ 1600 \end{bmatrix} \in \mathbb{R}^{d_i}$$

$$\text{EMAIL } i: \begin{bmatrix} x_1^{(i)}, x_2^{(i)}, x_3^{(i)}, \dots, x_{d_i}^{(i)} \end{bmatrix} \qquad x_j \in \{1, \dots, 10{,}000\}$$

Notice that **neither of the models** (the multivariate and the multinomial one), **take the word order into account**, but we solve the problem of counting since now we know how many times a word appears in the document.

In the **multinomial event model**, we assume that the way an email is generated is via a random process in which spam/non-spam is first determined (according to P(y)) as before. Then, the sender of the email writes the email by first generating x_1 from some multinomial distribution over words ($P(x_1|y)$). Next, the second word x_2 is chosen independently of x_1 but from the same multinomial distribution, and similarly for x_3 , x_4, and so on, until all n words of the email have been generated. Let's recall the definition of **conditional probability** to find the **joint probability of the data**:

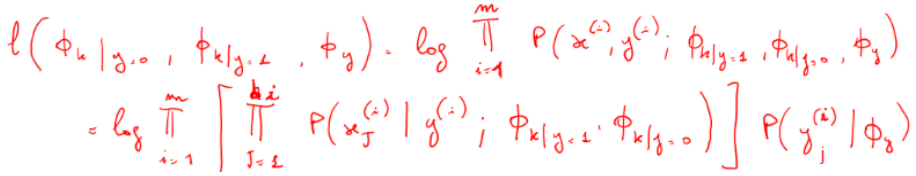
$$p(x, y) = p(x|y)\,p(y) = \left(\prod_{j=1}^{d_i} p(x_j|y)\right) p(y)$$

So, as before we apply the Naive Bayes assumption and assume that all features are **conditionally independent given the class label Y.** The only difference with the multivariate model is that $P(x_j|y)$ is a multinomial distribution instead of a Bernoulli one.
The parameters for this model are:

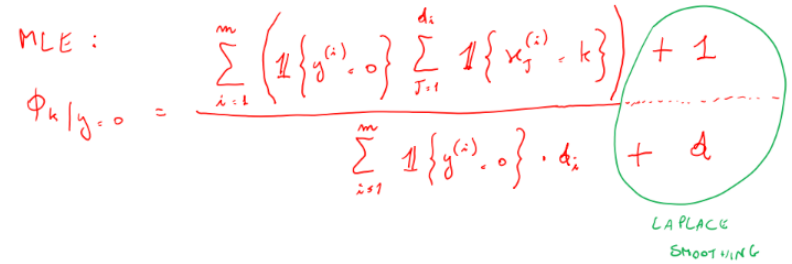
$$\phi_y = p(y=1) \qquad \phi_{k|y=0} = p(x_j = k | y = 0) \qquad \phi_{k|y=1} = p(x_j = k | y = 1)$$

So $\Phi_{k|y=0}$ is basically the probability of finding a word j being k if y=0 where x_j is the index (the position in the dictionary of the j-th word in the email). Again, we **don't consider the order of the words**.

The likelihood is the following:

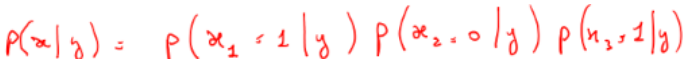


Again, we want to perform **Maximum Likelihood Estimation (MLE)** so:

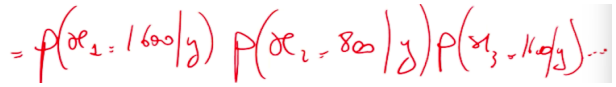


This parameter $\Phi_{k|y=0}$ is the probability that word k occurs somewhere in an email given that it is not spam. If d_i is the length of the i-th email, then what we're doing is going through the entire training set, and counting the number of times that both these conditions are met. The second part checks that the particular word is the k-th word, and the first condition checks that it's not spam. This is then divided by the total length of all the non spam emails. You can see this by noting that in the denominator, the expression inside the summation first checks whether the email is not spam (if it is, then it adds 0), and if so, multiplies the 1 with the length of that email. We also apply **Laplace Smoothing** so we add a 1 in the numerator and a 1 for each example in the denominator (so d = 10,000). A similar notation can be given in the case of a spam email (just change $y^{(i)}=0$ with $y^{(i)}=1$): this is the other parameter $\Phi_{k|y=1}$. Φ_y is the same a the multivariate case, so it counts the number of spam email and divides that by the number of examples.

