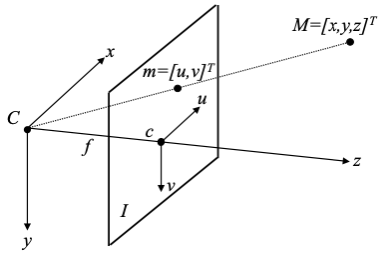
Module 2

Image Formation Model

@April 20, 2023

Images (2D) are formed through Perspective Projection of the 3D world: light rays go through the pinhole to hit sensitive material on the focal plane. We have the mathematical framework that we've discussed during Module 1.

CRF (Camera Reference Frame): we define 3D points $M = [x, y, z]^T$. x and y are parallel to u and v (image axes), the z axis coincides with the optical axis. The optical axis is the line that starts from the optical center (pinhole), goes through the image center (piercing point) and is orthogonal to the image plane. The distance between the optical center and the piercing point is f, the focal length.



$$\begin{cases} u = \frac{f}{z}x \\ v = \frac{f}{z}y \end{cases} \quad (1)$$

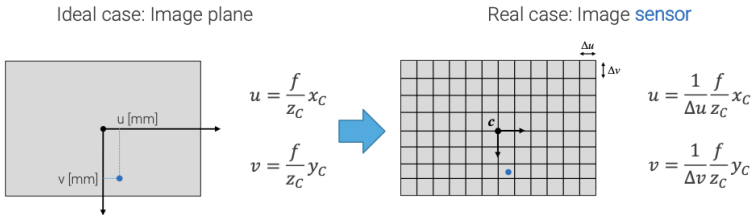
In the model as we've defined it, the focal length is my only parameter. Too simplistic. The objective now is to make it a realistic image plane. We want to address these issues:

1. The image coordinates are usually expressed in an image reference frame with the origin in the top left corner (avoid negative coordinates)
2. Images are not continuous planes, but grids of pixels
3. We don't normally know or want to project back the coordinates of a 3D point in the CRF, but in some generic World Reference Frame.

Complicate the System

We want a World Reference Frame different from the Camera Reference Frame in which we measure things. Our starting point is the 3D point in a WRF we have conveniently defined. We need a roto-translation to move from the WRF to the CRF, where we can then do Perspective Projection and move to a 2D system which is a grid of pixels whose center is the top left corner. I will have two classes of parameters: Extrinsic - not camera-dependent, they depend on the position of the camera in the world - and Intrinsic parameters (depend on the camera system chosen)

Digitization can be accounted for by including into the projection equations the pixel size Δu and Δv along the two axes: pixels are not always squared (need to see the sensor sheets). After this, we have a discrete grid of pixels. How do we move the center to the top-left? Translate the references (with u_0 and v_0 - coordinates of the piercing point - given in pixels).



Normally, we don't estimate $f, \Delta v, \Delta u$ but we reduce them to f_u, f_v (focal length measured in horizontal and vertical pixels - one f two measurement units). This makes 4 intrinsic parameters that represent the camera geometry.

$$\begin{cases} u = \frac{1}{\Delta u}\frac{f}{z_C}x_C + u_0 \\ v = \frac{1}{\Delta v}\frac{f}{z_C}y_C + v_0 \end{cases} \Longrightarrow \begin{cases} u = f_u\frac{x_C}{z_C} + u_0 \\ v = f_v\frac{y_C}{z_C} + v_0 \end{cases}$$

3D coordinates are measured into a WRF external to the camera, which is related to the CRF by a Roto-Translation: which is composed by a rotation around the optical center (3x3 rotation matrix $\mathbf{R}$) and a translation (3x1 vector $\mathbf{t}$). Therefore $\mathbf{M_c} = \mathbf{R}\mathbf{M_w} + \mathbf{t}$.

Note that any valid rotation matrix is an orthonormal matrix, which means that its rows and columns are orthonormal (any two vectors are orthonormal if they are orthogonal and of unit length). For an orthonormal matrix it is true that its inverse is its transpose matrix.

The Extrinsic Parameters (position of the camera w.r.t. the WRF) are 6:

- 3 independent parameters for the translation
- 3 independent parameters for the rotation ⇒ the rotation matrix has actually 9 entries, but the independent parameters - degrees of freedom - are just three (corresponding to the rotation angles around the axes of the Reference Frame)

The optical center in the CRF has coordinates (0,0,0).

The result now is a non-linear model: the mere mathematical formula. The non-linearity comes from the Perspective Projection (the intrinsic model).

$$\begin{cases} u = f_u \frac{x_C}{z_C} + u_0 \\ v = f_v \frac{y_C}{z_C} + v_0 \end{cases} \qquad \begin{bmatrix} x_C \\ y_C \\ z_C \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} x_W \\ y_W \\ z_W \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

Intrinsic camera model — Extrinsic roto-translation

$$\begin{cases} u = f_u \frac{r_{11}x_W + r_{12}y_W + r_{13}z_W + t_1}{r_{31}x_W + r_{32}y_W + r_{33}z_W + t_3} + u_0 \\ v = f_v \frac{r_{21}x_W + r_{22}y_W + r_{23}z_W + t_1}{r_{31}x_W + r_{32}y_W + r_{33}z_W + t_3} + v_0 \end{cases}$$

Projective Space and Homogeneous Coordinates

The standard 2D Euclidean plane can be represented as $\mathbb{R}^2$. In this space, parallel lines do not intersect and you can't represent points at infinity. We can now append a 3^{rd} fictitious coordinate to the Euclidean pairs, so that:

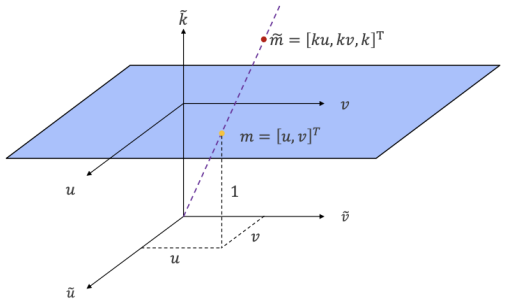
$$m = [u, v] \text{ becomes } \tilde{m} = [u, v, 1]$$

and assume that both vectors are equivalent representations of the same 2D points. Moreover, we do not constrain the new coordinate to be 1, but instead:

$$\tilde{m} \equiv [u, v, 1] \equiv [2u, 2v, 2] \equiv [ku, kv, k] \; \forall k \neq 0$$

In this representation, a point in the plane is represented by an equivalence class of triplets, wherein equivalent triplets differ just by a multiplicative factor. These are called Homogeneous/Projective Coordinates. The space associated is the Projective Space $\mathbb{P}^2$.

We can apply this "ruse" to every n-dimensional Euclidean Space.



Parametric equation of a 2D line: $m = m_0 + \lambda d$. A generic point along the line in projective coordinates $\tilde{m} \equiv \begin{bmatrix} m \\ 1 \end{bmatrix} \equiv \begin{bmatrix} u_0 + \lambda a \\ v_0 + \lambda b \\ 1 \end{bmatrix} \equiv \begin{bmatrix} \frac{u_0}{\lambda} + a \\ \frac{v_0}{\lambda} + b \\ \frac{1}{\lambda} \end{bmatrix}$

By taking the limit $\lambda \to \infty$ we obtain the projective coordinates of the point at infinity of the given line:

$$\tilde{m}_\infty = \lim_{\lambda \to \infty} \tilde{m} \equiv \begin{bmatrix} a \\ b \\ 0 \end{bmatrix}$$


The point at infinity of a line has projective coordinates equal to the direction of the line, but $k = 0$

All the points where $k = 0$, but $(0, 0, 0)$ are valid $\mathbb{P}^2$ points and they form the line at infinity.

We can extend this reasoning also the the points at infinity of a 3D line.


The point at infinity is a point in space, the vanishing point is a point onto the image plane

There's a relationship between the two though: the vanishing point for a set of 3D parallel lines is the projection of their point at infinity. Point at infinity cannot be represented in the 3D Euclidean Space (we would divide by the fourth coordinate - division by zero). With the homogeneous coordinates we can therefore represent and process both ordinary points as well as points at infinity.

NB → the origin of the Euclidean Space $(0,0,0)$ is translated in $\mathbb{P}^3$ as $(0,0,0,k), k \neq 0$.

Now we can express the Perspective Projection in homogeneous coordinates.

$$\tilde{m} \equiv \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \equiv \begin{bmatrix} f_u \frac{x_C}{z_C} + u_0 \\ f_v \frac{y_C}{z_C} + v_0 \\ 1 \end{bmatrix} \equiv \begin{bmatrix} f_u x_C + z_C u_0 \\ f_v y_C + z_C v_0 \\ z_C \end{bmatrix} \equiv \begin{bmatrix} f_u & 0 & u_0 & 0 \\ 0 & f_v & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_C \\ y_C \\ z_C \\ 1 \end{bmatrix} \equiv \mathbf{P}_{\text{int}} \tilde{M}_C$$

Which is now a linear transformation. As a reminder, everything is equal up to an arbitrary scale factor k.

@April 27, 2023

The Intrinsic Matrix is actually made of two parts:

$$\tilde{m} \equiv \begin{bmatrix} f_u & 0 & u_0 & 0 \\ 0 & f_v & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_C \\ y_C \\ z_C \\ 1 \end{bmatrix} \equiv \mathbf{P_{int}} \tilde{M}_C \equiv [\mathbf{A}|\mathbf{0}] \tilde{M}_C$$

- the 3x3 left-most matrix is the Intrinsic Parameter Matrix (usually referred to as **A** or **K**), which models the characteristics of the image sensing device. Its structure is always an upper right triangular matrix. The other 0 in the upper section is modeling the influence that y (vertical world) has onto u (horizontal coordinate of the pixel): skew. Allows to model a non-perfect (non-perpendicular) mounting of the sensor or not-orthogonality of the axis for damage to the silicon of the sensors (rectangles become parallelograms) - usually 0 as manufacturing process become good nowadays, and it's not worth increasing the complexity
- a column (3x1 vector) of zeros

We can write the comprehensive structure as $[\mathbf{A}|\mathbf{0}]$.

The roto-translation from WRF to CRF is already in matrix form from the origin. Though this is not linear, it can be linearized by appending another coordinate to the point: this time becomes a linear matrix, the Extrinsic Parameter Matrix **G**:

$$\tilde{M}_C \equiv \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \tilde{M}_W \equiv \mathbf{G} \tilde{M}_W$$

Now, we have both the transformation from the WRF to the CRF, and from the CRF to the pixel representation.

$$\tilde{m} \equiv \mathbf{P_{int}} \tilde{M}_C \equiv \mathbf{P_{int}} \mathbf{G} \tilde{M}_W \equiv \mathbf{P} \tilde{M}_W$$

By combining everything you can obtain one matrix **P** (which is full rank 3x4 as it is the product of a 3x4 matrix and a 4x4 matrix) and is known as the Perspective Projection Matrix (or PPM). We want a realistic model for image formation so that we can perform calibration.

Every pinhole camera can be represented by the PPM → the most basic is $\mathbf{P} \equiv [\mathbf{I}|\mathbf{0}]$ which is the essence of Perspective Projection. This form, which is defined as canonical or standard PPM allows us to see it as factorized $\mathbf{P} \equiv \mathbf{A}[\mathbf{I}|\mathbf{0}]\mathbf{G}$. (trust the math!)
It shows us the chain of operations:

1. Conversion of coordinates from WRF to CRF
2. Applies the generic parameter-free projection, i.e. divide by the third coordinate
3. Applies sensor specific transformations

Can further derive → $\mathbf{P} \equiv \mathbf{A}[\mathbf{R}|\mathbf{t}]$ (expand the general factorized form from G)

Lens Distortion

Although the pinhole model is a good theoretical model, it is not the real model behind our camera world. So, we add lenses: thin, to make light converge faster to a pixel. The thin lenses we've seen are ideal, while real converging lenses are not perfect and import distortion effects, modeled through additional parameters which do not alter the form of the PPM. This is not due to PP, as it preserves collinearity: it will not preserve parallelism of straight lines, but collinearity.

Therefore we need to add new parameters to the model.

Lens distortion is a very complex phenomenon: we have several variants based on how complicated you want to make it. We see a trade-off: we model lens distortion as a sum of radial distortion (how far are you from the lens center - influences the bending of straight shapes) and tangential distortion (e.g. real lenses are systems of lenses, with small misalignments, impurities…)

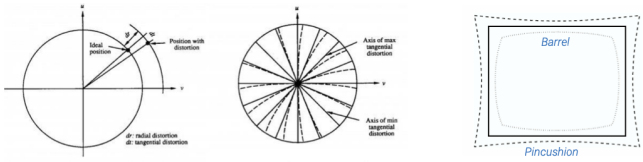
$$\begin{bmatrix} x \\ y \end{bmatrix} = \underbrace{L(r)\begin{bmatrix} x_{undist} \\ y_{undist} \end{bmatrix}}_{\text{Radial distortion}} + \underbrace{\begin{bmatrix} dx(x_{undist}, y_{undist}, r) \\ dy(x_{undist}, y_{undist}, r) \end{bmatrix}}_{\text{Tangential distortion}}$$

x_{undist} e y_{undist} are ideal coordinates; in general, we are still considering metrics (mm, cm…), not pixels.

The radial portion gives rise to one of two effects:

- barrel distortion → straight lines bend outwards (wide-angle lenses e.g. phones)
- pincushion distortion → straight lines bend inward (telephoto, big lenses)

So we need to model a $L(r)$ and tangential distortion. Still, you have a lot of choices.



Radial distortion function is defined only for $r > 0$ and such that at the very center of the lenses you have no distortion (=1). Then, when you move away you approximate to a Taylor Series Expansion up to a certain order. We count only the contributions on the even powers of r.

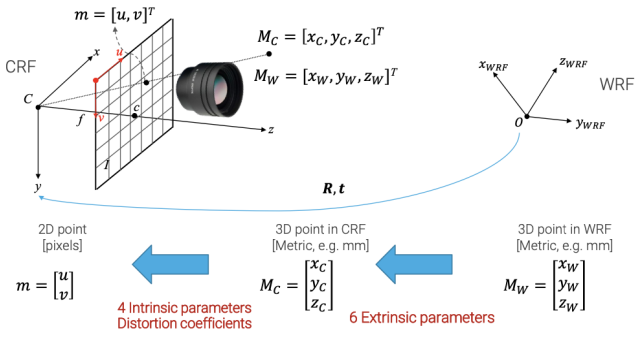
$$L(r) = 1 + k_1 r^2 + k_2 r^4 + k_3 r^6 + ...$$

The tangential distortion comes from the studies.

$$\begin{bmatrix} dx(x_{undist}, y_{undist}, r) \\ dy(x_{undist}, y_{undist}, r) \end{bmatrix} = \begin{bmatrix} 2p_1 x_{undist} y_{undist} + p_2(r^2 + 2x_{undist}^2) \\ 2p_1(r^2 + 2y_{undist}^2) + p_2 y_{undist} x_{undist} \end{bmatrix}$$

We have the radial distortion coefficients k_i and the two parameter weights p_1 and p_2. The ps can be ignored as mounting and production has improved. You use it if results are sub-par.

You use these parameters after applying the canonical PP and before sensor specifics.



Camera Calibration

Camera calibration is the process whereby all parameters defining the camera model are - as accurately as possible - estimated for a specific camera device. Depending on the application, either the PPM or also its independent components $(\mathbf{A}, \mathbf{R}, \mathbf{t})$ need to be estimated.

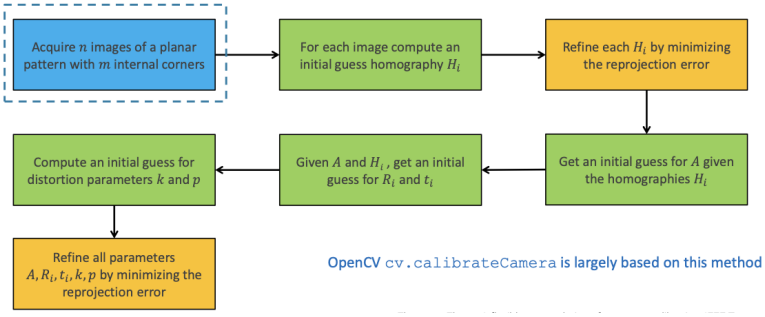
Two main approaches:

- relying on a single image of a 3D calibration object → the calibration is based on one image that features several (at least 2) planes containing a known pattern (simple in practice, but more difficult to find)

- relying on several (at least 3) different images of one given planar pattern (the images are rather easy to obtain).

In general, implementing a Camera Calibration software requires effort. In the main CV Toolboxes, we have something: e.g. in OpenCV we have a nice function `cv.calibrateCamera`. In general, this topic is a good excuse to show 3D-2D relationships

Zhang's Method for Calibration



Easy for 3D, Nightmare for planar targets.
Green Boxes = minimize algebraic error. Orange Boxes = minimize reprojection error

You need to acquire n images of a planar target with m internal corners, of which we know the square size. To obtain the acquisitions you move the pattern around, change its orientation etc. to improve the calibration stability.

Not all chessboards are created equal (as far as rotations are concerned) → you need odd number of corners on one dimension and even along the other to remove rotation ambiguities and the size of the squares, which are the metrics you will base the calibration on.

You also can choose the definition of the World Reference Frame in a specific position. In an unambiguous pattern:

- x axis is aligned with the shortest side of the chessboard
- y with the longest side
- the origin will be next to one of the internal corners (given whatever rules you want).
- z will be orthogonal to the plane formed by the x and y axes

Aware of the size of the squares, I can give a "precise" coordinate to every point of the chessboard; I can define coherent world coordinates that don't depend on the image and are independent of the single image.

I can now use Harris Corner Estimation to detect my internal corners.

Note that the PPM will be different for each image (while the intrinsic parameters reasonably stay the same - you still use the same camera - the extrinsic parameters change between images), as the WRF is attached to the calibration target, whose position is different for each image.

NB → So far we neglect lenses, we assume our camera is a pinhole model.

We said that the PPM is a 3x4 matrix, whose twelve numbers I don't know. First observation I can use: all the points will be on the pattern (I'm observing a planar scene); therefore I have $z = 0$ and I can make the matrix a 3×3. We call this new matrix Homography ($\mathbf{H}$), which is a general transformation between projective planes.

We have to find 9 numbers up to a certain scale.

