

CS4243 Computer Vision & Pattern Recognition

by Zong Xun

Fundamentals

A **signal** is a physical phenomenon which conveys information and energy.

Geometric Transformations

Two basic operations:

- Spatial transformation of coordinates
- Intensity interpolation that assigns intensity values to the spatially transformed pixels.
- Translation is not a geometric transformation.

Affine Transformations

Scaling, translation, rotation, and shearing.

The key characteristic of an affine transformation in 2-D is that it preserves points, straight lines, and planes (parallel lines remain parallel, and the midpoints of line segments remain in the transformed segments).

The 4 transformation can be expressed as:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

When an image is rotated, some pixels are no longer aligned perfectly with the original grid. Intensity interpolation helps determine the intensity values for these new pixel positions by estimating them based on the surrounding pixel values.

- Nearest-neighbor interpolation
- Bilinear interpolation
- Bi-cubic interpolation

Pixel-Wise Operations

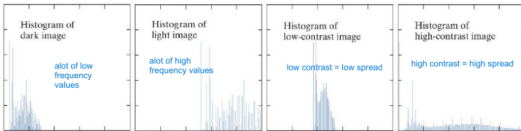
Apply function on any pixel of the image.

$$b(x, y) = F(a(x, y))$$

where F is x and y invariant: $b = 255 - a$.

Histogram: 1st statistical momentum

- How many pixels in each grayscale/color?
- Gives us a good description of the image.



Pixel-wise operations

- Histogram manipulation, Image normalization
- Grayscale Modification

$$b(x, y) = \left[\frac{a(x, y) - \min(a)}{\max(a) - \min(a)} (n_{\max} - n_{\min}) \right] + n_{\min}$$

- Histogram stretching. Replace range with 255. Spread intensity values to increase contrast.

Key definitions:

- Brightness is the measured intensity of all pixels comprising an ensemble that constitutes the digital image after it has been captured, digitized and displayed.

- Contrast refers to the amount of color or grayscale differentiation between various image features.

Notes

- When zooming in, the image may appear to be pixelated or blurry because the software has to interpolate and estimate the new pixel values, which can reduce sharpness and fine details.
- When zooming out, finer details may be lost as pixels are merged or skipped.

Local Operations, Filtering and Convolution

Convolution Theorem

Convolution theorem: If T is a linear shift (time) invariant system, and h is the impulse response of T , y can be obtained by the convolution of x and h .

$$Y(m, n) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} x(m, n) \cdot h(x-m, y-n)$$

where h is the kernel matrix and x is the digital image.

Linear Shift/Time Invariant Systems A system is said to be linear if

$$T[a_1x_1(n) + a_2x_2(n)] = a_1T[x_1(n)] + a_2T[x_2(n)]$$

for any two inputs $x_1(n)$ and $x_2(n)$ and for any complex constants a_1 and a_2 .

Shift Invariance Let $y(n)$ be the response of a system to an arbitrary input $x(n)$. The system is said to be shift-invariant if, for any delay n_0 , the response to $x(n - n_0)$ is $y(n - n_0)$. A system that is not shift-invariant is said to be shift-varying.

Properties of Convolution

- Commutative, Associative, Distributive
- Dimension of output is given by

$$\left\lfloor \frac{n+2p-k}{s} \right\rfloor + 1$$

- $p = \frac{k-1}{2}$, full = $n + 2p$, same = n , valid = $n - 2p$.
- Stride commonly used to **down-sample** images for DL and feature extraction, **but not in image processing**.
- Zero vs Average vs Random (padding)

Filters

- Lowpass: A kind of averaging/summation/integral operation. **Makes the image less noisy but blur**. An example is the Gaussian Filter.

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

- Filter matrix tends to be **symmetric**. Larger kernel, $h \times h$, has more significant blurring effect.
- Highpass: A kind of derivative/difference operation. **Used to detect lines and edges in different directions. Make the image sharper**. Have **alternating** signs in the filter matrix.
- Bandpass: [low, high].
- Bandreject: 1 - [low, high].