SUM. In the Multivariate Bernoulli Event Model we need to consider P(x|y)P(y) where P(y) simply consider how many spam or non-spam emails we have while, assuming Naive Bayes, P(x|y) we need to consider the features we have so in order to compute the likelihood we consider the probability that each feature takes a certain value given a certain y. For example:



We do it for y=0 and y=1, we compare the 2 probabilities and we end up with a decision (depending on a threshold). In the case of the Multinomial Event Model we have that the likelihood is equal to the probability that each position in the document takes a particular value (word) given a certain y:



Again, we do it for y=0 and y=1 and we make the decision accordingly.

EX. We have data that tells us how many times we played and how many times we didn't play and for each of these cases we know what the weather was like.

The weather data, with counts and probabilities

	outlook		temperature		humidity		windy		play	
	yes	no	yes	no		yes	no		yes	no	yes	no
sunny	2	3	hot	2	2	high	3	4	false	6	2	9	5
overcast	4	0	mild	4	2	normal	6	1	true	3	3		
rainy	3	2	cool	3	1								
sunny	2/9	3/5	hot	2/9	2/5	high	3/9	4/5	false	6/9	2/5	9/14	5/14
overcast	4/9	0/5	mild	4/9	2/5	normal	6/9	1/5	true	3/9	3/5		
rainy	3/9	2/5	cool	3/9	1/5								

$P(y = YES \mid x_1 = SUNNY, x_2 = COOL, x_3 = HIGH, x_4 = T)$

A new day

outlook	temperature	humidity	windy	play
sunny	cool	high	true	?

We have to give the probability of the new day given y=0 and y=1. So, the formula we have to use is the one that allows us to predict something at prediction time: we have to compute the probability of x given y=yes multiplied by the prior p(y=yes), all divided by the probability p(x). So:

- $P(x|y=1)=P(x_1|y=1)P(x_2|y=1)P(x_3|y=1)P(x_4|y=1)=2/9*3/9*3/9*3/9=54/6561=0.0082$
- $P(x|y=0)=P(x_1|y=0)P(x_2|y=0)P(x_3|y=0)P(x_4|y=0)=3/5*1/5*4/5*3/5=36/625=0.057$
- $P(y=1)=9/14=0.64$
- $P(y=0)=5/14=0.357$
- $P(x)=P(x|y=1)P(y=1)+P(x|y=0)P(y=0)=0.0083*0.64+0.057*0.357=0.0053+0.026=0.0259$

So:

- Likelihood of yes: $P(x|y=1)P(y=1)=0.0052$
- Likelihood of no: $P(x|y=0)P(y=0)=0.0206$

Final result: $P(y=1|x_1,x_2,x_3,x_4)=P(x|y=1)*P(y=1)/P(x)=0.0053/0.0259=0.2046$

Notice that the probability of playing is about 20% (the probability of playing is higher) so the answer is **no**.

If we have **continuous values** that come from a gaussian distribution we can get P(x|y=1) we have to consider the mean of those values and the standard deviation. So:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\sigma = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu)^2$$

$$f(w) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(w-\mu)^2}{\sigma^2}}$$

EX. Let's assume that at test time we get that on the new day we get that it is sunny, the temperature is 66 and so on. To compute the likelihood of yes and the likelihood of no we need to compute $P(x_2=66|y=yes)$ so:

The numeric weather data with summary statistics

	outlook			temperature			humidity			windy			play	
	yes	no		yes	no		yes	no		yes	no		yes	no
sunny	2	3		83	85		86	85	false	6	2		9	5
overcast	4	0		70	80		96	90	true	3	3			
rainy	3	2		68	65		80	70						
				64	72		65	95						
				69	71		70	91						
				75			80							
				75			70							
				72			90							
				81			75							
sunny	2/9	3/5	mean	73	74.6	mean	79.1	86.2	false	6/9	2/5		9/14	5/14
overcast	4/9	0/5	std dev	6.2	7.9	std dev	10.2	9.7	true	3/9	3/5			
rainy	3/9	2/5												

$$f(\text{temperature} = 66|\text{Yes}) = \frac{1}{\sqrt{2\pi}(6.2)} e^{-\frac{(66-73)^2}{2(6.2)^2}} = 0.0340$$

$p(x_2 = 66 \mid y = yes)$ $\quad p(x_2 \mid y, 1) = f(x_2; \mu, \sigma)$

- Likelihood of Yes = $\frac{2}{9} \times 0.0340 \times 0.0221 \times \frac{3}{9} \times \frac{9}{14} = 0.000036$
- Likelihood of No = $\frac{3}{5} \times 0.0291 \times 0.038 \times \frac{3}{5} \times \frac{5}{14} = 0.000136$

It is often the case that machine learning algorithms need to work with very small

Recall: *log(xy) = log(x) + log(y)*

numbers and multiplying lots of probabilities can result in floating-point underflow (the final result is 0), so we need to perform some **numerical stability**. One possible approach is considering the **sum of log probabilities** so instead of multiplying probabilities, we multiply their logarithms. The class with the highest final un-normalized log probability score is still the most probable:

$$c_{NB} = \underset{c_j \in C}{\operatorname{argmax}} \ \log P(c_j) + \sum_{i \in positions} \log P(x_i|c_j)$$

where c_J is the class prior. So we get the summation of the logs of all features plus the log of all priors.
So, if we want to classify whether an image contains a 5 or a 6, instead of comparing $P(Y=5|X_1,...,X_n)$ with $P(Y=6|X_1,...,X_n)$ we compare their logarithms:

$$\log(P(Y|X_1,\ldots,X_n)) = \log\left(\frac{P(X_1,\ldots,X_n|Y)\cdot P(Y)}{P(X_1,\ldots,X_n)}\right) = \text{constant} + \log\left(\prod_{i=1}^{n} P(X_i|Y)\right) + \log P(Y) = \text{constant} + \sum_{i=1}^{n} \log P(X_i|Y) + \log P(Y)$$

10. Bias/Variance and Regularization

10.1 Bias/Variance

Let's assume we have a classification problem so we have to fit a line so that we predict true or false. If we just fit a line (so our function is $h(x)=\Theta_0+\Theta_1 x$) the problem is that we may **underfit** so, a model that **performs badly both in training set and the testing set**. One solution may be add a polynomial term ($h(x)=\Theta_0+\Theta_1 x+\Theta_2 x^2$) in order to have a more complex classification. Notice that if we add too many multinomial terms we may **overfit**, so the line is way too complex so that the **training set performs quite good but the testing set has really low performance**. In the case of underfitting we have **high bias** while in the case of overfitting we have **high variance**. In detail:

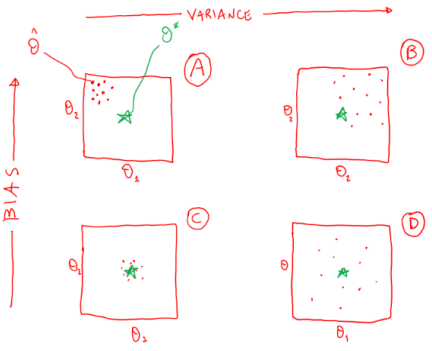
- The **bias** error is an error from erroneous assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting). In the example before, the model is so simple that if we give it more data, it won't consider it.
- The **variance** is an error from sensitivity to small fluctuations in the training set. High variance may result from an algorithm modeling the random noise in the training data (overfitting). In the example before, the model is too complex in fact if we add more data, the model will consider it all losing generalization power.

The **bias-variance tradeoff** is a central problem in supervised learning. Ideally, one wants to choose a model that both accurately captures the regularities in its training data, but also generalizes well to unseen data. Unfortunately, it is typically impossible to do both simultaneously.