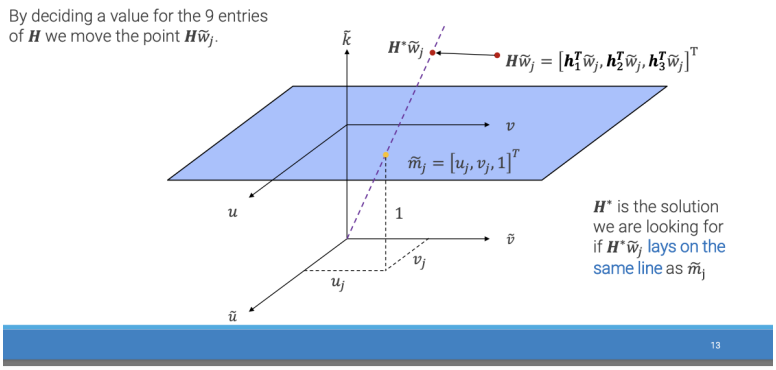
Given the image of a pattern with m corner, we can write three linear equations for each corner j where:

- 3D coordinates are known due to the WRF definition
- 2D coordinates are known due to the corners having been detected in the i-th image
- the only unknowns are the nine elements in H_i

$$\tilde{m}_j \equiv \begin{bmatrix} u_j \\ v_j \\ 1 \end{bmatrix} \equiv \begin{bmatrix} p_{1,1} & p_{1,2} & p_{1,4} \\ p_{2,1} & p_{2,2} & p_{2,4} \\ p_{3,1} & p_{3,2} & p_{3,4} \end{bmatrix} \begin{bmatrix} x_j \\ y_j \\ 1 \end{bmatrix} \equiv \mathbf{H}\tilde{w}_j \equiv \begin{bmatrix} -\mathbf{h}_1^T \\ -\mathbf{h}_2^T \\ -\mathbf{h}_3^T \end{bmatrix} \tilde{w}_j \equiv \begin{bmatrix} \mathbf{h}_1^T \tilde{w}_j \\ \mathbf{h}_2^T \tilde{w}_j \\ \mathbf{h}_3^T \tilde{w}_j \end{bmatrix}$$

Stacking $3m$ equations for the m corners gives us a system of equations, but how do we solve it in the projective space?

When are two 3D points equivalent in $\mathbb{P}^2$?



Estimating H_i (DLT algorithm)

Two points lay on the same line if their cross product is the zero vector.

$$\tilde{m}_j \equiv H\tilde{w}_j \Rightarrow \tilde{m}_j \times H\tilde{w}_j = 0 \Rightarrow \tilde{m}_j \times H\tilde{w}_j = \begin{bmatrix} u_j \\ v_j \\ 1 \end{bmatrix} \times \begin{bmatrix} h_1^T\tilde{w}_j \\ h_2^T\tilde{w}_j \\ h_3^T\tilde{w}_j \end{bmatrix} = \begin{bmatrix} v_j\, h_3^T\tilde{w}_j - h_2^T\tilde{w}_j \\ h_1^T\tilde{w}_j - u_j\, h_3^T\tilde{w}_j \\ u_j\, h_2^T\tilde{w}_j - v_j\, h_1^T\tilde{w}_j \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$h_i^T\tilde{w}_j = \tilde{w}_j^T h_i$

3x9 matrix

9x1 column vector

$$\begin{bmatrix} 0_{3\times1}^T & -\tilde{w}_j^T & v_j\,\tilde{w}_j^T \\ \tilde{w}_j^T & 0_{3\times1}^T & -u_j\,\tilde{w}_j^T \\ -v_j\,\tilde{w}_j^T & u_j\,\tilde{w}_j^T & 0_{3\times1}^T \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = 0_{3\times1}$$

Only 2 equations are linearly independent (multiply first one by $-u$ and second one by $-v$, then sum them to get the third one)

$$\begin{bmatrix} 0_{3\times1}^T & -\tilde{w}_j^T & v_j\,\tilde{w}_j^T \\ \tilde{w}_j^T & 0_{3\times1}^T & -u_j\,\tilde{w}_j^T \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = 0_{2\times1}$$

2x9 matrix

Given m corners, we can create a homogeneous, overdetermined linear system of equations:

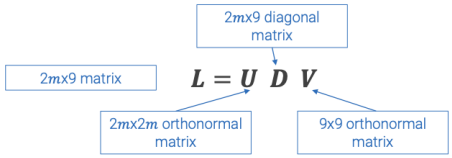
$$\begin{bmatrix} \mathbf{0}_{3\times1}^T & -\tilde{\mathbf{w}}_1^T & v_1\tilde{\mathbf{w}}^{\mathbf{T}}{}_1 \\ \tilde{w}_1^T & \mathbf{0}_{3\times1}^T & -u_1\tilde{\mathbf{w}}_1^T \\ \vdots & \vdots & \vdots \\ \mathbf{0}_{3\times1}^T & -\tilde{\mathbf{w}}_m^T & v_m\tilde{\mathbf{w}}^{\mathbf{T}}{}_m \\ \tilde{w}_m^T & \mathbf{0}_{3\times1}^T & -u_m\tilde{\mathbf{w}}_m^T \end{bmatrix} \begin{bmatrix} \mathbf{h_1} \\ \mathbf{h_2} \\ \mathbf{h_3} \end{bmatrix} = \mathbf{0}_{2m\times1} \implies \mathbf{Lh} = \mathbf{0}$$

To avoid the trivial solution ($\mathbf{h} = \mathbf{0}$) we look for solutions that have an additional constraint (e.g. $\|\mathbf{h}\| = 1$)

The solution $\mathbf{h}^*$ is obtained by the Least Squares Problem:

$$\mathbf{h}^* = \underset{\mathbf{h}\in\mathbb{R}^9}{\arg\min} \|\mathbf{Lh}\| \; s.t. \; \|\mathbf{h}\| = 1$$

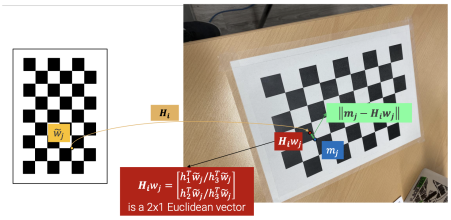
The solution to such problem can be found via Singular Value Decomposition of $\mathbf{L}$. In particular $\mathbf{h}^*$ is the last column of $\mathbf{V}$, $\mathbf{v}_9$.



@May 4, 2023

Now we're stuck on the Least Squares, which means that our solution could not be in the line. We don't have any control on how the error looks like (how big it is and how big it is for the different points). By minimizing the system of equations you minimize the algebraic error in the Least Squares sense, but in geometric terms, you get different errors for different corners (counterintuitive outcome). Plus, we're assuming the absence of lens distortion (which is there!).

Now, we're reasoning on the Euclidean space. The $H_i w_j$ is actually impossible, its a stand-in for a 2×1 Euclidean vector (this gets out of projective space) = 2D vector. I have also m_j, which is our observation. I can work on minimizing the distance between m_j and $H_i w_j$.



The linear system of equations minimizes the algebraic error. This will minimize the geometric error (also called reprojection error). Non-Linear cost function = use iterative algorithm (e.g. Levenberg-Marquardt)

$$\mathbf{H_i^*} = \arg\min_{H_i} \sum_{j=1}^{m} \|m_j - \mathbf{H_i} w_j\|^2 \;\; i = 1, ..., n$$

Why the optimization procedure is two-step? Non-linear iterative algorithms are highly sensitive to the initial condition. So we want to aim for a good initial guess. Getting the Homography is a two-step optimization process: first you guess a good starting point and then apply the refinement, that minimizes the error we really care about.

Now: the three columns of H are three (first, second and fourth) of the columns of the PPM, but we still miss one column. Then, we also miss the separate components.

To get the third column we need the intrinsic parameters. Now, we got an homography for each image → we know that all the images share the same intrinsics (don't active the AutoFocus = changes focal length). We know: $\mathbf{A}[\mathbf{R_i}|\mathbf{t_i}]$ → different R and t for each target image. k is about the projective coordinates (you've still got equality up to a scale factor k).

$$\mathbf{P_i} \equiv \mathbf{A}[\mathbf{R_i}|\mathbf{t_i}] = \mathbf{A} \begin{bmatrix} r_{i1} & r_{i2} & r_{u3} & t_i \end{bmatrix} \implies \mathbf{H_i} = \begin{bmatrix} h_{i1} & h_{i2} & h_{i3} \end{bmatrix} = \begin{bmatrix} kAr_{i1} & kAr_{i2} & kAt_i \end{bmatrix} \implies \mathbf{kr_i 1} = \mathbf{A}^{-}$$

It seems like we don't know anything. We though know that column vectors of an orthonormal matrix are orthogonal, so we can impose this on r_{i1} and r_{i2} = their dot product should be 0. Now I have obtained something I can actually compute = solvable. Plus, we know that r_{i1} and r_{i2} are of unit length (can't really write =1 because I'd neglect the k, but I can say that their length must be the same = equality between dot products).

And here I have my A and my H_i → I can compute R_i and t_i with the same observation.

I could also compute r_{i3} by vector product of r_{i1} and r_{i2} citing the orthonormality conditions, but the model is not ideal because of how I computed k → I don't have perfect unit length and not perfect orthogonality. Overall, you don't obtain an orthonormal matrix per se, but we can use SVD to obtain the closest orthonormal matrix by substituting $\mathbf{D}$ with $\mathbf{I}$ (the identity matrix) and assemble again.

So far we've ignored lenses. This is the part of the formation model which is different between different CV packages. Zhang (and us) will use the simplest model (OpenCV uses a different method that estimates both radial - 3 coordinates - and tangential distortions - 2 coordinates).

$$\begin{bmatrix} x \\ y \end{bmatrix} = L(r) \begin{bmatrix} x_{undist} \\ y_{undist} \end{bmatrix} = (1 + k_1 r^2 + k_2 r^4) \begin{bmatrix} x_{undist} \\ y_{undist} \end{bmatrix}$$

Solve this chicken-egg problem through an approximation. We need idealized coordinates: we use the PPM we've estimated so far to obtain the idealized coordinates of the undistorted image.

We can transform back pixel coordinates $\begin{bmatrix} u \\ v \end{bmatrix}$ to metric image coordinates $\begin{bmatrix} x \\ y \end{bmatrix}$ thanks to the estimated intrinsic matrix $\boldsymbol{A}$

$$\begin{bmatrix} ku \\ kv \\ k \end{bmatrix} \equiv \boldsymbol{A} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} ku \\ kv \\ k \end{bmatrix} \equiv \begin{bmatrix} f_u x + u_0 \\ f_v y + v_0 \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{u-u_0}{f_u} \\ \frac{v-v_0}{f_v} \end{bmatrix}$$

The same transformation holds between u_{undist}, v_{undist} and x_{undist}, y_{undist}

Then, the distortion equation in pixel coordinates become

$$\begin{bmatrix} \frac{u-u_0}{f_u} \\ \frac{v-v_0}{f_v} \end{bmatrix} = (1 + k_1 r^2 + k_2 r^4) \begin{bmatrix} \frac{u_{undist}-u_0}{f_u} \\ \frac{v_{undist}-v_0}{f_v} \end{bmatrix} \Rightarrow \begin{bmatrix} u - u_0 \\ v - v_0 \end{bmatrix} = (1 + k_1 r^2 + k_2 r^4) \begin{bmatrix} u_{undist} - u_0 \\ v_{undist} - v_0 \end{bmatrix}$$

28

Lens distortion coefficients

$$\begin{bmatrix} u - u_0 \\ v - v_0 \end{bmatrix} = (1 + k_1 r^2 + k_2 r^4) \begin{bmatrix} u_{undist} - u_0 \\ v_{undist} - v_0 \end{bmatrix} \Rightarrow \begin{bmatrix} u - u_0 \\ v - v_0 \end{bmatrix} - \begin{bmatrix} u_{undist} - u_0 \\ v_{undist} - v_0 \end{bmatrix} = (k_1 r^2 + k_2 r^4) \begin{bmatrix} u_{undist} - u_0 \\ v_{undist} - v_0 \end{bmatrix}$$

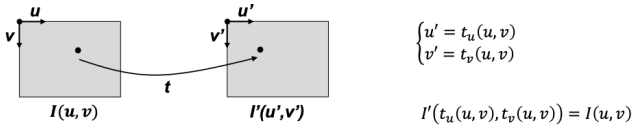
$$\begin{bmatrix} u - u_{undist} \\ v - v_{undist} \end{bmatrix} = \begin{bmatrix} (u_{undist} - u_0) r^2 & (u_{undist} - u_0) r^4 \\ (v_{undist} - v_0) r^2 & (v_{undist} - v_0) r^4 \end{bmatrix} \begin{bmatrix} k_1 \\ k_2 \end{bmatrix}$$

We need a model between pixel coordinates → we use the intrinsics I estimated in the previous step. Now I can write my distortion equations in pixel coordinates. I obtain two constraints in two variables for each corner. I stack all the constraints I have to obtain overall $2mn$ constrains (over-determined). Use Least-Squares Estimation and compute pseudo-inverse matrix.

Everything I have is an initial guess, so I need to minimize again a geometric error, w.r.t. what I have once I get the full model (difference between where it ends and where it should have ended). The residual we obtain is what you get after running the function = quality of the calibration. Below 1, indication of a good calibration.

Very first thing you do → remove lens distortion, as you want to see your model as an ideal pinhole camera. Warping is a transformation of an image = I change pixel coordinates. Filtering acts on pixel value.

Image warping is a simple operation → point-wise transformation. Warping doesn't necessary have to be a lens distortion. It can be a simple rotation or a full homography (undo perspective projection on a planar scene e.g. optical character recognition).



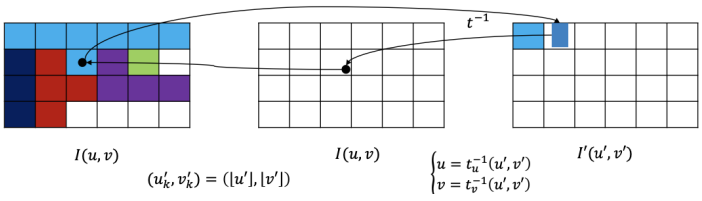
This is an example of Forward Mapping

Problem with Forward Mapping → after mapping integers to rotations, we won't end up in discrete coordinates, but continuous. But I want to go from pixel to pixel. So I need a mapping function: truncation, nearest neighbor (rounding).

Problem: whatever choice of mapping, I have two intrinsic problems in Forward Mapping:

- folds → I can have more than one pixel mapping in the same position (fixable)
- holes → some pixels of the destination image may not be hit because of the roundings

Try backward mapping. We start from the target image (the output). Use the inverse mapping. This doesn't solve the fact that my result will be in a continuous space. But now, when I apply a mapping function I get exactly one pixel. I can claim this problem solves holes because I compute a value for each output pixel (so it's by design). I can do something even smarter than truncation or nearest neighbor. Each pixel can be seen as an interpolation between the four pixels nearby (bilinear interpolation).



Bilinear interpolation → given our non-integer coordinate near these four pixels. I will have Δu and Δv (offsets - how much in the middle between pixels am I). I want to use them as weights. I want to compute $I(\Delta u, \Delta v)$ as the linear interpolation of two values I don't have I_a and I_b, which I have to compute. How? I perform linear interpolation between I_1 and I_2. I can write a proportion based on similar triangles to obtain I_a and I_b:

$$\frac{I_a - I_1}{\Delta u} = I_2 - I_1$$

I can apply then the same formula to obtain $I(\Delta u, \Delta v)$.

Final result:

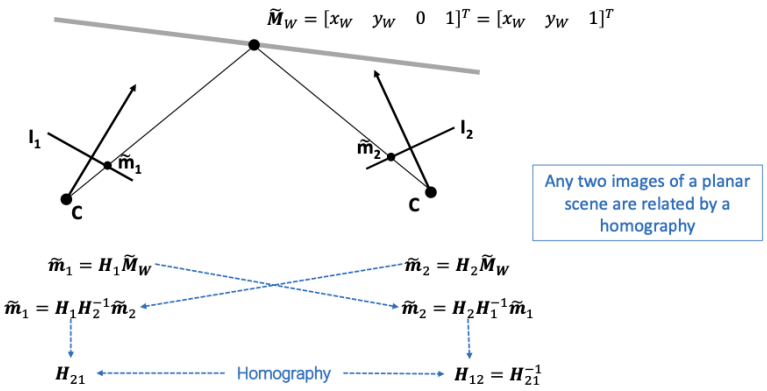
$$I(u', v') = (1 - \Delta u)(1 - \Delta v)I_1 + \Delta u(1 - \Delta v)I_2 + (1 - \Delta u)\Delta v I_3 + \Delta u \Delta v I_4$$

The closer a point to a pixel, the smaller the other weights become. For general uses, bilinear interpolation is nicer, smoother, but not as sharp as Nearest Neighbor (NN) Interpolation (in case of looking for edges, it is the best option). Plus, NN Interpolation is faster.

How do I undistort then? Backward warping through Bilinear Interpolation. I need the inverse mapping (start from each pixel of the output image and compute its value on the input image). How do I use my lens distortion model? I need a model from the undistorted image coordinates and tells me what to read in the distorted image. Which I have from Zhang's Radial Distortion (compute distorted pixels from the ideal undistorted image).

@May 8, 2023

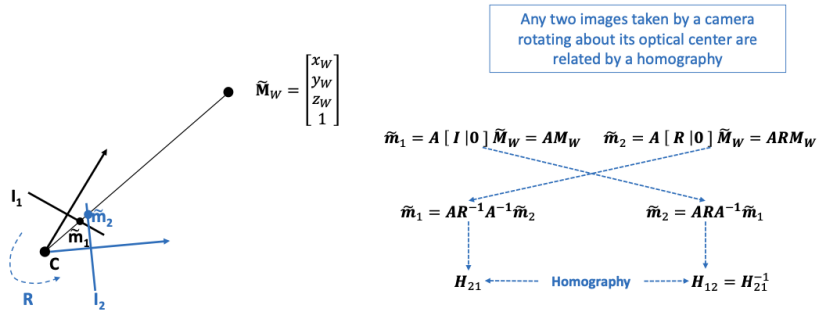
What we have studied for Camera Calibration can be used beyond Camera Calibration: we can show that any two images of a planar scene are related by homographies. For a given image, if we put our world reference frame on the plane $z = 0$, with the usual choice of axes, we get an homography as a relationship. If I add another camera, I still have a relation by homography. I can relate the two images obtained by the cameras.