Laplacian

The Laplacian operator computes the second derivative of the image

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

where it can be rewritten as:

$$\nabla^2 f = f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1) - 4f(x, y)$$

$$G_x = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

It is a highpass filter.

Sobel Operator

The **Sobel operator** is used for edge detection by computing the **gradient** of image intensity. Highlight regions of rapid intensity change. Consists of two 3×3 convolution kernels:

Horizontal Gradient (G_x) – detects vertical edges

$$G_x = a * \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Vertical Gradient (G_y) – detects horizontal edges

$$G_y = a * \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Applying these kernels results in two gradient components:

- G_x detects vertical edges (changes in horizontal intensity).
- G_y detects horizontal edges (changes in vertical intensity).
- May add diagonal edge detectors too, although mathematically problematic due to lack of orthogonality.

$$h_{s135} = \begin{bmatrix} -2 & -1 & 0 \\ -1 & 0 & 1 \\ 0 & 1 & 2 \end{bmatrix}, h_{s45} = \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix}$$

G represents the **strength of an edge** at each pixel and the **gradient direction** (edge orientation) is given by

$$G = \sqrt{G_x^2 + G_y^2} \quad \theta = \tan^{-1} \left(\frac{G_y}{G_x} \right)$$

Sometimes, $|G| = |G_x| + |G_y|$. Then, we may apply thresholding on $|G|$

$$|G_{thres}| = \begin{cases} 1 & |G(i, j)| > Threshold \\ 0 & else \end{cases}$$

Interpretation of Gradient Magnitude

- G measures how sharp the intensity change is at a pixel.
- **Higher** G values indicate strong edges.
- **Lower** G values indicate smooth or uniform regions.
- $G = 0$ means no significant intensity change (flat region).

Applications of Sobel Operator

- Finding strongest edges & their directions
- Border Tracking
- Vectorization

Unsharp Masking

Mathematically, the sharpened image I_s is given by:

$$I_s = I_o + k(I_o - I_b)$$

where I_o = original image, I_b = blurred image, k = scaling factor controlling the sharpening intensity and $I_o - I_b$ = high-frequency details.

1. Apply Gaussian blur to create a soft version of the image.
2. Subtract the blurred image from the original to extract high-frequency details.
3. Scale the extracted details and add them back to the original image.

Median filter

Replace center pixel with the median of $n \times n$ window.

- Good for dealing with dot noises, or multiplicative noise.
- **No unsharpening/blurring effect**, better than mean filter.
- Patch size invariant (kernel size)

Noise

White noise is a random signal having equal intensity at different frequencies, giving it a constant power spectral density. The presence of noise in an image might be **additive** or **multiplicative**. In the Additive Noise Model, an additive noise signal is added to the original signal to produce a corrupted noisy signal that follows the following rule: $w(x, y) = s(x, y) + n(x, y)$.

Dot noise: Mostly result of the multiplication of a [0,1] or [1,255] matrix with your image. Could be due to channel disconnection or saturation.

Signal to Noise Ratio is used to characterize image quality.

$$SNR_{dB}(a) = 10 \log_{10} \left(\frac{power(a)}{power(noise)} \right)$$

where $power(noise) = \sigma^2(n)$ (variance).

Energy, Power, and Entropy

$$E(a) = \sum_i \sum_j a^2(i, j) \quad P(a) = \frac{1}{MN} * E(a)$$

- Measure of the degree of randomness (higher with noise)

$$Entropy(a) = - \sum_k P_k \log_2(P_k)$$

where K refers to # gray-levels, and P_k is the probability associated with gray level k (use histogram).

Image Transforms

Image processing tasks are sometimes best performed in a domain other than the spatial domain, e.g. Fourier domain.

Key steps:

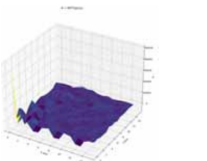
1. Transform the image
2. Carry the task(s) in the transform domain.
3. Apply inverse transform to return to the spatial domain. A few image transforms are not invertible, but they often are.