To introduce the topic, let's first make some assumptions:

- We assume a data distribution D where the samples are pairs (x, y) are well distributed for the train and for the test.
- The samples are independent.

We have a process of learning that considers the set S of training samples (x^i, y^i) distributed according to our data distribution. We use the learning algorithm (also called estimator) and estimate the optimal hypothesis $\hat{h}$ or, equivalently, $\hat{\theta}$ (notice that we are going to represent the true parameters with Θ* or h*). S is a random variable (each sample is independent), the learning algorithm is a deterministic function (given a certain data, we end up with a set of parameters when convergence occurs) and the output is a random variable as well (it depends on the set of data which is random). We call this distribution the **sampling distribution**.



So far, we have used a data view (a scatterplot with the line fitted by model between the data) but, to better understand bias/variance we have to introduce the **parameter view**. Let's assume we have an algorithm A, we sample m points and we estimate the parameters according to those points. We can do the same for the case of algorithm B and we get another estimate. The same for C and D. So if the parameters we have estimated are centered around the true parameters, we have a low bias: the more we are off with respect to the true parameters, the

more bias we have. If the estimated parameters are dense the variance is low: the more sparse the more variance we have.

So, if we have a high bias but a low variance (A), it means the model is so simple that it won't vary a lot if we change the set of points we have because we have zero variation: notice that the estimation is wrong because we are still far from the true parameters. In the opposite case (D) we have a large variance but a small bias, so depending on the set of samples we have the solution would change a lot due to the complex model (but maybe overall the average might be closer to the true parameters with respect to the previous case).

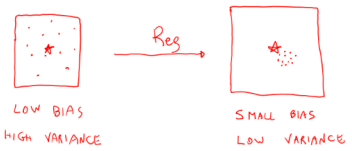
Now, let's assume we have a set of data. If we increase the data it won't change anything in the case with a large bias since we would still be far from the true parameters but we can **decrease the variance**. In the extreme case, we can assume to add an infinite number of samples and if that's the case then the variance tends to 0.

$m \to \infty \qquad \mathrm{VAR}[\hat{\theta}] \to 0$ "STATISTICAL EFFICIENCY": RATE $\mathrm{VAR}[\hat{\theta}] \to 0$ AS $m \to \infty$

The algorithm is called **consistent** when:

$\hat{\theta} \to \theta^* \qquad m \to \infty$

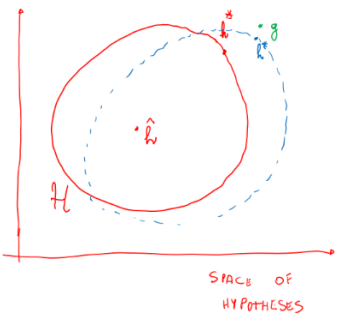
If all the estimated parameters are exactly the true ones (so $E[\hat{\theta}]=\Theta^*$) for all m then the algorithm is called **unbiased**.



So, two possible ways of **fighting high variance** are:

1. Let **m**→∞ (what we have seen so far);
2. **Regularization** (it can allow us to go from a low bias and high variance to a small bias (probably a little larger than before) and low variance).

Now, let's try to generalize: let's assume we have g that is the best possible hypothesis. H is the set of all possible hypotheses we can get to. Given a limited amount of training data, $\hat{h}$ is our hypothesis. h* is the hypothesis that is the best estimate we can get in H (the closest to g but still in H). Notice that if we increase the bias, we reduce the variability so the H circle will shrink (it won't necessarily move toward g, it will simply shrink). We can define:



$$\varepsilon(h) = E_{(x,y)\sim D}\left[\mathbb{1}\{h(x) \neq y\}\right] \quad \text{RISK / GENERALIZATION ERROR}$$

$$\hat{\varepsilon}_s(h) = \frac{1}{m}\sum_{i=1}^{m} \mathbb{1}\{h(x^{(i)}) \neq y^{(i)}\} \quad \text{EMPIRICAL RISK}$$

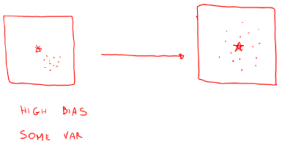
The **risk** (or generalization error) is what we get with an infinite amount of data while the **empirical risk** is the error we get on the m samples we have. We also have:

- **ε(g)** = **Bayes/Irreducible Error**: the error we get if we were able to estimate g (the best hypotheses). Remember that even in the best case, the error will never be 0.
- **ε(h*)-ε(g)** = **Approximation Error**: the error between the best hypotheses possible and the best hypotheses we can get within the class of ML model we are considering. This is due to the choice of the class.
- **ε(ĥ)-ε(h*)** = **Estimation Error**: the error between the best hypotheses we can get within the class of ML model we are considering and the one we actually get. This is due to the choice within the model (e.g. polynomial degree within the logistic regression) and the amount of data.
- **ε(ĥ)**: Estimation Error + Approximation Error + Irreducible Error

The reason for the estimation error are the choice of the estimation model (e.g. too low or too complex polynomial degree) and the amount of data, so it can be decomposed into the **Estimation Variance** and the **Estimation Bias**. The Estimation Variance is the variance that we know. The Estimation Bias and the Approximation Error give us the Bias. So finally we can write **ε(ĥ)** as:

- **ε(ĥ)**: Variance + Bias + Irreducible Error

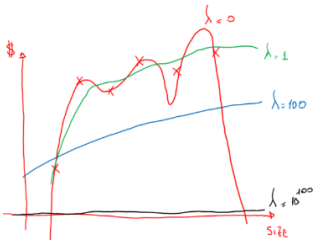
A way of **fighting high bias** is to **make H bigger**. So if we increase the complexity of the model (e.g. the degree of the polynomial), we expect to have something that is more dispersed and hopefully closer to h*.



10.2 Regularization

Let's assume we have linear regression model and we want to **minimize the discrepancy** between $y^{(i)}$ and $\Theta^T x^{(i)}$. If we include some **regularization**, it constrains/regularizes or shrinks the coefficient estimates towards zero. In other words, this technique allows us to use a complex model but constrains the complexity we could get, so as to avoid the risk of overfitting.

$$\min_{\Theta} \frac{1}{2} \sum_{i=1}^{m} \| y^{(i)} - \Theta^T x^{(i)} \|^2 + \frac{\lambda}{2} \|\Theta\|^2$$



Where λ is a hyperparameter that controls how much we want to regularize the model. Regularization significantly reduces the variance of the model, without substantial increase in its bias. Till a point, this increase in λ is beneficial as it is only reducing the variance (hence avoiding overfitting), without losing any important properties in the data. But after a certain value, the model starts losing important properties, giving rise to bias in the model and thus underfitting. Therefore, the value of λ should be carefully selected.
For can also apply it to the likelihood estimation and in this case since we want to maximize it, we can add a subtraction because the larger Θ, the further we are from the maximum:

$$\text{ARGMAX}_{\Theta} \sum_{i=1}^{m} \log P(y^{(i)} | x^{(i)}; \Theta) - \lambda \|\Theta\|^2 \qquad -\frac{\lambda}{2}\|\Theta\|^2 \rightsquigarrow -\frac{\lambda}{2}\sum_{j=1}^{m} \Theta_j^2$$

Now, the norm of Θ^2 means all the components of Θ squared summed up so (notice there is **no regularization for j=0**, generally **we don't regularize the bias**), we can rewrite it as:

$$\ell(\Theta) = \sum_{i=1}^{m} y^{(i)} \log h_\Theta(x^{(i)}) + (1 - y^{(i)}) \log\left(1 - h_\Theta(x^{(i)})\right) - \frac{\lambda}{2}\sum_{j=1}^{m} \Theta_j^2$$

So, gradient ascent becomes:

$$\Theta_j := \Theta_j + \alpha \left[\sum_{i=1}^{m} \left(y^{(i)} - h_\Theta(x^{(i)})\right) x_j^{(i)} - \lambda \Theta_j \right] \qquad j \in [1, 2, \dots, n]$$

For j=0 we fall into the normal case (so regularization isn't added because of the $x_0^{(i)}=1$). Remember that for Logistic Regression $h_\Theta(x^{(i)})=g(\Theta^T x^{(i)})$ because of the sigmoid function.