H_{12} because it goes from pixels from camera 1 to pixels in camera 2. From the expressions, you can deduce that $H_{12} = H_{21}^{-1}$. This shows that if I have pixels of the planar objects in a scene with a camera and I know the homography, I can obtain the same pixels from the second camera.

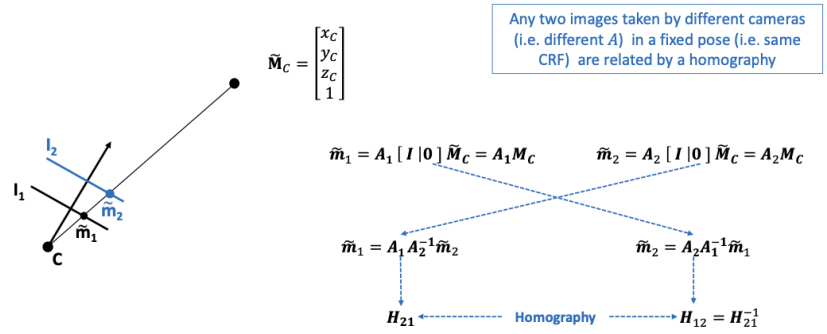
An example is Inverse Perspective Mapping. If you can calibrate your dashcam, camera of the vehicle, w.r.t. the ground plane, then you can compute the homography between this camera and the virtual camera which is looking at the plane exactly from top (top-down view of the whole plane - doesn't really exist, it is like there's a camera floating on top of the plane). I have a way to compute the homography between an image and the image plane of the virtual camera. I can then use backward warping to obtain pixel-by-pixel the image from the virtual camera. Shows me what the world looks like from top: useful for planning, control, etc. It dramatically distorts what's not on the plane.

Two images taken by the same camera which is moved, i.e. is rotated about its optical center C are also related by a homography. If I have a Camera 1 and a Camera 2, the relationship between the two is a rotation, meaning I have a rotation matrix that transforms pixels in the WCF of the first camera into pixels of the WCF of the second. If I write the two PPMs of the cameras, I can show they are related by a homography. Remember that WRF is something we have control on (and must be the same for both the cameras), and we can choose for one (e.g. Camera 1) to be the easiest possible (CRF), meaning I obtain a simplified PPM - I have no extrinsics (actually, they are identities). The other camera will necessarily (RF must be shared) be the Rotated CRF. As there's no translation, I don't need the homogeneous coordinates, so I can reason in terms of the Euclidean M_W. Then I compute M_W from one side and plug it on the other side, following the same operation as before.



Useful on its own: imagine a scenario where you have a dashcam on a camera, which you can't guarantee it is perfectly mounted. If you can estimate that it just rotates and the entity of the rotation, you can obtain to warp the image to the condition it would be were the camera perfectly mounted.

Can also show that two images taken exactly in the same position (same CRF) but with different intrinsics (different cameras) are also related by homographies. The difference between the two images is the focal length, which changes the matrix A.



We can also both rotate the camera about the optical center and change intrinsics. By the same derivation one could show that the two images are still related by a homography.

Why is this useful? To recover depth, I need more than one image, from which I draw correspondences to establish distances. What we would like to have is a constraint on how the cameras are mounted and on the vertical coordinates of objects in an image. This is difficult to obtain physically, but by using homographies, I can act like I have "virtual sensors". Looking for corresponding points is now easier. Rectification for Stereo Images.

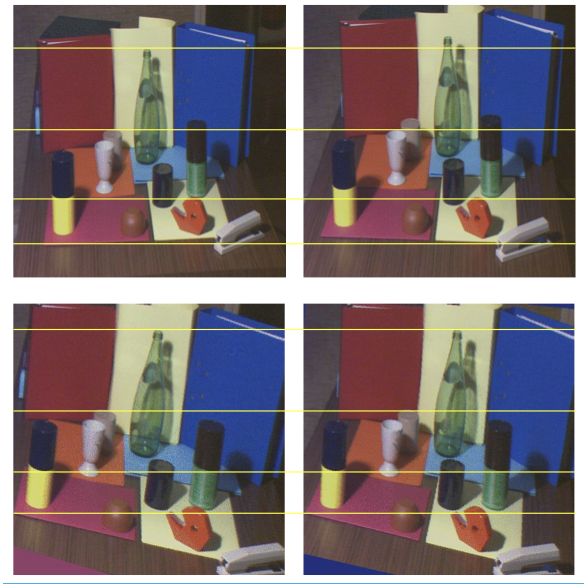


Image Classification

Image Classification is the "Hello World" of Machine Learning for Computer Vision. All the other problems in MLforCV leverage Image Classification. You have some kind of image in input and the task is to choose among some predefined categories which one best represents

the image. Strong assumption: we have a set of predefined categories. As human beings, we can redefine these sets at runtime (our brain is engineered for flexibility). We are constraining ourselves in a predefined setting.

Images, and the world they're sensed in, have a lot of variables: a class has a huge set of differences under the same umbrella term (Intraclass Variations), Background Clutter, Occlusions, Viewpoint variations (weird poses), dramatic Illumination Changes (still open problems). First thing you want to control is light. The world is a weird place, so I can have outlier cases.

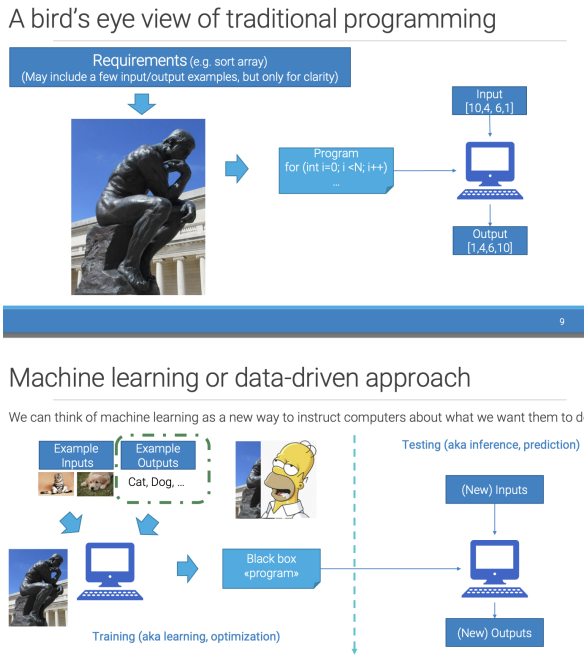
Design what is the input: RGB image → 3 dimensions: height, width and 3 channel dimensions. In Classical Computer Vision, most of the theories work with grayscale images (matrices). In ML we tend to process color images, so we talk about Tensors.

The Output is a class → since the input are numbers, we can process and produce a number which is then compared to a previous class-number mapping (always the same everywhere).

Now, sequence of steps. Classic computer vision could start by computing Edge Detection. But when you start to design rules, you're faced with the variability of the environment. You have very very brittle rules. Industrial Applications of CV are based on Classic Computer Vision → we can work in a highly controlled setting. We though want flexibility, but want a clear boundary on this flexibility.

Data and attaching labels to data is the most important thing to do. Now we will collect images according to what is the variability we have to withstand.

In ML and Data-Driven Approaches, Data is the "star of the show". We are just glorified babysitters for models. Understand the impact of metrics: now we decide what is the metric by which we say "This is good". One of our control tools. Focus switching to Data Preparation (MLOps).



Datasets for Image Classification

- MNIST → 10 Classes, representing handwritten digits from 0 to 9. We have 50K training images and 10K Test images, all 28x28 grayscale images. People still use it, but nowadays almost everything works on MNIST (but not on something else).
- CIFAR 10 → Subset of a larger dataset (Alex Krizhevsky) → collected to have something similar to MNIST, but more challenging. 10 classes, same distribution as MNIST, but the images are 32x32 RGB images. Nowadays, good for learning but easily saturated (as MNIST is)
- CIFAR 100 → another subset of the same dataset with the same structure. Lot less training images per class and a lot of variability. Competing on CIFAR 100 is a more valid result
- ImageNet (the original)/ImageNet 21K → 14 million images, 400x350, which is a more realistic image resolution. The original ImageNet has 21K classes, collected out of WordNet (dataset of words - 50K Concepts arranged in a hierarchy - only the concretes are used). We have algorithms to pre-train models on this.
- ILSVRC/ImageNet 1K→ Competition held from 2009 to 2017 where a subset of images from ImageNet was created with 1.3M training images for 1000 classes. Used for validation. It is optimized for top-5 accuracy. We get 5 answers and we consider the image well classified if it belongs to one of these five categories. Issue: it's an easier problem. E.g. in ImageNet we have 5 classes for turtles, corresponding to 5 different breeds. Recently, multi-label accuracy proposed to overcome this problem
  Vaishaal Shankar et al., Evaluating Machine Accuracy on ImageNet, ICML 2020 → relabeled all of ImageNet to ensure consistency.

When applying Machine Learning methods, we are given/create:

- a training set $D^{train} = \{(x^{(i)}, y^{(i)}) | i = 1, ..., N\}$
- a test set $D^{test} = \{(x^{(i)}, y^{(i)}) | i = 1, ..., M\}$

where $x^{(i)} \in \mathbb{R}^f$ are the features representing the real word items we care about, and $y^{(i)}$ are the outputs we want to predict for that item, i.e. the label in image classification.

Assumption on which ML works: the two sets contain independent and identically distributed (not usually a big deal, unless dealing with time series) samples from the same unknown distribution.

Classifier: k Nearest Neighbor (kNN)

Relies on distances in feature space. Very fast at training time (no training), very slow at test time (this is where the computation happens). I don't have to state anything about the data to assume, as this is a non-parametric instance-based algorithm.

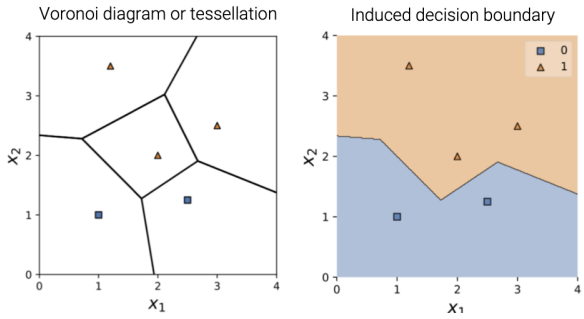
Training phase:

1. Store all the training examples and labels

Test Phase:

1. Compute distance in feature space between test sample and all the training examples
2. Retrieve labels for the closest k training examples
3. Aggregate them (e.g. majority vote) to define the predicted label for the test sample

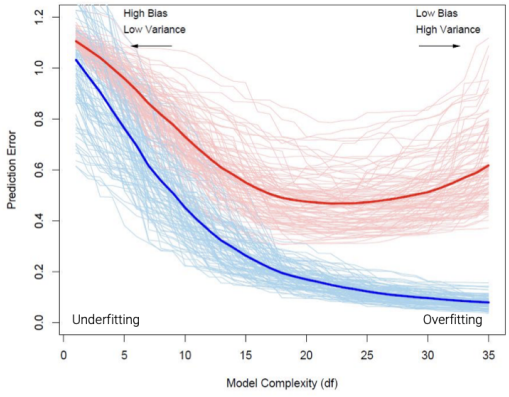
Powerful: being a universal approximator means that the decision boundary (the hypersurface that separate classes) doesn't do any assumption on shapes, so will do any shapes one wants. We have to choose k and how do we compute the distance.



We cannot learn k: if you try to optimize for k, you obtain the global minimum $k = 1$. Parameter-free → you don't have any parameter to learn at training time, but hyperparameters are still there and you set them before you start the training (some of them you decide them, others you validate). We can't learn them as they are too hard to learn or, as k, learning would not yield good results.

When we change hyperparameters we change model capacity → how complex my decision boundaries are.

Trial and Error in ML → Model Selection: we try several values and keep the best. I can use test set, but I get no guarantees for Generalization Error. The need for Validation Set is to solve two issues: hyperparameters (model selection) and generalization error estimation.

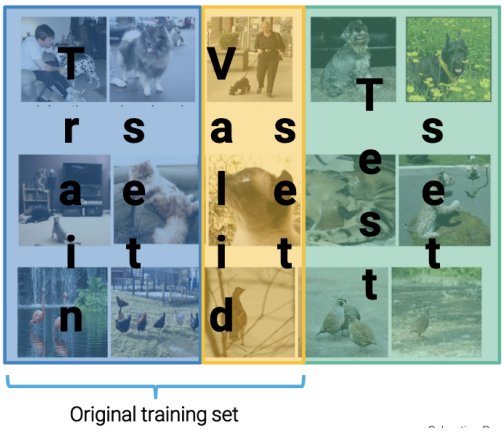


When we vary model complexity, the training error and validation error usually follow this trend

To obtain the best generalization, we want the model to work in the “sweet spot” where the training error is small, but also the gap between the training and validation error is also small.

- Underfitting → training error is large
- Overfitting → the gap between the training and validation error is large

Sebastian Raschka, Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning, arXiv 2018



Expiration Date on Data → at least Validation and Test must be changed once in a while to prevent overfitting when performing multiple rounds of model selection.

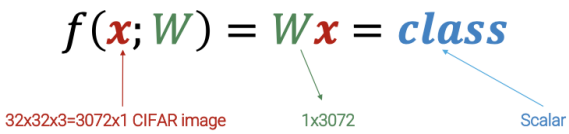
Notation → Channel - Rows - Columns [how PyTorch saves them in memory]

KNN on CIFAR10 → 38% accuracy, which is not too bad for a super simple algorithm. To see why it gets it wrong so many times, look for the neighbors: we see that it looks for closer vectors (it uses raw pixel values as features), so it’s dominated by backgrounds (e.g. blue backgrounds for planes).

The Curse of Dimensionality → kNN claims to be a universal approximator, so it ideally would learn a good function (it can approximate every decision boundary). The problem is that I don’t have “enough data” (should cover my space - point is, it would be unfeasible. We can never collect all). As we increase dimension, the same number of points cover less and less of the space.

@May 15, 2023

So we will never collect enough data to actually learn the boundaries in a precise/generalized way. Flexibility becomes more of a problem. Choose something which is not a universal approximator - every universal approximator takes no decision on the functional form of the decision boundary. Use a parametric approach: we have more inputs as we decide the function we will use in our decision variables. We will learn ideally a good number for θ. The simplest classifier we can learn is the linear classifier. Machine Learning is about subtle, seemingly unimportant decisions, that instead have a huge impact.

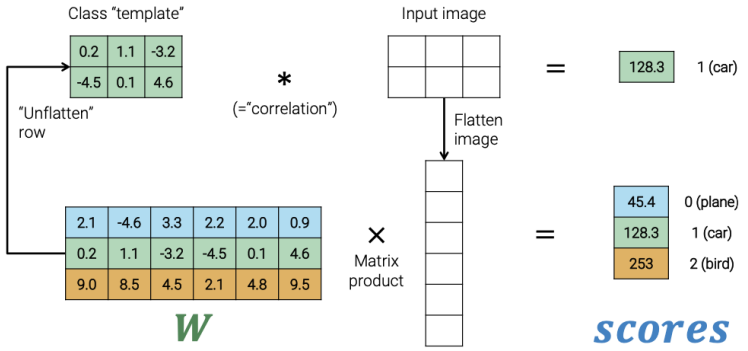


If we’re in CIFAR10, x is our input, which is processed as a 3072×1 vector. I expect my output to be a scalar, and consequently W is a 1×3072. But what are you forcing the model to assume about the problem? Linear Separability of the classes, plus we’re treating it as a regression problem (classes are a continuous end, which is not actually true). The model uses the weights to make this a reality though.

We want to introduce an output of a different and sensible shape (10×1): the model give me an array of numbers (degree of belief) which expresses how much I believe the image represents the correspondent category (e.g. a cat or a plane), aka a score for each class. The array is also known as logits.

You take an input image and you flatten it and multiply a weight matrix and this to obtain the scores. We could also have obtained the same result by “unflattening” the rows of the matrix W and create an image the same size of the input which then means I perform a correlation (I

don’t want to flip) with the original image. High score when the template is in the image. I can see this then as a template matching job.



In ML, linear kinda means always Affine, so I have also a vector of biases, with the same size of the output. Now $\theta = (W, b)$: the one we obtain when we flatten both W and b.

Now, the hypothesis space is the space of all the functions that our model can represent. Learning means solving an optimization problem to find the best function $h \in \mathbb{H}$, so that $h^* = \arg\min_{h \in \mathbb{H}} L(h, D^{train})$. When we have parametric models our h^* is completely defined by its θ. So the optimization problem changes in terms of finding the “best” parameters $\theta \in \Theta$:

$$\theta^* = \arg\min_{\theta \in \Theta} L(\theta, D^{train})$$

What should we minimize?

The number of errors my model makes (another way of saying I improve/maximize my accuracy). This is known as 0-1 Loss in binary problems. It is not actually used, as the error rate does not tell us in which direction we should improve our model.

We introduce an hack → the Loss Function, Objective function, Cost function. Reducing the loss doesn’t always imply the accuracy will improve (this is not the 0-1 Loss anymore): it’s a proxy. Anyway, we expect that if the loss is high, the classifier is performing poorly, and we also expect low accuracy and if the loss is low, the classifier is good, hence we expect high accuracy. So, we prefer values of the parameters that minimize it on the training set. We will work will losses that reduce to sums of the values for the single samples.

$$\theta^* = \arg\min_{\theta \in \Theta} L(\theta, D^{train})$$
$$L(\theta, D^{train}) = \frac{1}{N} \sum_i L(\theta, (x^{(i)}, y^{(i)}))$$

Our first idea is RMSE between the prediction and the training label→ we have iterative algorithms to optimize an L2 Norm. Problems: my truth class is a number and I have a vector → We use one-hot encoding at this point (completely arbitrary decision = we don’t have a real reason why we choose it). You limit the model to perform a decision boundary you’re not really interested in in the end.