A transform is done by modulating (multiplication) the signal with a limited set of basic signals (aka wavelets, kernels).

The inverse transform would be done by multiplication of the transformed signal by the same or different kernels. **The hypothesis insists that any image can be decomposed into its basic components by transformation and composed back to its original shape by the inverse transform.** An image can be represented in the transform domain in a format completely different from the ordinary spatial domain format.

Walsh/Hadamard Transform



- f is the image, x and y are pixels indexes
- F is the transformed image, u and v are horizontal and vertical sequence/frequency indexes.
- + and Σ on exponential are binary/modula-2 operations
- $b_i(.)$ is the i^{th} bit of . in an n-bit representation
- We assumed a square NxN image
- $n = \log_2 N$

Walsh/Hadamard Transform

WHT is a non-sinusoidal **orthogonal** transform that decomposes an image into Walsh functions, which are square waves consisting of +1 and -1 values.

Given an image matrix $I(x, y)$ of size $2^n \times 2^n$, the WHT is:

$$H = WIW^T \iff I = WHW^T$$

where W is the Hadamard matrix, and $W = W^T$.

Because of this orthogonality, treat W_N as a set of n mutually-orthogonal “square-wave” basis patterns. The Hadamard matrix W_N is recursively defined as:

$$W_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad W_N = \begin{bmatrix} W_{N/2} & W_{N/2} \\ W_{N/2} & -W_{N/2} \end{bmatrix}$$

Sequence Ordered Walsh-Hadamard Transform

The Walsh-Hadamard Transform can be arranged in **sequence order**, where the rows are sorted according to the number of zero-crossings (sign changes). The sequence-ordered Walsh functions are useful in signal processing, pattern recognition, and feature extraction.

- Lower sequences have fewer zero-crossings and correspond to low-frequency components.
- Higher sequences have more zero-crossings and represent high-frequency details.

WH Domain Filtering Algorithm

1. Image is a , gray level, square, $M \times M$, $M = 2^m$
2. Compute h , sequency ordered Hadamard matrix, $M \times M$
3. Transform: $A = h * a * h$, ($*$ is matrix multiplication)
4. Configure your WH filter, F , again an $M \times M$ matrix
5. Do the filtering $A_f = A \cdot F$ (element-wise multiplication)
6. Do the inverse transform $a_F = \frac{(h * A_F * h)}{M^2}$

Texture

- Texture consists of texture primitives (texture elements) called **texels**.A texture primitive is a contiguous set of pixels with some tonal and/or regional property.
- Texture description is based on **tone** and **structure**. Tone describes pixel intensity properties in the primitive, while structure reflects spatial relationships between primitives.
- Texture description is **scale-dependent**.
- A *constant* texture, is constant, slowly changing or approximately periodic. In a strong texture, the primitive set is well defined and the structure is rather regular. In other words, elements and spatial relations between them are clearly determinable.
- In a *weak* texture definition of a crisp set of primitives is relatively more difficult and spatial correlation between primitives is also low. An extremely weak texture could be considered as a *random* texture.

Texture processing

Statistical Methods

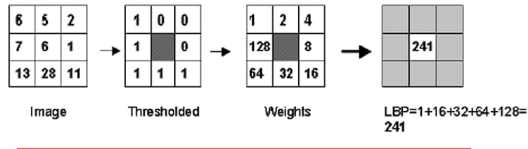
Histograms Limitations: Histograms only capture the distribution of pixel intensities without considering the spatial relationships between pixels. This means different textures can have identical histograms, making it difficult to distinguish between them using histograms alone (First Order Statistics).

Want: Capture spatial relationships and patterns within textures for better differentiation between textures.

Local Binary Patterns

To locally threshold the brightness of a pixel's neighborhood at the center pixel gray level to form a binary pattern.