Now, let's see the **probabilistic interpretation of Regularization** (similarly as we did for the Logistic Regression). Let's assume we have S which is a set of m examples of the shape $(x^{(i)}, y^{(i)})$. Our goal is to get the most likely set of parameters Θ given the training set S, that, using the Bayes rule is:

$$P(\Theta | S) = \frac{P(S|\Theta)\,P(\Theta)}{P(S)}$$

Leaving out the denominator (since it doesn't depend on Θ), what we want to get it the argmax of the numerator that is (notice that we plug in Logistic Regression to represent P(S|Θ)):

$$\text{ARGMAX}_{\Theta} P(\Theta | S) = \text{ARGMAX}_{\Theta} P(S|\Theta)\,P(\Theta) = \text{ARGMAX}_{\Theta} \left(\prod_{i=1}^{m} P(y^{(i)} | x^{(i)}; \Theta) \right) P(\Theta)$$

If we assume the prior to be gaussian distributed with mean 0 and a certain variance $\tau^2 I$, then P(Θ) can be written as:

$$P(\Theta) = \frac{1}{(2\pi)^{n/2} |\tau^2 I|^{1/2}} \exp\left(-\frac{1}{2}\Theta^T (\tau^2 I)^{-1} \Theta\right) \qquad P(\Theta): \Theta \sim \mathcal{N}(0, \tau^2 I)$$

If we plug P(Θ) into the argmax above then we fall into the above formulation of regularization.

So, what we have done so far (without regularization) was a **frequentist approach** that is applying MLE:

$$\text{FREQUENTIST}: \quad \text{ARGMAX}_{\Theta} P(S|\Theta) \quad \text{MLE}$$

With regularization we get a **bayesian approach** so we get a prior distribution P(Θ) and we maximize P(Θ|S) and this is called **MAP** (**Maximum a Posteriori**) (so we get another element that is the prior):

$$\text{BAYESIAN}: \quad \text{PRIOR DISTN. } P(\Theta) \quad \text{ARGMAX}_{\Theta} P(\Theta|S) \quad \text{MAP} \quad \text{MAXIMUM A POSTERIORI}$$

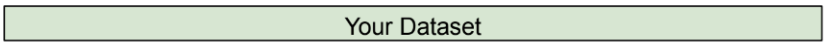
10.3 Hold-out Cross Validation

Remember that if the model we choose is too simple, we will probably experience overfitting, so we need to make it more complex (e.g. by using polynomial features) but if it becomes too complex we will probably experience overfitting. With overfitting the training error is lower but what we care about is actually the testing set. The same applies to regularization. So we have to split the dataset so that we can train the model, choose the right hyperparameters and see how they perform, and finally test the system we have created. In general:

1. Split S into S_{train}, S_{dev}, S_{test}.
2. Train each model S_{train}.
3. Choose the model with lowest error on S_{dev}.
4. Optional: evaluate on S_{test} and report performance.

We have some approaches:

Idea #1: Choose hyperparameters that work best on the data — **BAD**: would always choose most complex model

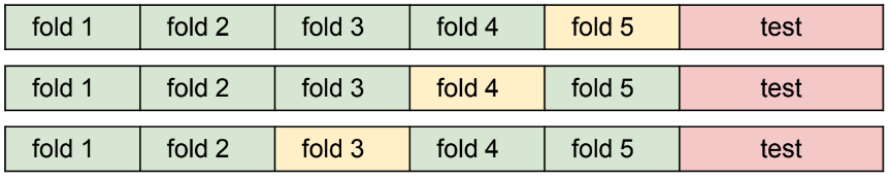


Idea #2: Split data into **train** and **test**, choose hyperparameters that work best on test data — **BAD**: No idea how algorithm will perform on new data

Idea #3 (**hold-out cross validation**): Split data into **train**, **val**, and **test**; choose hyperparameters on val and evaluate on test — **Better!**

Another approach is the **K-Fold Cross Validation**:

Idea #4: **k-fold cross-validation**: Split data into **folds**, try each fold as validation and average the results



This is usually used for small datasets but not in deep learning.