We rely on Maximum Likelihood Estimation → we don’t have actual probability density functions though. We use another hack → we make our scores numbers between 0 and 1 and call them probabilities: we use the Softmax Function.

$$p_{model}(Y = j | X = x^{(i)}; \theta) = \text{softmax}_j(s) = \frac{\exp(s_j)}{\sum_{k=1}^{c} \exp(s_k)}$$

Softmax function is more properly a vectorial field ($\mathbb{R}^n \to \mathbb{R}^n$). Is a mapping between vectors. Two effects: I don’t have negative entries and I will emphasize differences with favor to the largest score (it will receive most of the probability mass). Softmax basically decides, throws itself into one class more often than not (should be $\text{soft}\arg\max$ → smooth and differentiable one-hot encoding of the $\arg\max$. Not implemented like this, as we can run into numerical issues due to the $\exp$ function. If though we apply $\text{softmax}_j(s + c)$ we can:

$$\text{softmax}_j(s + c) = \frac{\exp(s_j + c)}{\sum_k \exp(s_k + c)} = \frac{\exp(s_j)\exp(c)}{\sum_k \exp(s_k)\exp(c)} = \frac{\exp(s_j)}{\sum_k \exp(s_k)} = \text{softmax}_j(s)$$

then we can compute it as $\text{softmax}(s - \max_k s_k)$, so we stay in the area (by construction) in which the exponential function behaves.

We can now generate different pdf based on different thetas.

$$\text{softmax}(f(x^{(i)}; \theta)) \doteq p_{model}(Y | X = x^{(i)}; \theta)$$

We want our probability on the training set to be high, my model seen as generator of pdf must yield high probability to the set of labels I actually know are true. The logarithm removes the curse of products, the $-$ is to minimize instead of maximize. To learn θ.

$$\begin{aligned}\theta^* &= \arg\max_\theta p_{model}(y^{(1)}, ..., y^{(N)}|x^{(1)}, ..., x^{(N)}; \theta) \\ &= \arg\max_\theta \prod_{i=1}^{N} p_{model}(Y = y^{(i)}|X = x^{(i)}; \theta) \\ &= \arg\max_\theta \sum_{i=1}^{N} \log p_{model}(Y = y^{(i)}|X = x^{(i)}; \theta) \\ &= \arg\max_\theta \sum_{i=1}^{N} -\log p_{model}(Y = y^{(i)}|X = x^{(i)}; \theta)\end{aligned}$$

Normally you want Cross Entropy to be around .5-ish.

If we put everything together, we want to use a proxy loss that improves accuracy, but we never justified it. If I put softmax and CE together:

$$p_{model}(Y = j|X = x^{(i)}; \theta) = \text{softmax}_j(s) = \frac{\exp(s_j)}{\sum_{k=1}^{C} \exp(s_k)}$$
$$-\log p_{model}(Y = y^{(i)}|X = x^{(i)}; \theta)$$

What I end up with is $-$score of the correct class + the log of the sum of the exps of the scores (called the logsumexp - all the frameworks used have a stable numeric routine to compute it).

$$-\log \frac{\exp(s_{y^{(i)}})}{\sum_{k=1}^{C} \exp(s_k)} = -s_{y^{(i)}} + \log \sum_{k=1}^{C} \exp(s_k) \approx s_{y^{(i)}} + \max_k s_k$$

Approximation of the max function = the exponential function highlights/amplifies small differences, therefore when you sum the only value that counts is basically the highest, while the others are negligible (they've been pushed towards 0 by the $\exp$). By taking this liberty, we enter an approximation of the Cross Entropy Loss: this leads to often higher accuracy, as I penalize the incorrect class.

Implementations in PyTorch:

- `NLLLoss` → wants the log softmax (want the log of the probabilities).
- `CrossEntropyLoss` → inputs are the raw, unnormalized scores for each class (you don't need softmax, it is within the loss)

They should be the same thing, but they are not. If you want to use the model just for classification, you can speed up the matter by using CEL as the argmax is enough for you. The other is for dealing with more complex stuff.

We need an iterative algorithms (as we're in a Non-Linear problem) → Gradient Descent!! Adds important hyper parameters. In Zhang's Algorithm, we spent a good portion of the Linear Section to find good starting points. Now, we start randomly (which is weird somehow, but for now we can't do differently from this) - you basically hope to converge somewhere, results obtained like this will be very sensitive to the initializations; good idea to do some trials and then use an average. We have slight number of assumptions we can leverage. Epochs: how many times you will repeat GD passes (forward pass in which you classify the training data and obtain the loss, backward pass, in which based on the loss, you compute the gradient and then we update the parameters in the opposite direction of the gradient. Now we have two sets of hyper parameters: ones of the models and those of the optimization process. Then the learning rate (how big is your step in the Gradient Descent process).

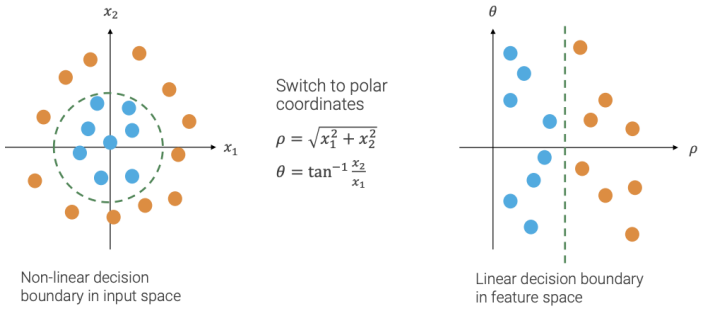
We're now at a certain point in our optimization step of a linear classifier $f(x^{(i)}; \theta) = Wx + b$ and we have our current loss. Now we do another forward pass and obtain a set of scores. Now Gradient Descents computes the gradient of the loss with respect to $\theta = (W, b)$ as big as θ in terms of dimension. The simplest way the gradient is computed is numerically: if I want to compute the gradient of the loss for the first entry w.r.t. b. The gradient tells me what happens to the loss if I move a bit in one direction; so the simplest way is to move a bit along the dimension of the first entry of b (the bias has a direct effect on the score). One computes the new loss and then computes $\frac{\text{newLoss - currentLoss}}{\text{smallStep}}$. Does the sign make sense? The gradient points in the direction of maximum increase in the neighborhood. If this is positive, I should change b to be less. This would decrease the scores, but the loss? The numerator does not change, as it is related to the score of the correct class and we're not taking into account the correct class. We're changing (decreasing) the denominator. So, the ratio increases and the loss decreases because of the $-$ in front (the $\log$ is a monotonic function). Instead, when I increase the score of the correct class, I'm directly modifying the score of the correct class, therefore the ratio will increase and the loss will decrease.

Can compute the gradient analytically (exploiting the rules of calculus, chain rule in particular) or numerically (slow and approximate). The way we're doing this is through Automatic Differentiation (Reverse Automatic Differentiation or Backpropagation).

We can leverage the template matching representation to reshape the rows into templates: we're still using pixels and cannot go farther than 40% accuracy because the background is still dominant.

Image Representations

Working directly with pixels is not enough for the models to perform well. We will have an algorithm to solve the problem, and require a representation to be solved in



I can choose to improve the algorithm or improve the representation

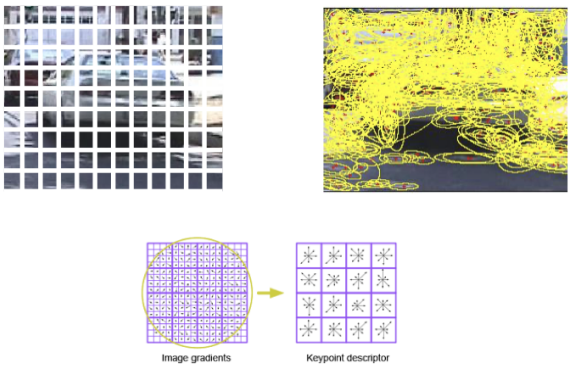
I can then project the classifier in the representation I found back into my original space to obtain more complex decision boundaries.

Dialogue between NLP and Computer Vision (they influence and "steal" from each other): e.g. the Bag of Words representation. Only a small part of the words alone, without any structure (we lose the spatial structure), give me the context. We want to obtain how often a given structure appears in the image. This has been imported in CV. Problem: we work at different levels of richness of data (text is informative and discrete by nature, image is a discretized plane with no clear boundary, so tokenizing is kinda unnatural). A collection is an histogram.

@May 18, 2023

To be able to compute such a histogram, we need to define the codewords (e.g. in NLP, a dictionary is enough to define the x axis). In CV we don't have words, while language is made out of discrete entities which are rich in information; in images you need a lot of pixels to obtain a single concept. And this is something one needs to consider.

The way we define words and discretize an image is visual words → non-overlapping regular patches extracted from the image, you have a regular region, with a lot of repetition and ambiguities. In a sense it is rudimental, so we have also more refined approaches, like taking interest points (using a detector like Harris or DoG, e.g.). Here, though it seems more elegant and less trivial, we lose information, as one would discard background pieces (e.g. cars like to appear on roads) and background provides context. As we will see when we will study Visual Transformers, the regular grid is how we tokenize an image nowadays. This gives us the tokens.

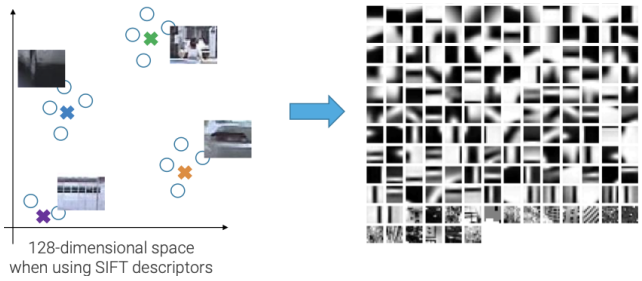


To be more sophisticated one could use grids of several sizes (scales).

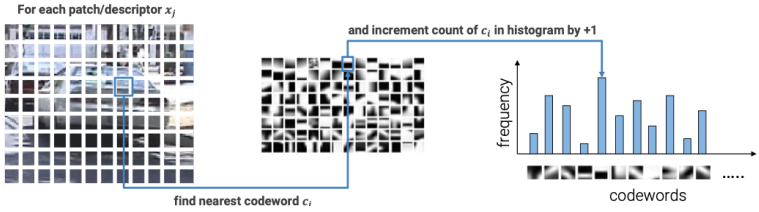
Now how do we represent the words? We can flatten the pixels inside the patches in a 1D vector; compute a 128-SIFT descriptor of the patch (proved to be better) to capture the geometry of the patch. In an attempt to add information (e.g. we work in grayscale), one could also compute a basic color description (but it is not a basic)

How do we create the codebook (our vocabulary)? We cluster our descriptors, usually with the simplest approach possible, which is k-means. The result is clusters of similar SIFT descriptors which will have a centroid each (the mean descriptors) → we will use these as our words. By using patch-space (flattened patches as items we cluster) we can plot the result and see that you can see structures similar to those

seen using modern Convolutional Approaches results obtained on the first layers. So our images speak, but there's no color (modern CNN allows us to use information from the RGB channels effectively).



How do I compute the Histograms: Vanilla Bag of Words → extract each patch (SIFT-Descriptor of the patch) and you search for the closest/most-similar codeword in the codebook, then you increment its bar by one. Already very effective.

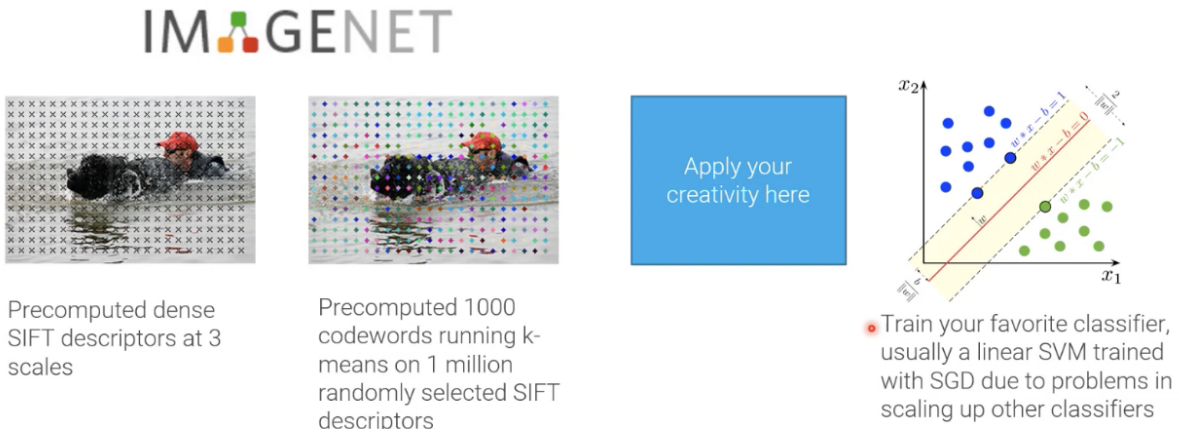


What we do now is obtain a new representation: the features used to represent the image and use them to train the linear classifier, obtaining better results because of more representative info. We obtain a fixed-size representation out of varying input sizes. We also need to appropriately normalize the histogram to make the representation invariant w.r.t. the input sizes.

Nowadays we don't use it anymore. One improvement is VLAD. Now, what Vanilla BoW does is: if c_i is the nearest codeword for a given patch, we add a $+1$ to the respective histogram and that's it. Assume though that you have four patches. Let's assume the nearest codeword is c_i for all of them, but all the patches are not equal, so I'd be losing information by adding $+4$ to c_i. The better thing to do is to not consider them as a hard assignment, as they are all different from c_i in different ways, but consider the position with respect to the codeword. Instead of having just one histogram, you will have a D-dimensional (128 for SIFT) "histograms" - they are more correctly data structures - each with as many entries as there are codewords. If descriptors are D-dimensional and there are K codewords, VLAD creates a $D \times K$ descriptor which stores the differences of input words w.r.t. the codewords, i.e. their relative position w.r.t. the codeword. The "histograms" can now can have also negative entries. VLAD: Vector of Locally Aggregated Descriptors.

From BoW to Deep Learning

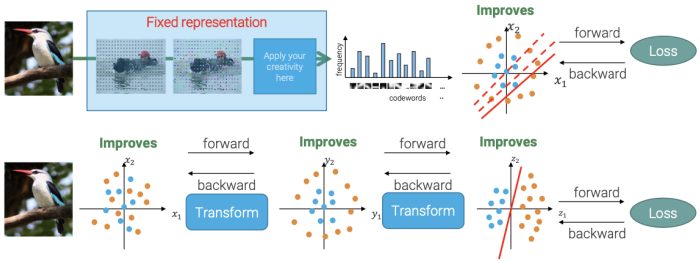
Bag-Of-Visual-Words was the dominant paradigm until 2012, and this was was what ImageNet was originally conceived for and something researchers invested a lot of time.



The winners of ILSVRC11 extracted 10k patches per image and computed SIFT and color descriptors, reduced dimension using PCA (stripped down redundancies), then used an improved variant of VLAD (VLAD++) and a linear classifier in the form of SVM trained with SGD. There was a rudimental form of ensembling, as they used both histograms computed using SIFT and histograms computed using color

and then averaged. Progress was though very stagnating. Then came AlexNet, the revolution, with a completely different approach (Convolutional Neural Network) → in the span of 5 years, the problem was basically solved.

The key innovation of all these methods is pure Machine Learning, when before one had mostly Human Learning through a fixed representation of data in a vector space (Fisher Vectors). All this effort though will conclude in trying to get an hyperplane in the right position (what if data is not linear). Here learning is about representation. Data has little to say here. Deep Learning introduces the idea that you can transform the data to improve it, and at the end of a chain of transformations you can reach a configuration in which you can effectively use a linear classifier.



Different between Shallow Classifiers and Deep Classifiers

The last step will always be a linear classifier; what changes here is that in Deep Learning we also learn the representation. And this is the revolution.

Turns out that you can go very far by stacking linear transformations: so, the final form will be a linear classifier

$$f(\mathbf{x};\theta) = W_2\mathbf{h} + b_2$$

but $\mathbf{h}$ will be computed by another linear layer

$$= W_2\phi(W_1\mathbf{x} + b_1) + b_2$$

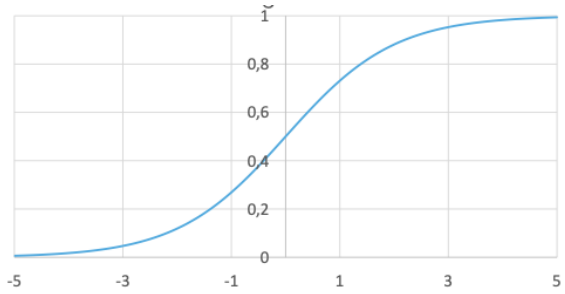
where $\phi(.)$ is an activation function. Now dimensionality is not completely fixed, as we have that b_2 have fixed dimensions (10x1) as it must correspond to the output, W_2 will then have dimensions 10xD and $\mathbf{h}$ must be Dx1 (which depends on the more internal representation. So we have a new (and rather important) hyper parameter. In this new model, θ becomes more complex (as it contains all the hyper parameters). It is finally learning.

Activation Functions

Activation functions are non-linear functions, which are applied to every element of the input tensor (so they don't change the dimensionality):

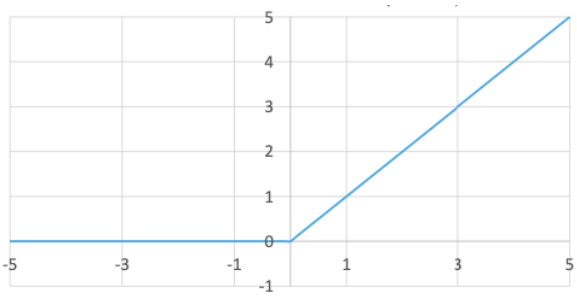
$$f(\mathbf{x};\theta) = W_2\phi(W_1\mathbf{x} + b_1) + b_2$$

The two most common examples are the Sigmoid and the Rectified Linear Unit (ReLU)



$$\phi(a) = \frac{1}{1+\exp(-a)} = \sigma(a)$$

This was the historical choice, as it was considered a smooth, continuous function, good for back propagation as it had a smooth gradient. Actually it was a struggle. Needed a lot of care for



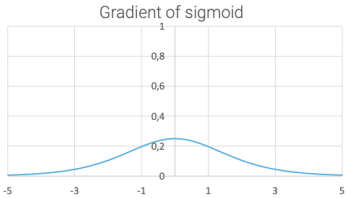
$$\phi(a) = \max(0, a) = ReLU(a)$$

After realizing the problem, the ReLU was proposed (different nonlinearity). ReLU is not differentiable in one point (0), but it doesn't make a real difference. The gradient is the identity on the positive side and 0 otherwise. This makes the recovery from a bad

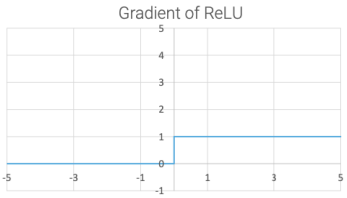
initialization to fit in the linear part of the sigmoid as to avoid the dampening (which slows a lot)

side fast (if you're on the positive side). If you're on the negative side, you don't get any contribution. If this happens a lot, the neurons die.