The LBP operator is **gray-scale invariant**: texture is described in the neighborhood consisting of P ($P > 1$) equally spaced points on a circle of radius $R > 0$ centered at the center pixel. *Does not depend on specific gray levels (intensities of the pixel), but rather on their relative ordering.*



$$LBP_{P,R} = \sum_{p=0}^{P-1} s(g_p - g_c)2^p$$

$$s(x) = \begin{cases} 1 & , x \geq 0 \\ 0 & , x < 0 \end{cases}$$

Histogram bins represent the number of local texture patterns, and the height of the bin tells us how often that pattern appears.

Grey Level Co-occurrence Matrix

$$\Phi_{d,\theta}(i, j) = \sum_{u=1}^U \sum_{v=1}^V \rho(x(u, v), x(u', v'), i, j)$$

$$\rho(x, x', i, j) = \begin{cases} 1 & \text{if } x(u, v) = i \text{ and } x(u', v') = j \\ 0 & \text{otherwise} \end{cases}$$

- Varies rapidly with distance in fine textures and slowly in coarse textures.

- Suppose the part of a textured image to be analyzed is an $M \times N$ rectangular window. An occurrence of some gray-level configuration may be described by a matrix of relative frequencies describing how frequently two pixels with gray-levels $[i, j]$ appear in the window separated by a distance d in direction θ .
- These functions capture different properties of the texture.

- **Maximum:**

$$\max_{i,j} \Phi(i, j)$$

- **Energy:**

$$\sum_{i,j} \Phi(i, j)^2$$

- **Entropy** (Texture Homogeneity):

$$-\sum_{i,j} \Phi(i, j) \log(\Phi(i, j))$$

- **Correlation** (Image Linearity Metric. Linear directional structures in direction θ result in large correlation values in that direction. This can also measure the image coarseness):

$$\sum_{i,j} \frac{(i - \mu_i)(j - \mu_j)}{\sigma_i \sigma_j} \Phi(i, j)$$

- **Inverse Difference Moment** (extent to which the same tones tend to be neighbors):

$$\sum_{i,j} \frac{1}{1 + (i - j)^2} \Phi(i, j)$$

- **Inertia** (texture dissimilarity measure):

$$\sum_{i,j} (i - j)^2 \Phi(i, j)$$

The Dissimilarity metric is computed by taking the absolute difference between pairs of pixel intensities and weighing them by their corresponding frequency of occurrence.

Laws’ Texture Energy Filters

Laws’ filters are designed to capture texture patterns using convolution with predefined kernels. Each filter is formed by the outer product of two 1D vectors of length 5.

- 1D kernels:
 - L5 (Level) = [1 4 6 4 1]
 - E5 (Edge) = [−1 − 2 0 2 1]
 - S5 (Spot) = [−1 0 2 0 − 1]
 - R5 (Ripple) = [1 − 4 6 − 4 1]
 - W5 (Wave) = [−1 2 0 − 2 1]
- 2D filters: Formed as $F_{XY} = X^T \cdot Y$, e.g., $L5 \times E5$.
- Processing steps:
 - Convolve image with each filter.
 - Compute local energy (e.g., absolute response).
 - Use as features for texture classification or segmentation.

Karhunen–Loeve Transform (KLT)

KLT, also known as PCA in signal processing, is used to extract features from local image patches using eigenvalue decomposition of their covariance structure. **More computationally efficient compared to 1st and 2nd order statistics.**

- Apply $n \times n$ sliding windows over the image.
- Each window is reshaped into a vector, creating a data matrix of size $k \times n^2$, where k is #sliding windows.
- Typical neighborhood sizes: $n = 3, 5, 7$.

Covariance Matrix and Eigen Decomposition

$$C(x) = E \left[(x - \mu_x)(x - \mu_x)^T \right]$$
$$(C(x) - \lambda_i I)e = 0$$

- μ is the mean vector, I is the identity matrix.
- λ_i and e are the eigenvalues and eigenvectors.