Gradients play a central role in optimization. The gradient of the output of activation functions with respect to input is very different between sigmoid and ReLU. Yet, ReLU can still give rise to "dead" neurons, if for all inputs the output is negative.

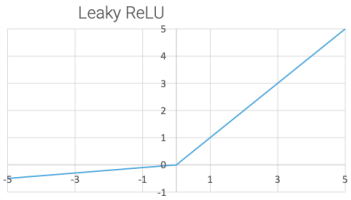


$$\frac{d\,\sigma(a)}{da} = \sigma(a)(1-\sigma(a))$$

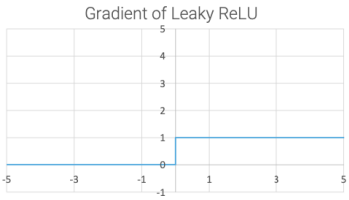


$$\frac{dReLU(a)}{da} = \begin{cases} 1 & if\ a \geq 0 \\ 0 & ow \end{cases}$$

To avoid dead neurons, one could produce a small response also for negative inputs. The simplest variant is called Leaky ReLU, which has a small but non-zero constant gradient also for negative inputs. So, I maintain the nonlinearity, but to avoid killing the neurons, there's a small slope in the negative side. There's now a small gradient (which is dampening), but you don't have dead neurons anymore.



$$Leaky\ ReLU(a) = \begin{cases} a & if\ a \geq 0 \\ 0.01\,a & ow \end{cases}$$

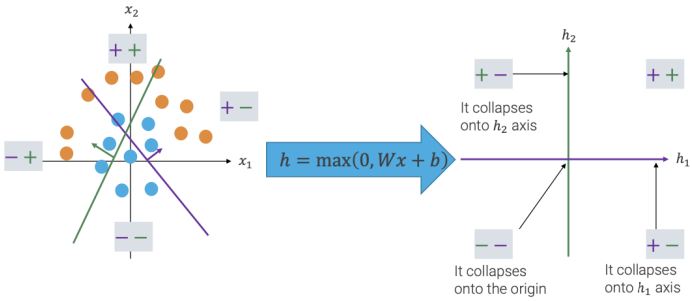


$$\frac{d\ Leaky\ ReLU(a)}{da} = \begin{cases} 1 & if\ a \geq 0 \\ 0.01 & ow \end{cases}$$

ReLU is a baseline, it can improve with some variants. Why couldn't we improve NN even if we had them from the 60s? We didn't have the hardware, but also the architectural switches we had (e.g. from Sigmoid to ReLU).

If activation functions were not there, we would be simply compute a more complicated linear classifier, in which you factorize the weights and biases.

Is ReLU enough, are we really progressing? Consider a bidimensional input space with some points to classify and I can't separate them using linear hyperspaces. What we do is compute new coordinates for the features according to where they stand on the hyperplanes defined by the coordinates of W. Applying a linear layer does not change separability. ReLU essentially takes the point in the 2nd quadrant and collapse them into the h_2 axes. In the 4th quadrant, collapse them into h_1 and 3rd quadrant is collapsed into the origin.



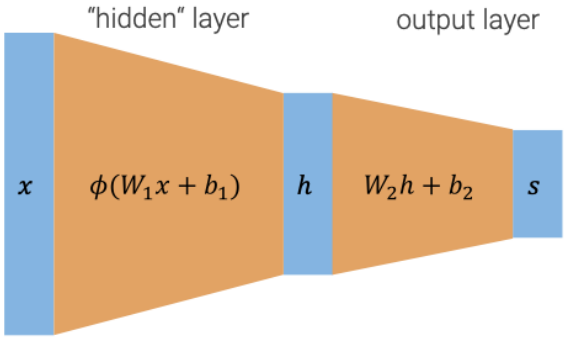
Now one can learn a linear classifier (that usually doesn't separate the points perfectly), but this shows that the function is highly non-linear. If we project, we get a complex decision boundary.

So, by using ReLU, we're able to learn complex decision boundaries.

Deep Networks

$$f(\mathbf{x};\theta) = W_2\mathbf{h} + b_2 = W_2\phi(W_1\mathbf{x} + b_1) + b_2 = \mathbf{s}$$

We call $\mathbf{x}$ our input tensor, $\mathbf{h}, \mathbf{s}$ our activations (outputs of layers - still tensors) and W_i, b_i our parameters (lead us from one activation to another one). Every layer is called (often) a fully-connected layer, as every element of the input influences every element of the output. A neural network with 2 or more layers is also called MLP (Multi-Layer Perceptron). The number of layers is called Depth of the network (if >2, the network is considered Deep). The number of dimensions computed by each layer, i.e. the dimensionality of $\mathbf{h}_i$ is the width of the network. Dimensionality can change between layers.



Why fully connected layers? Just think of matrices and biases. It is a framework inspired by neurons, and simplified. You can think of an activation as the dot product between the rows of W and the input x plus a bias b. You can think of those weights as the weights that the neuron applies to the input x and the activation receives. All the entries of x have influence on all the entries of h (thus, fully connected). b is a sort of threshold to say "fire or not", which is why we introduced ReLU: ReLU is more biologically plausible because it is an all-or-nothing. This is just an analogy.

The original Perceptron was created by Frank Rosenblatt (Leader of the Data-Driven Approach) for the Navy in the 50s: believed in the Machine Learning approach to Artificial Intelligence and investigated that. It was the first successful ML in AI, today we would call it linear classifier. It was a machine, literally (it was actually created physically - changing weights = changing resistances among circuit). The hype and the discussions are still the same, after all.

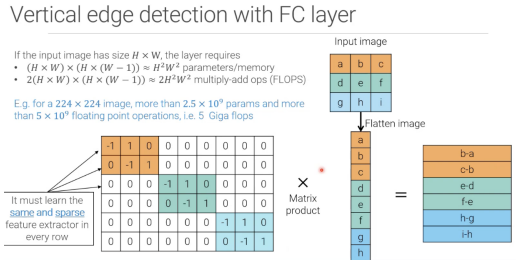
Fight in AI between Symbolic approaches and Data-Driven approaches. Minsky together with Papert (leaders of the Symbolic Approach) wrote a book in the 60s about Perceptrons to shine a light on what they are and what are the limits: Perceptrons can't solve the XOR problem, but also the connectivity problem.

Minsky and Papert also showed also that MLP could solve the XOR problem. Any neural network with two or more layers enjoys some form of universal approximation property: you can in fact approximate any continuous function to any desired precision. But why do we care about depth?

Consider the function "Parity" - extension of XOR to more than two inputs (four): output of 1 if the 1 in the input configuration is odd. If you display it as a Karnaugh Map and group the 1s, they are completely separated and you cannot group it, so you need to recognize each of the 1-gates separately. The number of 1s grows exponentially with the number of inputs, so the number of hidden units grows too much for two layers: we are still subject to the curse of dimensionality. It is shown that deep representations can approximate parity with a number of neurons that grows linearly if you accept to have several layers.

Convolutions

Let's assume that to classify the bird, the very first representation we do is to compute edges, so we flatten the tensor, compute and unflatten h_1 to find edges. What does W_1 look like?. Consider the case of Vertical Edge Detection in a Fully Connected Layer. If my input image has size $H \times W$, I will have a representation that has $(H \times W)$ on one side and $(H \times (W-1))$ on the other (simplest implementation just computes the difference between nearby pixels). I flatten my input image and perform the matrix product.

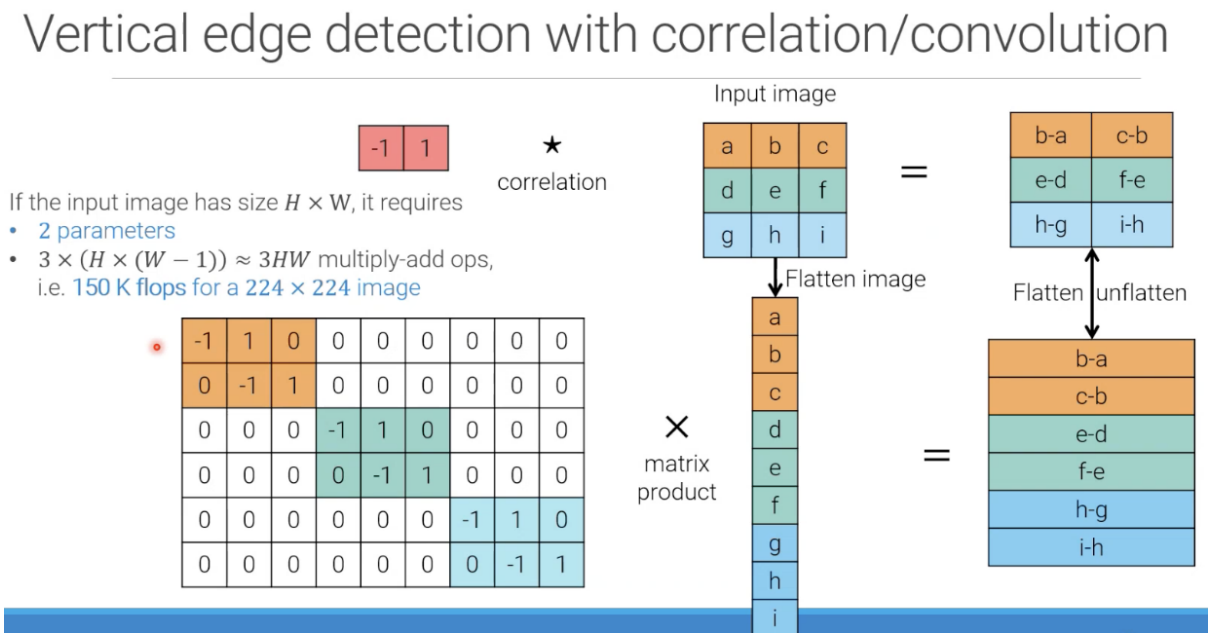


For each entry I have to multiply and sum (FLOPS = floating point operations). For the standard size of ImageNet, the number for the first layer is huge; plus the GD Algorithm needs to retrieve a peculiar structure which is full of zeros (so sparse) and has the same repeating structure. So we want to embed the knowledge that we are processing images into our network.

- Information is local → I can look at a small neighborhood. Technically speaking, pictures have a local receptive field, the rest is redundant
- The structure of the image repeats itself → same operation on all the rows. The convolution should be the same everywhere.

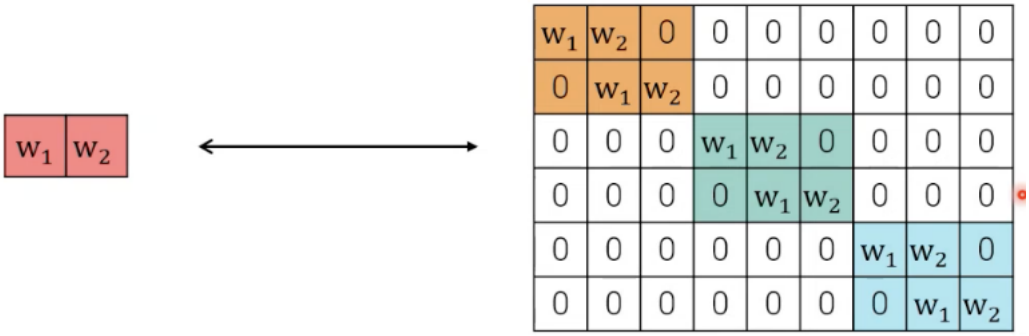
If we can actually process images with convolutions instead of fully-connected layers, we are doing something that is more sensible: Convolutions embody inductive biases dealing with the structure of images: the network has a knowledge about images that comes from the design of the layers. A nice by-product is that we don't need to flatten (practical advantage, we preserve the spatial structure).

In traditional image processing and computer vision, we usually rely on convolution/correlation with hand-crafted filters (kernels) to process images. Unlike linear layers, in a convolution we don't flatten input and output, so we preserve the spatial structure of images. Also, we process only a small set of neighboring pixels at each location → each output unit is connected only to local input units (local receptive field). The parameters associated with the connections between an output unit and its input neighbors are the same for all output units. Thus, parameters are said to be shared and the convolution seamlessly learns the same detector, regardless of the input position. Convolutions embody inductive biases dealing with the structure of images: images exhibit informative local patterns that may appear everywhere across an image.



Now, assuming we act as the previous case, we just need two parameters to learn (the same for every part of the image). And also the number of operations to perform is orders of magnitude less. So we have a lot of practical advantages. The inverse stays: so it is not that the other method is less powerful, this is just more suitable. If you wish, this is less powerful as it is a constrained form of matrix.

Correlation kernel corresponds to a linear matrix in the fully-connected layer but it is peculiar:



Same parameters in every row, and full of 0s.

More constrained form that comes from the idea that it is better for parsing images. But it is the same thing.

Another property that we have from Correlation (Convolution from now on - we don't flip kernels). What we do is Cross-Correlation that we call in another way.

We get another benefit, which is equivariance to translation. To obtain a representation, we can either translate and then apply convolution, or apply convolution and then translate. Why is this important? Form of Data Efficiency: a limit of NN is that we need a lot of prototypes to generalize, and not having equivariance would make it more difficult, as I'd need more data. We're not equivariant to rotation and scale.

Usually, when we switch to Visual Transformers we lose the assumptions on which we base the use of Convolution. So, using images $\neq$ applying Convolution.

Convolutions we use (in Deep Learning) are an extension of the 2D kernel we slide over the grayscale image. Our inputs are either colored input images (tensors) or activations inside the channel. So In general, the convolutions we apply are a generalization to vector-valued functions: we will also have a depth for the kernel and a depth for the image. This is still 2D $\rightarrow$ defined by the dimensions in which you slide (notice we don't slide over the channels).

$$[K * I](j,i) = \sum_{n=1}^{3} \sum_{m} \sum_{l} K_n(m,l) I_n(j-m, i-l) + b$$

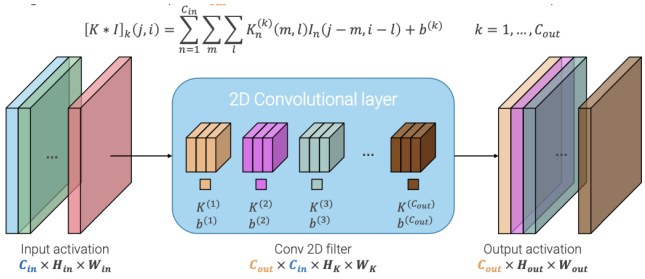
In each location (j,i) I will sum all the contributions of all channels. Whenever I'm at the location and the kernel is there, I will just consider the product of all the entries in the kernel with the corresponding entries in the image and then sum across channels to come out with one single output.

As usual, we compute affine functions, so we have a bias term, which is the same for all locations again. In CNN, we have a hidden dimension (the depth) which is fixed by the dimensionality of the input and is therefore not specified ("5x5 Convolution" = 5x5x3 parameters). By applying this kernel we compute a number for each position.

@May 22, 2023

Even if I compute a 3D tensor with a 3D kernel, which is a 2D convolution I obtain an image, called "feature map". The representation is learned, but is a single panel of values. We want more features (horizontal edges, color transitions…), though. So, we can use more than one kernel, for more flexibility, to obtain a new panel.

The feature maps we have obtained from images, ideally, respect the same constraints. So, it makes sense to compute Convolutions again. We can see a 2D Convolutional layer as a structure with 4 hyperparameters: partial dimension of the kernel H_k, W_k, the depth of the kernel C_{in}, which is fixed depending on the input activation, the number of kernels I have, which is the same as saying how many feature maps I want out, C_{out}.



Convolutional layers are linear layers (particular, constrained, less powerful forms of linearity). We still want to insert non-linear activation functions between each layer.

If you don't do anything else, you will shrink the input activation: due to the kernel sliding I lose the border (padding "valid")

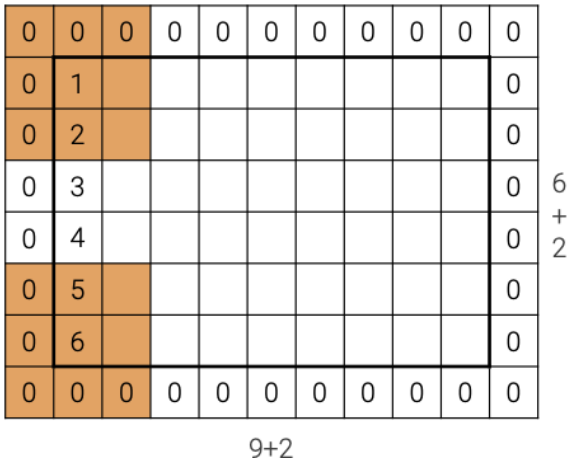
$$H_{out} = H_{in} - H_k + 1$$
$$W_{out} = W_{in} - W_k + 1$$

In Image Classification, it doesn't seem like a big deal, but from a practical point of view, it is easier to think if images don't shrink while stacking Convolutions.

We generally use Zero-Padding: add as many 0s you need around the image so that you consider the whole input while sliding (padding "same").

$$H_{out} = H_{in} - H_k + 1 + 2P$$
$$W_{out} = W_{in} - W_k + 1 + 2P$$

In Image Classification it seems a good choice. When you enter the world of Semantic Segmentation, one can find better results using smarter strategies, e.g. "extending the images" - copying the last value of the column.



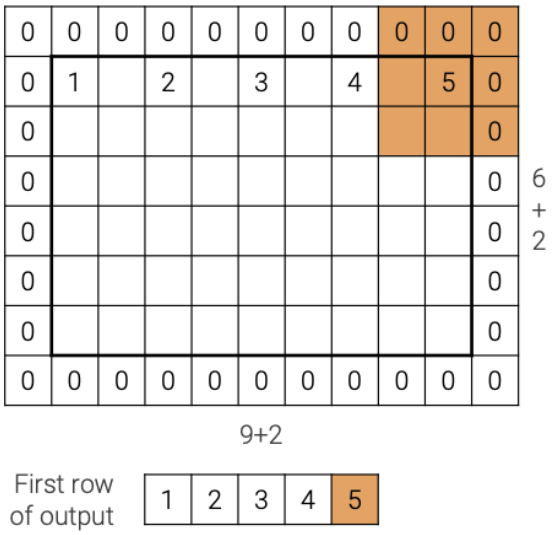
Usually, $P = \frac{H_K - 1}{2}$ to have an output with the same size of the input

Locality needs to be larger the more you proceed inside the network (e.g. locality of a corner or a square is dimensionally different). If I just stack convolutions, the size of the region I'm looking at (the receptive field, i.e. the input pixels affecting a hidden unit) grows linearly.

$$r_L = [1 + L(H_K - 1)] \times [1 + L(W_K - 1)]$$

What's the problem? If I want to be able to detect aggregated local features at later stages, I need a lot of layers (too much). So, the strategy is to down-sample the activations inside the network. The way to perform downsampling in CNNs is by using strided operations → operations that skip pixels (they are applied every n input pixels).

$$H_{out} = \lfloor \frac{H_{in} - H_K + 2P}{S} \rfloor + 1$$
$$W_{out} = \lfloor \frac{W_{in} - W_K + 2P}{S} \rfloor + 1$$



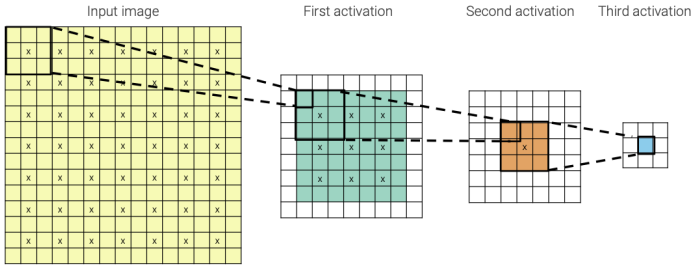
The equations contain a floor to keep the numbers integer.

Side-effect: my activation becomes smaller → convolution is costly, so using strided operations makes it more manageable. The expression for the receptive field becomes more complicated:

$$r_L = [\sum_{l=1}^{L}((H_K - 1)\prod_{i=1}^{l-1} S_i) + 1] \times [\sum_{l=1}^{L}((W_K - 1)\prod_{i=1}^{l-1} S_i) + 1]$$

If all layers have the same stride (constant stride), the product becomes S^{l-1}. This shows that a constant stride will let the size of the receptive field grow exponentially

$$r_L = [\sum_{l=1}^{L}((H_K - 1)S^{l-1}) + 1] \times [\sum_{l=1}^{L}((W_K - 1)S^{l-1}) + 1]$$



Receptive Field with Stride = 2

For the very first layer nothing changes, but e.g. the one before doesn't come from a 5x5 field anymore, but from 7x7, and then 15x15…

So we need strided operation to be able to capture enough concepts without using too many layers.

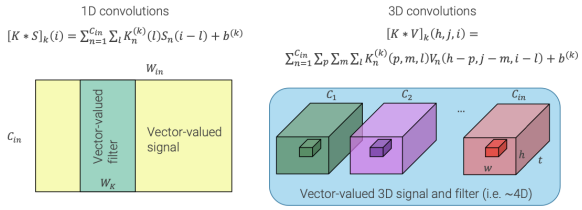
Convolutions allow us to use a smaller number of parameters than linear layers. For a generic convolutional layer, with shape $C_{out} \times C_{in} \times H_K \times W_K$:

$$C_{out}(C_{in}H_kW_k + 1)$$

The $+1$ is for the biases.

We have one order of magnitude less parameters than linear classifiers in general. Once you decide the size of the output using the rules seen for striding, the rest comes easy.

For each of the numbers computed as outputs, a dot product is computed between the number and the corresponding input values. Convolutions, even for mildly sized activations, have a huge number of FLOPS. If GPU support MACs one can halve the number of operations, because can compute the multiplication and sum on the same cycle.



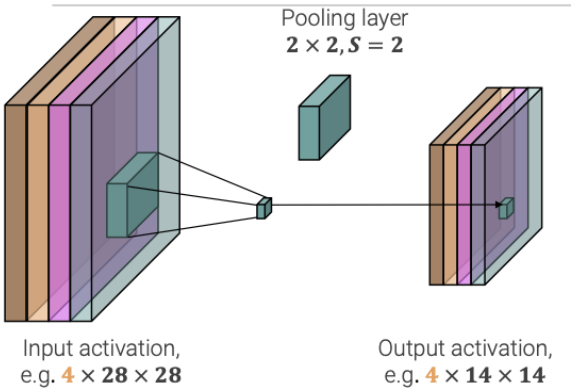
3D convolutions to process 3D signals where you have channels (e.g. videos, in which you have 3 dimensions + the channels).

Layers in CNNs

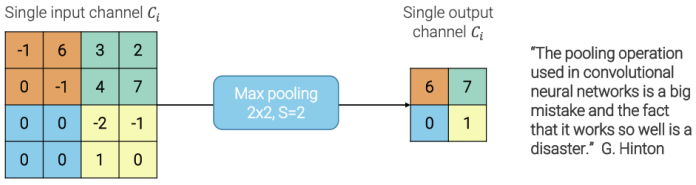
CNNs (or convnets) are a composition of convolutional layers (obviously), non-linearities, fully connected layers, pooling layers and normalization layers.

Pooling Layers

Simpler form of convolution, you have a kernel, you slide it over the activation and compute the output: you don't learn any weights, you specify both the dimension and the function you compute inside the kernel (e.g. average - easier training, max - the most used, it is a non-linear function). Key difference: each channel is processed independently, each kernel has depth = 1: historically, one wants to aggregate feature maps without influencing the depth, but also, one would have just a depth-one feature map (I would compress too much the output channel). Normally pooling downsamples, but it's not mandatory. Downsampling comes from stride, not from the pooling itself.

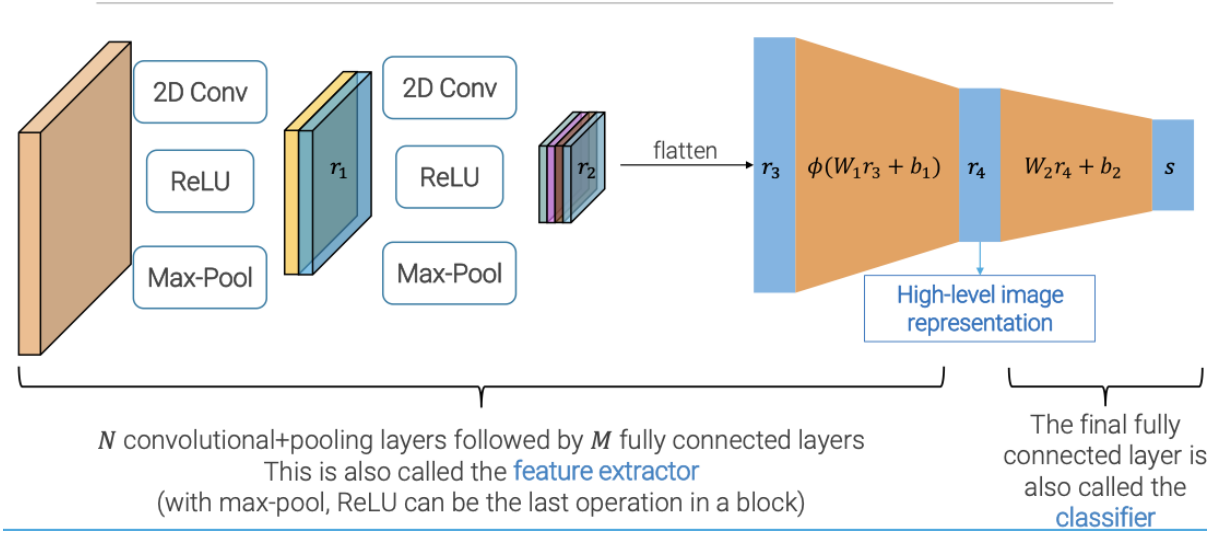


We don't have learning parameters (better for GD, but I need to hope it works); plus, it provides invariance to small spatial shifts (small depends on the amount of stride that precedes it).



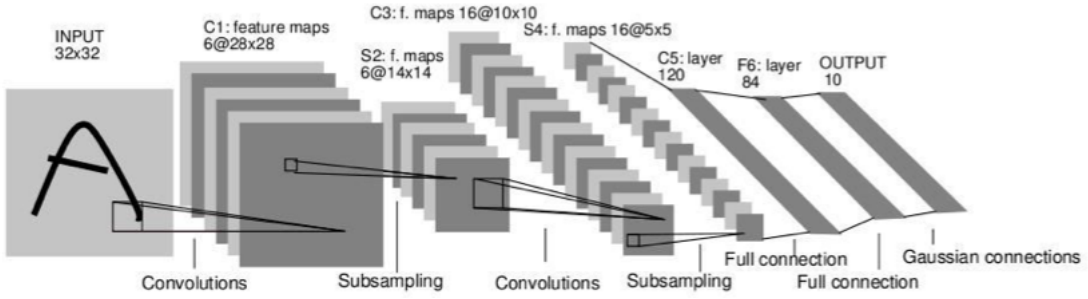
Max Pooling Operation

Plain Vanilla Network



Why do we use fully-connected layers? At a certain point, it is not true anymore that the concept stays the same, so the global arrangement of things is important. Also, image classification is a global task, so I need a global receptive field (although, on a small, heavily-processed version of my image).

LeNet5 was used for checks processing in banks. Why the deep learning explosion came like 15 years later? The vanilla structure is there.



They were sure the data was at the center of the image, so they didn't care about padding. C5 is a linear layer, applied as a convolution that doesn't move.

LeNet5 maintained that as depth increases, the number of channel increases and spatial dimension decreases, uses average pooling and 5x5 convolutional layers with no padding. Chosen non-linearities are the sigmoid or tanh with carefully selected amplitudes. We have multiple fully connected layers and a radial basis function (RBF) classifier. Sparse connection matrix between layers with no batch normalization.

The only thing that is still true today is that increasing the depth, we increase (or at most stay constant) the number of channels and decrease the spatial dimension. Then, if you use pooling at all, you use max. We basically use smaller kernels. Another reason of success is the presence (or not) of Batch Normalization

Batch Normalization

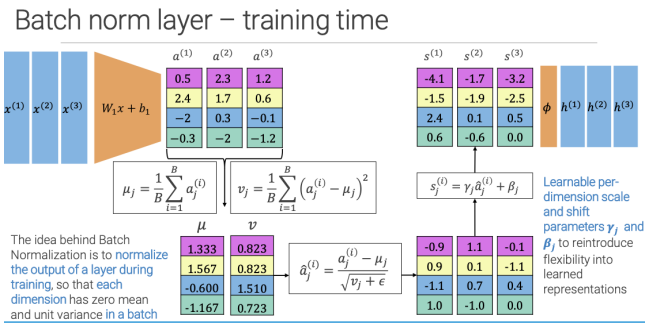
@May 29, 2023

Invented roughly in 2014. We don't have a real understanding of the reasons why we need it, but we use this. The intuition used on the paper was: training Deep Networks was hard, even with all the measures known up to that moment (e.g. ReLU, Initialization schemes,…).

Going beyond 10-11 layers, the performance would crash, hypothetically due to Internal Covariate Shift (ICS). Consider the easiest classifier:

$$f(x;\theta) = W_2 h + b_2$$
$$= W_2 \phi(W_1 x + b_1) + b_2$$

When we train it, seen from the perspective of the most external classifier, we perform some small changes. If we have just one layer, the input doesn't change. When we increase the number of layers, though, we also change h, so we update W_2, b_2 w.r.t. the current h, but when you update them, you modify also h. Lots of compound changes and with more of ten layers, this gets too much. Shift in the distribution of the network activations due to the change in network parameters during training. Covariate Shift already existed in ML.



Based on the intuition, the idea is to constrain. The Batch Norm is a complicated layer, in which you need to be aware of a mini-batch of inputs. At Training time you have a mini batch of inputs B that will be processed all at once. What Batch Norm does is compute the mean and variance for each channel of the output, the mean is computed along the mini-batch dimension. So I'm mixing responses. Normalizes each dimension/channel independently. This constrains the hidden representation. We re-inject some flexibility by adding to each dimension independently some scale and shift learnable-parameters.

In input I have a batch of activations $[a^{(i)}|i = 1, ..., B]$, each with dimension D. 2D learnable parameters γ_j and β_j, with $j = 1, ..., D$. Perform 4 steps:

1. mean across the minibatch → $\mu_j = 1\frac{1}{B}\sum_{i=1}^{B} a_j^{(i)}$ $(1 \times D)$
2. variance → $v_j = \frac{1}{B}\sum_{i=1}^{B}(a_j^{(i)} - \mu_j)^2$ $(1 \times D)$
3. norm → $\hat{a}_j^{(i)} = \frac{a_j^{(i)} - \mu_j}{\sqrt{v_j + \epsilon}}$ $(B \times D)$
4. move around → $s_j^{(i)} = \gamma_j \hat{a}_j^{(i)} + \beta_j$ $(B \times D)$

All the steps are differentiable and you back-propagate through them when you perform GD. This guarantees it all does not undo itself. It might emerge that the parameters reverts to mean and standard deviation if in principle you should have not used BN. At training time we also keep a running average of mean and variance.

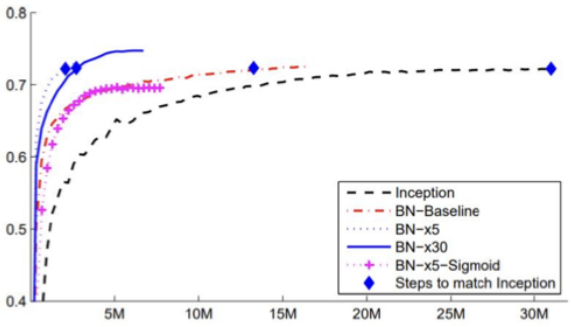
This is the only layer that behaves differently at train and test time, as the activations include a lot of randomness between epochs and I want the output to depend only on the input deterministically. I therefore use the running averages of mean and variance obtained at the end of training to be deterministic, they're not computed for each minibatch. So, in this test time, everything is constant and can be fused all together in a linear affine operation.

Pros:

1. It allows the use of higher learning rates
2. Careful initialization is less important
3. Training is not deterministic, it acts as regularization
4. No overhead at test time, can be fused with previous layers

Cons:

1. We don't really understand why it is so beneficial
2. The need to distinguish between training time and test time makes the implementation more complex and error prone
3. Hugely dependent on the hardware you get. Doesn't scale well to "micro-batches" (batches with 2/4 items).

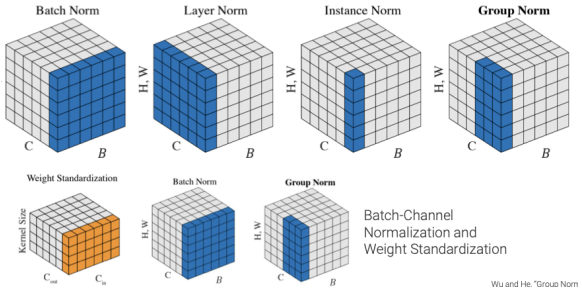


Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
BN-Baseline	$13.3 \cdot 10^6$	72.7%
BN-x5	$2.1 \cdot 10^6$	73.0%
BN-x30	$2.7 \cdot 10^6$	74.8%
BN-x5-Sigmoid		69.8%

Used also for Convolutional Layers. In Fully Connected Layers mean and variance are $1 \times D$ vectors, in Conv Layers you average everything by the number of channels, considering also the spatial dimension (keep the same properties as Convolution).

Evolutions

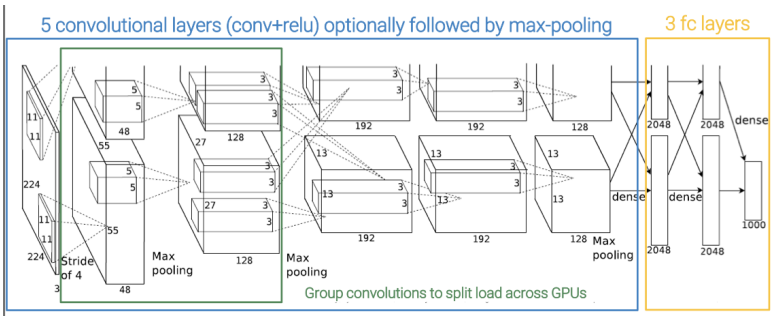
Investigated also alternatives, that different on the way the dimensions are selected to compute mean and variance. Layer Norm → averages all the channels at all spatial location of each image independently from the other (test and training time are the same). Instance Norm → normalize each channel of each image independently. Group Norm → form of layer norm and instance norms: you form groups of channels that go together for each image.



Successful Architectures

Good Practices come from basically the ILSVRC. People wanted to win, so moved along the gray areas: ensembles (run the same network on the same image using different crops) and heavy augmentations were legal.

AlexNet (2012)



A minor modification on top of LeNet5, using ReLU instead of sigmoid as non-linearities. Input: RGB image 224x224. Uses two streams, as GPUs had low memory back then (applies model parallelism). Double kernel needed, half on the bottom GPU and half on the top GPU,

working independently (do not want to slow by communicating between GPUs). Group Convolutions (still used for other reasons). There's some communication between the two GPU.

Breakdown of AlexNet

How many parameters, how costly it is, how much RAM it requires from the GPU for activation and parameters (assuming it is all in one GPU for simplicity).