Eigenfilter Bank

- The eigenvectors ($n^2 \times 1$) is reshaped to form a bank of $n \times n$ adaptive filters. These filters capture **dominant variation directions** in the patches.
- The eigenvalues indicate how much of the variance is projected onto each of these directions (higher = more information).
- Convolve each filter with the image to obtain detail images:

$$D_i^A = A \otimes F_i^A, \quad 1 \leq i \leq n^2$$

Feature Vector Extraction

- From each detail image D_i^A , compute summary statistics (e.g., mean, energy).
- Combine statistics into a feature vector for classification or recognition tasks.

Multi-Scale / Multi-Directional

It involves analyzing images at different scales (resolutions) and across various orientations (directions) to extract both fine details and broader/coarser structures. This technique is especially useful for capturing features that may vary in size and orientation within the image, such as **edges, textures, or patterns**. Often not stationary.

Stationary implies that the image has a homogeneous structure or pattern, and the same statistical characteristics (like intensity distribution, frequency content, or texture) can be observed across all parts of the image.

Scale Sensitivity

- **Texture description is highly scale-dependent.** To reduce scale sensitivity, a texture is **described in multiple resolutions** and appropriate scale selection improves discrimination.
- **Gabor filters** and **wavelets** are well suited for **MSMD** filtering.

Direction Sensitivity

- **Texture description is highly direction dependent.** Useful for detecting oriented patterns such as horizontal, vertical, or diagonal edges.
- **Wavelets:** Apply **directional filters** on different levels of the wavelet pyramid to extract MSMD features.
- **Gabor:** Gabor filters are directional / multi-scale too

Applications

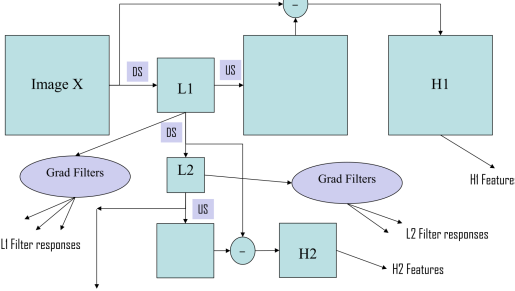
- Used in texture classification via:
 - Pyramid or tree-structured discrete wavelets
 - MSMD Gabor filter banks
- Typically outperform conventional texture characterization methods.

Wavelet-Based Texture Analysis

- A wavelet is a mathematical function used for image feature extraction and compression.
- It serves as a basis function localized in both frequency and space/time.
- Wavelet transforms are efficient and computationally lightweight. Can be implemented in both **spatial domain** and **transform domain**.
- Useful in MSMD texture analysis—e.g., moving from details to general features.

Wavelet Pyramid Construction

- Downsample (zoom out) to obtain a **low-pass version** of the image L_1 . Upsample and subtract to produce a **high-pass version** H_1 .
- L_i builds the **Gaussian pyramid**, while H_i builds the **Laplacian pyramid**.
- From each level of the pyramid, features can be extracted:
 - Use H_i for fine detail (edges, textures).
 - Use L_i for coarser structure.
 - Apply directional filters (e.g., gradient detectors) on L_i .
- For compression, lower pyramid levels may be retained.



Remarks

- Wavelets allow analysis of textures at multiple resolutions.
- Features be adaptive or global/local depending on design.
- The Gaussian levels are smooth enough that when you apply an oriented derivative you get clear edge/texture orientation at that scale, without swamping it in noise.
- Classify based on number of edges, corners, and shapes in general.

Motion Detection and Optical Flow

Image motion analysis and object tracking combine two separate but interrelated components:

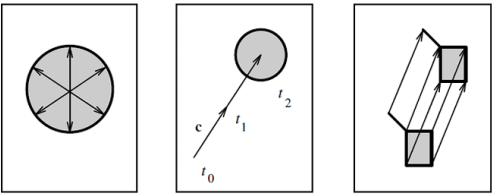
- Localization and representation of the object of interest
- Trajectory filtering and data association

Three main groups of motion-related problems:

1. Motion Detection
2. Moving object detection/localization (direction? speed?)
3. Derivation of 3D object properties

Basic Assumptions

- **Maximum velocity:** Speed of moving objects are not high compared to FPS rate
- **Small acceleration:** Acceleration is close to 0
- **Common motion:** Objects are rigid and a single motion vector is application on all parts of the object
- **Mutual correspondence:** Rigid objects exhibit stable pattern points.



To generalize, image motion analysis and object tracking combine two separate by interrelated components:

- Localization and representation of object of interest.
- Trajectory filtering and data association.

Differential Motion Analysis Methods

Assuming a stationary camera position and constant illumination, taking the difference in frames across time makes motion detection possible.

$$d(i, j) = \begin{cases} 0 & \text{if } |f_1(i, j) - f_2(i, j)| < \epsilon \\ 1 & \text{otherwise} \end{cases}$$

$d(i, j) = 1$ if

1. $f_1(i, j)$ is a pixel on a moving object, $f_2(i, j)$ is a pixel on the static background,
2. $f_1(i, j)$ is a pixel on a moving object, $f_2(i, j)$ is a pixel on another moving object.
3. $f_1(i, j)$ is a pixel on a moving object, $f_2(i, j)$ is a pixel on a different part of the same moving object.
4. Noise, inaccuracies of stationary camera positioning, etc

Cumulative Difference Images contain information about motion direction and other time-related motion properties, as well as about slow motion and small object motion.

$$d_{cum}(i, j) = \sum_{k=1}^n \frac{k}{n} \cdot |f_1(i, j) - f_k(i, j)|$$

Recent images are given higher weights to reflect the importance of current motion and specifies the location of the current object.

Issues

Motion field and Aperture problems.

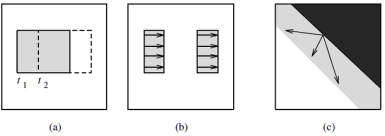


Figure 16.4: Problems of motion field construction. (a) Object position at times t_1 and t_2 . (b) Motion field. (c) Aperture problem—ambiguous motion. © Cengage Learning 2015.

Solution: Detect moving edges.

Moving Edges Detection

Roberts, Laplace, Prewitt and Sobel (or any edge detectors) can identify the spatial edges. The temporal gradient can be approximated using the difference image, and the logical AND can be implemented through multiplication.

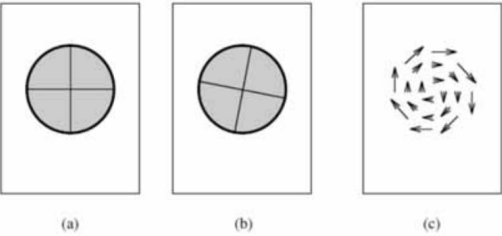
$$d_{med}(i, j) = S(i, j) \cdot D(i, j)$$

where $S(i, j)$ is the edge magnitude of one of the frames and $D(i, j)$ is the absolute difference image.

Optical Flow Algorithm

Optical flow reflects image changes due to motion during a time interval dt , and the optical flow field is the velocity field that represents the 3D motion of object points across a 2D image. **It represents only those motion-related intensity changes in the image that are required in further processing.**

- Necessary precondition of subsequent higher-level processing that can solve motion-related problems.



Optical flow. (a) Time t . (b) Time $t + dt$. (c) Optical flow.

Assumptions: (1) The observed brightness of any object point is constant over time. (2) Nearby points in the image plane move in a similar manner (velocity smoothness constraint)

Algorithm:

- Gray level of (x, y) at time t , $I(x, y, t)$
- Expand using first-order Taylor approximation:

$$I(x + dx, y + dy, t + dt) \approx I(x, y, t) + I_x u + I_y v + I_t$$

- Substituting into the brightness constancy assumption:

$$I_x dx + I_y dy + I_t dt = 0$$

This is the **optical flow constraint equation**. Expresses the relationship between spatial and temporal derivatives of the image intensity and the velocity components.

- If we can approximate I_x , I_y , I_t , then c can be determined:

$$-I_t dt = I_x dx + I_y dy = \Delta I \cdot c$$

where ΔI is a 2D image gradient.