Layer	Kernels	Kernel H/W	S	P	Activations H/W	Activations Channels	#Activations	#params (K)	flops (M)	Activations memory (MB)	Parameters memory (MB)
input					227	3	154587	0	-	75,5	0,0
conv1	96	11	4	0	55	96	290400	35	26986,3	283,6	0,4
pool1	1	3	2	0	27	96	69984	0	80,6	68,3	0,0
conv2	256	5	1	2	27	256	186624	615	114661,8	182,3	7,0
pool2	1	3	2	0	13	256	43264	0	49,8	42,3	0,0
conv3	384	3	1	1	13	384	64896	885	38277,2	63,4	10,1
conv4	384	3	1	1	13	384	64896	1327	57415,8	63,4	15,2
conv5	256	3	1	1	13	256	43264	885	38277,2	42,3	10,1
pool3	1	3	2	0	6	256	9216	0	10,6	9,0	0,0
flatten	0	0	0	0	1	9216	9216	0	0,0	0,0	0,0
fc6	4096	1	1	0	1	4096	4096	37758	9663,7	4,0	432,0
fc7	4096	1	1	0	1	4096	4096	16781	4295,0	4,0	192,0
fc8	1000	1	1	0	1	1000	1000	4097	1048,6	1,0	46,9
					Minibatch:	128	Totals:	62378	290851	1.406	714

Complete Architecture Breakdown

227 → unclear number coming from numbers… Declared as 224. Minibatches of 128.

First Convolutional layer has a large Stride (4) → also known as Stem Layer: bring down the spatial dimensions as soon as possible to bring down the costs, while rapidly increasing the receptive field.

When using more advanced optimizers, you don't only need to store the activation itself, but you also need the gradient of the loss w.r.t everyone of its entries (which is another tensor of the same size of the input activation). For every parameter, we will have its value and the gradient of the loss w.r.t. it. The final result will be a lower bound on the memory.

As they don't contribute too much, you can actually ignore pool layers.

Summary of meaningful observations:

- We have a stem layer at the beginning of the network
- Nearly all parameters are in the fully connected layers
- The largest memory consumption comes from the activations due to the first convolutional layers
- The largest number of flops is required by the convolutional layers
- Activations and parameters have a similar memory footprint at training time
- In total, we learn more than 60 million parameters, more than 290 Gflops to process a mini-batch, which accounts for 2.2 Gflops/image

ZFNet (2013)

Replica of AlexNet with some minor additions. The winner of ILSVRC 2013 is actually the company of the First Author of ZFNet. It is based on visualizations (e.g. kernels of the first layers) and ablation studies → too aggressive stem layer resulted in dead filters and missing frequencies in the first layer filters and aliasing artifacts in the second layer activations. Ease it a bit using 7x7 convolutions with stride 2 in the first layers and stride 2 also for the second (5x5) layer. Everything was still a bit static: the sequence of channels increased when dimensionality reduced, but in the end it decreased again → maintain a feasible network.

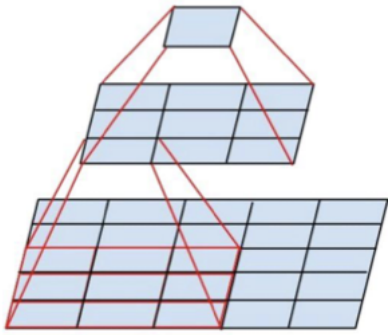
VGG

@June 1, 2023

Can we actually have a more rule-based approach design, while exploring depth? This result comes from the Visual Geometry Group, Oxford University. The main focus of their research was understanding what was going on: they used only combinations of 3x3 convolutions with stride and padding both set to 1, as to not influence spatial resolution of the activations (preserve sizes). Increase exponentially the receptive field with 2x2 max-pooling with stride 2 and padding 0. The result is that the number of channels doubles after each pool. Abiding to these rules, they trained deeper and deeper networks. Small consideration to do: Batch Norm had still not been invented yet, so they couldn't get over depth 19 (19 layers with trainable weights). They actually started with 11 weight layers, which was a network that was possible to train from scratch using random initialization, then used the values obtained and initialized all the corresponding layers in the deeper networks with those weights, while random initializing the new layers added. It is not the best way, but

worked. AlexNet in 2012 proposed Local Response Normalization (promote competition between neurons). It is dropped for good. Introduced a highly regular structure of convolution-pooling. The two most used variants are the deepest ones, VGG-16 and VGG-19.

Why 3x3 convolutions? They noticed that stacking 3x3 convolutions allows to have a larger receptive field (and the result is equivalent in terms of receptive field to use a 5x5 or 7x7 convolution), with a slower growth in the number of parameters and the number of flops. So Gradient Descent has an easier life and one can go deeper. Also, one gets the benefit of having two non-linearities. Where do we lose? The memory for activations increases (doubles in the case of 2 stacked convolution), because you need to store also the middles. This structure is still influential, although it is not anymore the reference. The stacking of convolutions and idea of regularity (stages) are still maintained. Stages = processing activations at the same spatial resolutions.



Summary of VGG-16: it is way bigger and heavier than AlexNet. It has more than double the parameters of AlexNet, the computational cost for the convolution layer stays constant due to the design of the network used. The normal tradeoff is that activations cost way more than parameters to store in the memory, with the layers most responsible for consumption being the first (due to the strict rules imposed, there is no stem layer).

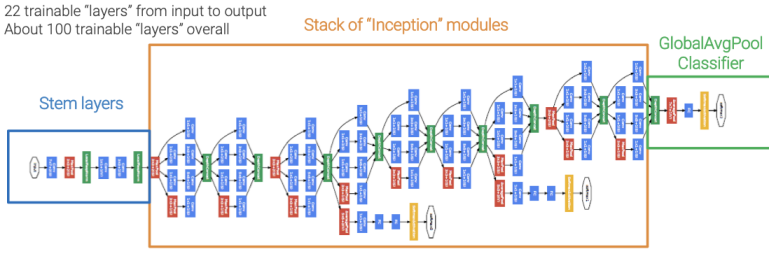
layer	Kernels	K_W/H	S	P	Actv H/W	Actv Channels	#Activations	#params (K)	flops (M)	Activations memory (MB)	Params memory (MB)
input					224	3	150528	0	-	73,5	0,0
conv1	64	3	1	1	224	64	3211264	2	22196,3	3136,0	0,0
conv2	64	3	1	1	224	64	3211264	37	473520,1	3136,0	0,4
pool1	1	2	2	0	112	64	802816	0	411,0	784,0	0,0
conv3	128	3	1	1	112	128	1605632	74	236760,1	1568,0	0,8
conv4	128	3	1	1	112	128	1605632	148	473520,1	1568,0	1,7
pool2	1	2	2	0	56	128	401408	0	205,5	392,0	0,0
conv5	256	3	1	1	56	256	802816	295	236760,1	784,0	3,4
conv6	256	3	1	1	56	256	802816	590	473520,1	784,0	6,8
conv7	256	3	1	1	56	256	802816	590	473520,1	784,0	6,8
pool3	1	2	2	0	28	256	200704	0	102,8	196,0	0,0
conv8	512	3	1	1	28	512	401408	1180	236760,1	392,0	13,5
conv9	512	3	1	1	28	512	401408	2360	473520,1	392,0	27,0
conv10	512	3	1	1	28	512	401408	2360	473520,1	392,0	27,0
pool4	1	2	2	0	14	512	100352	0	51,4	98,0	0,0
conv11	512	3	1	1	14	512	100352	2360	118380,0	98,0	27,0
conv12	512	3	1	1	14	512	100352	2360	118380,0	98,0	27,0
conv13	512	3	1	1	14	512	100352	2360	118380,0	98,0	27,0
pool5	1	2	2	0	7	512	25088	0	12,8	24,5	0,0
flatten	1	1	1	0	1	25088	25088	0	0,0	0,0	0,0
fc14	4096	1	1	0	1	4096	4096	102786	26306,7	4,0	1176,3
fc15	4096	1	1	0	1	4096	4096	16781	4295,0	4,0	192,0
fc16	1000	1	1	0	1	1000	1000	4100	1048,6	1,0	46,9
					Minibatch:	128	Totals:	138.382	3.961.171	14.733	1.584

Architecture Breakdown Summary for VGG-16

Inception v1 (GoogLeNet)

Winner of the 2014 ImageNet Challenge (vs VGG-Net). Objective: have high accuracy, but process a whole lot of images with a budget. We don't want simple rules, we want to tune every hyperparameter we have in the direction of efficiency.

> "The main hallmark of this architecture is the improved utilization of the computing resources inside the network. This was achieved by a carefully crafted design that allows for increasing the depth and width of the network while keeping the computational budget constant"



We have the skeleton of the structure we still use nowadays:

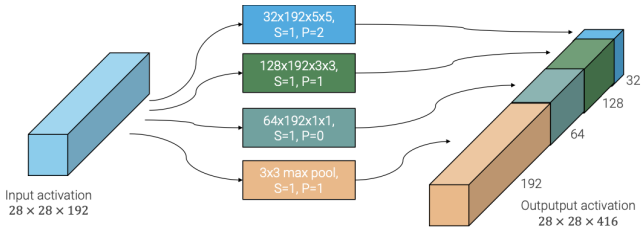
- Stem Layers → the importance of the stem layer actually comes from Inception v1
- The Body of the Network → it is the changing part between the various networks

- A Classifier → the idea of having several fully connected layers is dropped due to computational budget

We have a change in how we define what is a layer. The layers are now the blocks. Then we also have parallel layers. This structure is a consequence of the so-call inception modules.

Stem layers of Inception are fairly aggressive, going from 224x224 to 28x28 images. Very very coarse spatial resolution activations to not spend the same amount of resources as VGG. Still, aware of AlexNet as per ZFNet lessons, uses only strides of 2 and the largest convolution is 7x7.

Naive Inception Module



4 layers in parallel, with stride and padding fixed to preserve spatial resolution.

- 5x5 convolutional layer
- 3x3 convolutional layer
- 1x1 convolutional layer
- 3x3 max pooling → uses same stride and padding of a 3x3 convolution. We have pooling without stride basically.

The output is stacked along the channel dimension to create a new representation.

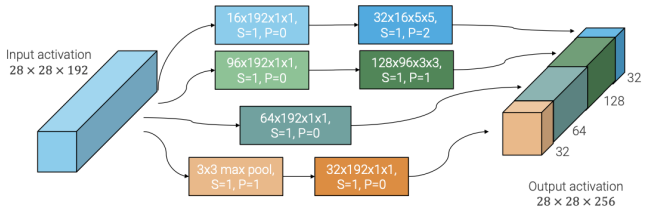
Two issues:

1. We have no control on the number of output channels of the max-pooling, so the output activation will grow the number of channels no matter what we choose for the convolutions, that can only add to it.
2. 5x5 convolutions and 3x3 convolutions are costly and due to the channel increase they become prohibitively expensive if we stack a lot of them

A 1x1 convolution is basically a scaling of the input. I will have a kernel (with a bias) which is basically a vector with the same dimension as the number of channels in the input activation. It is a meaningful operation, as there is no spatial aggregation of information but they mix channels. They are cheap, they compute new representation of each location in isolation, but still mixing channels. This is the cheapest way known to diminish the number of channels.

Consider one location of the input activation: if we look at the channel, they are x features that we computed so far on that location (a row vector $1 \times x$). The result we want is the $1 \times c$ vector formed by all channels at the same spatial location in the output. I can then see the 1x1 convolution as a dot product between the row vector and the kernel ($x \times c$) + the biases ($1 \times c$). This is a linear layer applied to each spatial location of an activation (they slide over images). So stacking with ReLUs in between, we can think of it as sliding an MLP at each location. And as we know, the MLP is a universal approximator.

Inception Module



We have the same operations as before, but we have a 1x1 convolution on each path, to reduce the number of output channels, before the convolutional layers and after the max pooling. This allows a gain in terms of speed and number of flops.

From now on, the 1x1 convolutions will be used almost everywhere. We have a lot of hyperparameters to change the number of channels as we like.

Other idea of GoogLeNet (other name of Inception v1, homage to LeNet5): we get rid of the fully connected layers at the end. The problem is that the last (small) activation before the fully connected layers has a lot of channels; so, no matter how much you're conservative, when you flatten the activation, the matrix becomes huge. The idea is: we have still a spatial structure, which is very coarse, and we might not care where a feature actually is. So, let's get rid of the spatial dimensionality by averaging all and get down to a scalar for each channel: I apply an average pooling on each channel to compute the global average of the response for each feature throughout the image.

The structure now is: I arrive to a spatially coarse activation, I apply global average pooling and project them to the number of classes you need with the image classifier (one single fully connected layer). It is shown that this is equivalent to the previous configuration with flattening and 3 fully connected layers.

This is negligible w.r.t. VGG (1M parameters and basically no flops).

Detail: if I process larger images, the thing I need to take care of is that the kernel of the pooling should be adaptable (not fixed): in Pytorch: `torch.nn.AdaptiveAvgPool2d` allows to compute in theory any input image size (in the surroundings of 224x224, though)

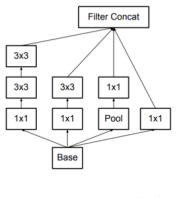
So, to sum up GoogLeNet has a way more efficient design, with 7M parameters to achieve sota, a lot less flops and memory consumption than VGG. Downside: we don't have guiding light to apply to another problem besides ImageNet. Also, with Inception, you're giving gradient descent all the options (it has a lot of paths opened, so the optimal path for GD is surely there). We can see this as a lot of ensemble of VGG-Styled Networks (if you follow the single paths).

	inception 1x1				inception 3x3			Inception 5x5			Maxpool		Activations						
Layer	Ks	H/W	S	P	C_out	1x1	H/W	C_out	1x1	H/W	1x1	H/W	H/W	Channels	#Activations	#params (K)	flops (M)	Acts memory (MB)	Params memory (MB)
input													224	3	150528	0	-	73,5	0,0
conv1	64	7	2	3									112	64	802816	9	30211,6	784,0	0,1
pool1	1	3	2	1									56	64	200704	0	231,2	196,0	0,0
conv2	64	1	1	0									56	64	200704	4	3288,3	196,0	0,0
conv3	192	3	1	1									56	192	602112	111	88785,0	588,0	1,3
pool2	1	3	2	1									28	192	150528	0	173,4	147,0	0,0
incep1	64	1	1	0	128	96	3	32	16	5	32	3	28	256	200704	163	31380,5	196,0	1,9
incep2	128	1	1	0	192	128	3	96	32	5	64	3	28	480	376320	388	75683,1	367,5	4,4
pool3	1	3	2	1									14	480	94080	0	108,4	91,9	0,0
incep3	192	1	1	0	208	96	3	48	16	5	64	3	14	512	100352	376	17403,4	98,0	4,3
incep4	160	1	1	0	224	112	3	64	24	5	64	3	14	512	100352	449	20577,8	98,0	5,1
incep5	128	1	1	0	256	128	3	64	24	5	64	3	14	512	100352	509	23609,2	98,0	5,8
incep5	112	1	1	0	288	144	3	64	32	5	64	3	14	528	103488	605	28233,4	101,1	6,9
incep6	256	1	1	0	320	160	3	128	32	5	128	3	14	832	163072	867	41445,4	159,3	9,9
pool4	1	3	2	1									7	832	40768	0	47,0	39,8	0,0
incep7	256	1	1	0	320	160	3	128	32	5	128	3	7	832	40768	1042	11860,0	39,8	11,9
incep8	384	1	1	0	384	192	3	128	48	5	128	3	7	1024	50176	1443	16689,7	49,0	16,5
avgpool	1	1	1	0									1	1024	1024	0	6,4	1,0	0,0
fc1	1000	1	1	0									1	1000	1000	1025	262,1	1,0	11,7
													Minibatch:		128	Totals:	6.992	389.996	3.251	80

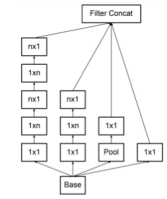
Architecture Breakdown for GoogLeNet (Inception v1)

Inception v3

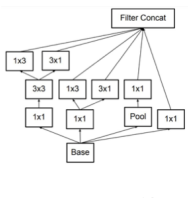
The global design stays the same. They borrow the idea of VGG of factorizing convolutions, so (e.g.) the 5x5 convolution becomes a stack of 2 3x3 convolutions. Also they play with the idea with different Inception modules at different spatial resolutions.



Factorization A, used for fine-scale activations (35x35)



Factorization B, used for mid-scale activations (17x17)



Factorization C, used for coarse-scale activations (8x8)

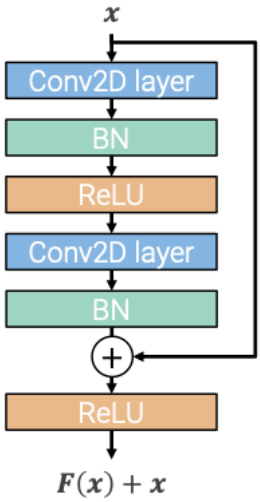
Inception v2 was actually proposed with v3 and immediately shown to be less performant, therefore it's like it never existed.

Residual Networks

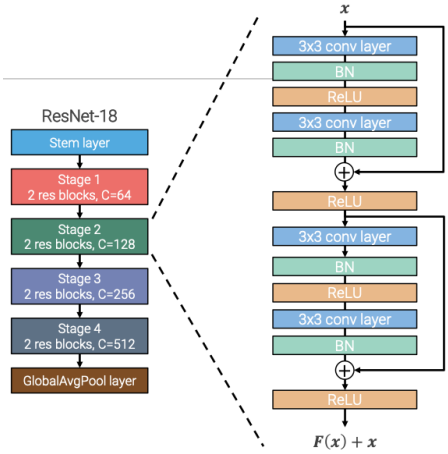
VGG told us: "if you grow depth, you will have better performance", but stopped at 19 layers. People at Microsoft Asia tried with 56 and found that training and test errors were worse (thought of an overfitting the training set, but it's more of a training problem). The observation was that it is not expected and quite odd, as by construction, I should get a solution which is at least as good as a low-number-of-layers network one. The problem is: Gradient Descent is not able to navigate this higher space. One way around the problem is to change the parametrization of the problem.

If we know that the identity is a good solution to some layers, let's bootstrap GD to start closer to the identity, so that it can optimize and improve over the identity. Instead of having a VGG-Style block now (with Batch Norm which has now been invented), we have the same stacking with a skip (or residual) connection every two convolutional layers: you take the input activation to the end of the block, you sum it with what's been processed in the block and

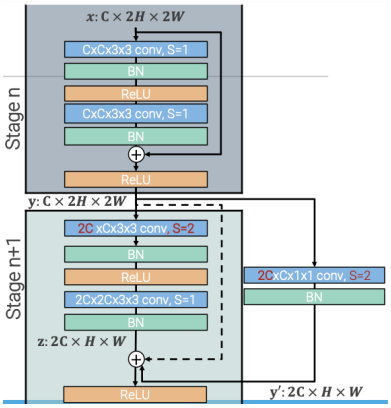
then go through a ReLU. The function we're now computing is not an $H(x)$ anymore, but a $F(x) + x$, with the $F(x)$ initially initialized to be close to 0 (starts as the identity function, very very close to it). Gradient Descent will move from there, and we know this is a better start than random initialization.



From ResNets on, everything has Residual Connections. Uses also Stem Layers and Global Average Pooling. Stages are designed with the following rules: 3x3 convolutions only, as in VGG, with Batch Norm and each stage is a repetition of blocks, when you move from a stage to the other the number of channel doubles and the spatial resolution halves, and each stage is constituted of two subsequent residual blocks. Simple structure as VGG, with skip connections. Naming convention of VGG: ResNet-X, with X number of layers with learnable parameters.



Stem Layer is gentler, with a better engineering balance (down to 56x56). The end is very similar. The only difference is when we want to move from a stage to another stage (as the number of channels doubles and the spatial dimension halves) as there's no pooling inside the residual blocks. How do we fix dimensionality? We create a skip connection which is not an identity and has a 1x1 convolutional layer that changes the dimensionality (2C output channels and stride 2) to match.

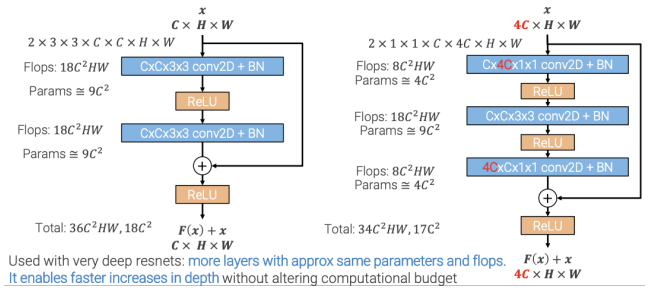


Just adding residual connections brings the results to where one would expect. Results in every challenge astonishing in full AlexNet style (“We knew they were here to last”).

@June 5, 2023

Simple modification to VGG-Style Network to add skip connections. By construction, there exists a solution that makes deeper networks no worse than shallower networks. So, stacking networks shouldn’t lose performance. We add an inductive bias to the design, so that by adding layers I don’t lose and let GD improve from there.

The effect is to reduce training error (it decreases with depth). Easier to train w.r.t. the vanilla plain VGG. As a byproduct, you get better generalization. This is true for CIFAR10. On ImageNet, you need another step. Now, stacking standard residual blocks makes it costly to reach high depths (increase of a factor 2-layers every time, in terms of flops and parameters). In order to try deeper architectures, we need depth to increase faster (blocks roughly equivalent to the original residual blocks, but with more layers e.g. 3 instead of 2).



Bottleneck RB → maintain the 3x3 convolution, surrounded by 2 1x1 convolutions with the task of:

1. bring the number of channels from 4C down to C, the 3x3 conv costs then exactly the same
2. brings the number of channels back to 4C so one can have the right shape to perform the residual connection.

Total number of FLOPS and parameter is even a bit less than the standard ResBlock.

Proposed families of deeper networks. ResNet-50 is a good default choice because of Transfer Learning. Transfer Learning is the process of using and adapting a pre-trained neural network to new datasets.

One entity pays the price of a huge learning on a large dataset (e.g. ImageNet 21K). The network has encoded a lot of knowledge, therefore, when you need another task which is similar (e.g. Flower Classification), it has been shown that you can initialize your CNN with the weights learned from some large dataset, then it is easier to solve a new dataset. One variant is to initialize the CNN and don’t touch it (frozen approach), or initialize and train (fine-tuning) on the new dataset.

Process:

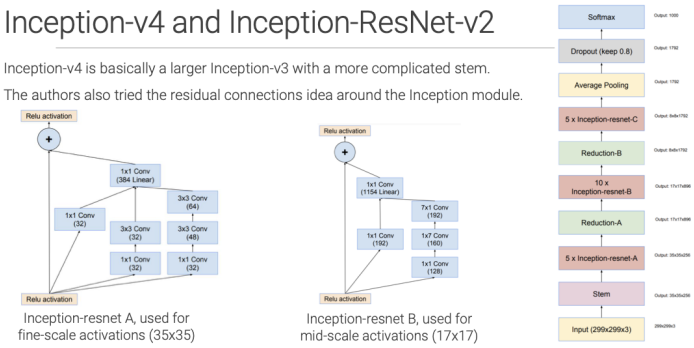
1. Someone does the heavy lifting and then publishes the weights.
2. Then you want to solve another problem, not related. You have now a different number of classes, so you perform network surgery: cut the head of the network and substitute it with a new fully connected layer which is purely random. Now:
    a. Freeze the rest and then train only the fully connected layers (you don’t necessarily need to compute gradients for PyTorch tensors). Use the backbone as a pre-trained feature extractor.
    b. Train everything: both the new head and the feature extractor (backbone fine-tuning). When you do the fine tuning, at the very beginning you don’t want the head to skew the results of the backbone, which is at a better stage. So freeze the backbone for a while to preserve it, then unfreeze it when the head reaches a more quality level. Also, the backbone is supposedly good, so an idea is to not let it change too much, so use a smaller LR (one order of magnitude less than the original) to not move too much. Generalize: progressive LR. The extractor probably works well. The specificity grows with depth (e.g. stage 4). So use progressive learning rates (in the sense that they are different for each stage, like higher for the deeper stages).

When you transfer learn, take care of Batch Normalization, especially with small datasets. Keep them frozen even when fine-tuning, to avoid the risk of skewing the statistics (mean and deviation) to damage the other layers.

Skip connections make the loss landscape smoother. Why? ResNet can be unraveled (take each possible path to it) and see that they are an ensemble of VGG style networks. Ensembles make for smooth losses even if the vanishing gradient problem is not really solved.

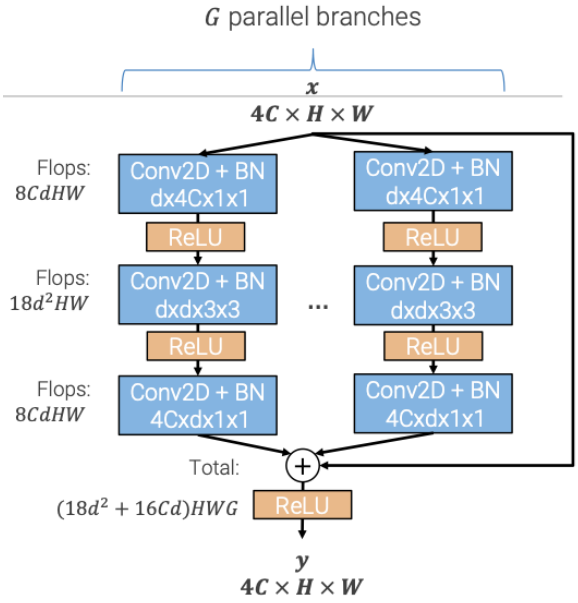
Inception-v4 and Inception-ResNet-v2

Idea of combining an Inception Module (v3), with a skip connection and the need to reconcile dimensions (used 1x1 convolutions). proposed also Inception-v4, nothing special. Inception-ResNet becomes the new state-of-the-art by Google. Then, ResNet wanted regularity with the same multi-branch structure



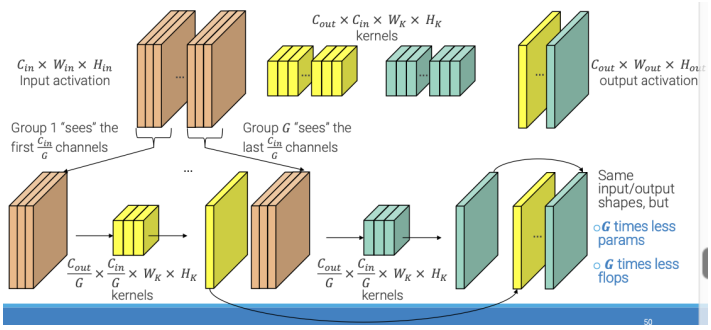
ResNeXt

Inception modules are effective multi branch architectures that compute different representations that are then concatenated. Need a regular way. Start from the bottleneck residual block, as the Inception ResNet v2 starts with a 1x1 convolution, has an irregular amount of 3x3 convolutions and then another 1x1 again to make dimensions match. The idea of ResNext is to start from it and add multi-branches. Add a hyperparameter G = cardinality of the block (how many branches). At the end, sum everything. But the hyperparameters are not the same as the bottleneck, to avoid linear increase of flops consume with G. Once I design G, I need to process a smaller number of channels d, which is computed based on G and the flops consumed by the number of flops consumed by a bottleneck residual block with G=1.



This structure can be realized and will perform as well as group convolutions.

Group convolutions (AlexNet's way to fit all in more GPUs): each kernel sees less channels.

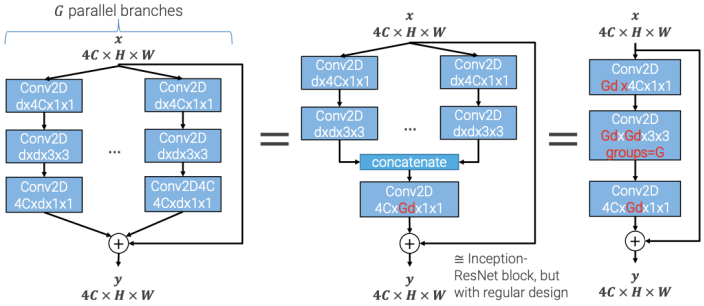


Case of G=2: two branches, each of them producing an activation that has a $4C \times H \times W$ from an activation that is $d \times H \times W$. Focus on $y^{(1)}$, which will have $4C$ channels. The generic k-th channel will be the convolution of the set of weights and $a^{(1)}$. So if we expand this out this will be the product of the first number in the k-th kernel of w times the first entry in the activation and so on… we have d entries in the activation and d entries in the weights kernel. Same happens for $y^{(2)}$. $y^{(k)}$ is the output of $y^{(1)}$ $y^{(2)}$ and the input of the k-th channel $x^{(k)}$. I can consider it as being the sum of the convolutional part plus the input. The sum is what we've seen. Note that there is no ReLU (no nonlinearity). So I can realize this summation by performing a single 1x1 convolution with 2d input channels (concatenation along depth). Associative property of summation.

Result: you re-fall into Inception-ResNet, but with a more regular design.

The first 1x1 convolutions process the same activations → can have a single 1x1 convoluton with Gd output channels, produce an activation and then split along depth.

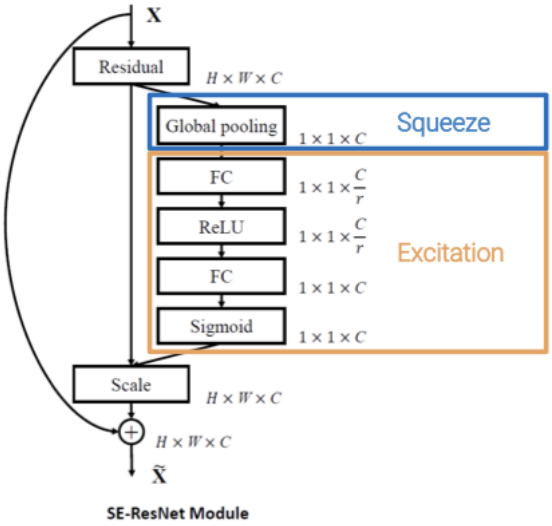
The middle takes an activation with Gd channels, splits them and process them separately, then re-concatenates. So it has Gd input, splits in G groups, reconcatenates into Gd output channels. So it can be seen as a group convolution with G groups. Nice compact form.



More effective. If you keep increasing G for ResNeXt-50, performance increases and d diminish. Using a convolution to process all the input channels is basically a waste. More effective to have many, smaller convolutions (costs the same in terms of Flops, but more accurate). With ResNeXt, we stop hunting for depth; we instead work on other hyperparameters by keeping the depth manageable (e.g. width). Also works on bigger input resolutions.

Squeeze-and-Excitation Net (SENet)

Last Edition of the ImageNet Challenge, which was won by building on top of ResNeXt. New module, with a squeeze (squeezes out partial dimensions by using Global Pooling), what I obtain is how active my channels are in the activations. Then there's a bottleneck structure, which creates some kind of competition/interaction between channels. This goes through a sigmoid to obtain a number for each channel between 0 and 1, that we use as a scaling factor for the input activation. Form of weighting. This leads to a form of gain (even marginal).

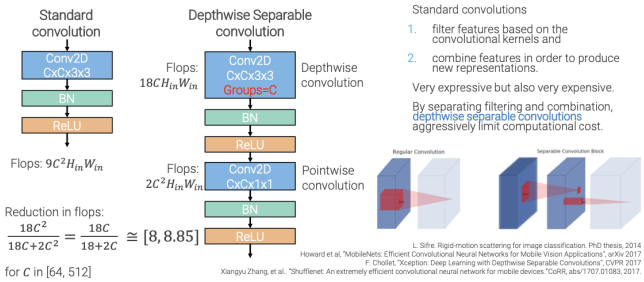


SE-ResNet Module

Can also be seen as a form of Self-Attention, due to the scaling factors obtained from one's own activations.

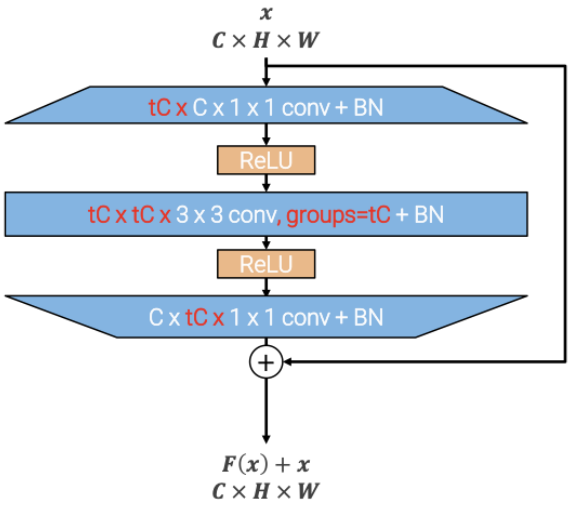
Depthwise Separable Convolutions and MobileNet-v2

The interest shifted to running things on lower-spec hardware. Push further into the idea of grouped convolutions. If you have as many groups as input channels, each group has 1 channel (depth=1) and processes it independently. Spatial reasoning is very fast, but no meaningful. So we make it follow a Batch Norm, a ReLU and a PointWise Convolution (1x1 convolution with the same number of channels). If you stack this new approach, you get almost an order of magnitude less reduction in Flops.



Another iteration of MobileNet produced a combination of Residual Blocks and DSC as the 3x3 convolution in the bottleneck residual block processes spatial information in a compressed domain, which may result in information loss going through the ReLU.

This is obtained by inverting residual blocks (the first now expands the number of channels), the central still does the work, the last one brings it back to the size for summing. The operation in the middle is a depth-wise convolution to spend the less possible.



This is MobileNet-v2. Reduced computational cost due to the the move into a compressed domain. Though, when we need to classify (we're in the "head"), we need to expand the representation to a reasonable amount of channels.

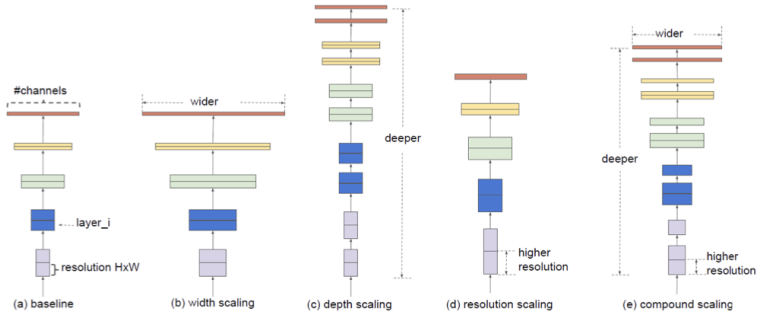
Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

Wide ResNet

@June 8, 2023

Scaling in depth is not the only way to scale a model. We can also scale in width (number of channels in the activations at each layer). We call the result Wide ResNet/WRN-n-k, with n number of layers and k the multiplier for the number of channels. Faster in practice than ResNet-152. Even if the number of flops is the same, this is a crude indicator of velocity (less layers to traverse per number of flops means faster) + wider layers tend to be more GPU friendly.

If I want to improve over my baseline ResNet-50, what should I do optimally? Answer in EfficientNet



EfficientNet

The answer is Compound Scaling, which means I scale width, depth and resolution altogether. Increasing each dimension in isolation will work at first, but eventually will saturate and plateau.

Intuitively, we can deduce that scaling dimensions are not independent. Larger images with higher resolution would require more depth to have a larger receptive field; Also, you have more information, so you may want a higher number of channels.

How do we scale? Set up an optimization problem and solve it heuristically. I can grow my network to fill the GPU.

$$\begin{gathered}\text{depth multiplier: } d = \alpha^\phi \\ \text{width multiplier: } w = \beta^\phi \\ \text{resolution multiplier: } r = \gamma^\phi \\ \text{s.t. } \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2, \\ \alpha \geq 1, \beta \geq 1, \gamma \geq 1\end{gathered}$$

Notice that flops in convolutions scale linearly with depth, but quadratically with width and resolution. By forcing the equation there to approximate to 2, flops will approximately increase by 2^ϕ when scaling by ϕ (i.e. double for each increase of ϕ)

ϕ is the compound coefficient that expresses how much I want to scale up. I start from a baseline B0 with $\phi = 0$ (and consequently no multipliers). The only thing I need to define are α, β, γ → I assume to have double the number of flops ($\phi = 1$) and look for α, β, γ according to this constraint and check which combination works better. When I found the best α, β, γ I keep them fixed for $\phi > 1$.

Results are better with compound scaling according to the optimization problem.

They also proposed the “best” baseline architecture, with a Neural Architecture Search → try some blocks and see which combination works best with a train and validation set. The result follows a similar trend to MobileNet-v2, with a different approach to the dimensions of convolutions (no regularity). NAS always adds Squeeze and Excitation Layers in the Inverted Residual Blocks.