- Problem converted to minimization of squared error term

$$E^2(x, y) = (I_x dx + I_y dy + I_t)^2 + \lambda(\mu)$$

- Finally, reduces to solving the following:

$$(\lambda^2 + I_x^2)u + I_x I_y v = \lambda^2 \bar{u} - I_x I_t$$

$$I_x I_y u + (\lambda^2 + I_y^2)v = \lambda^2 \bar{v} - I_y I_t$$

where λ is the Lagrange coefficient, and e.g. I_x^2 denotes partial derivatives squared as error terms. \bar{u} , \bar{v} are mean values of the velocity in directions x and y in some neighbourhood of (x, y) .

If more than two images are to be processed, efficiency may be increased by using the results of one iteration to initialize the current image pair in the sequence.

Issue: One equation, two unknowns.

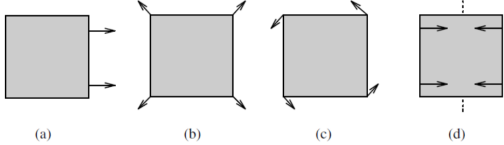
- **Lucas–Kanade:** for each pixel, assume constant motion field, and hence optical flow (u, v) is constant within a small neighborhood W . You now have multiple equations for 2 unknowns (u, v) .

Output: A dense 2D vector field $(u(x, y), v(x, y))$ representing motion at each pixel.

- This helps linearise intensity changes over small motion.

Optical Flow Applications

- **Translation at constant distance from the observer** (represented as a set of parallel motion vectors)
- **Translation in depth relative to observer** (forms a set of vectors having a common focus of expansion)
- **Rotation at a constant distance about the view axis** (results in a set of concentric motion vectors)
- **Rotation of planar object perpendicular to view axis** (forms one or more sets of vectors starting from straight line segments)



Object Tracking

An algorithm detects objects and then tracks their movements in space or across **different camera angles**. Object tracking can identify and follow **single** or **multiple** objects in videos.

There are four stages of tracking:

1. Target Initialization (point tracking via Lucas-Kanade)
2. Appearance Modelling
3. Motion Estimation
4. Target Positioning

Interest/Anchors/Features Points

Good for object tracking, Harris Corner Detector, LK PT Algo.

Background Modeling

Goal: Identify moving objects by comparing the current frame to a background reference.

- Compare each frame with an empty/background scene.
- The first frame is often used as the background.
- Subtracting reveals blobs corresponding to moving objects.

Challenges: Environmental motion (e.g., wind, lighting), Camera shake or drift, Any dynamic change in the background

Dynamic Background Modeling

- Background is updated over time using a history of K frames. Median (or mean) intensity at each pixel location is computed. (This helps accommodate gradual lighting or background changes.

Median Filtering

1. For each pixel, compute the median of intensity values over the past K frames.
2. Acquire frame $K + 1$ and compute the difference with the current median background.
3. Apply thresholding to suppress noise and highlight significant changes.
4. Use morphological operations (e.g., blurring, hole filling) to clean up noise.
5. Update the median background with the new frame, discarding the oldest.

6. Repeat for subsequent frames.

Mixture of Gaussians (MoG)

- Each pixel is modeled as a **mixture of Gaussians** over time. Updated per pixel based on intensity history.
- Some Gaussians represent the **background**, others the **foreground**. The algorithm classifies pixels based on which Gaussian component the value matches.

Advantages:

- If lighting is constant, one Gaussian per pixel is sufficient.
- If lighting changes slowly, an adapting Gaussian can handle variation.
- Can distinguish between background motion (e.g., tree branches) and true foreground.

Motion Models to Aid Tracking

Goal: Estimate and predict the state (e.g., position) of an object over time in the presence of noise.

- Tracking:** Determine position at time $t + \Delta t$ in 2D/3D space.
- Control theory:** Estimate system state from noisy observations (state space model).
- We model hidden state \mathbf{x} and observe measurement \mathbf{z} (both are feature vectors).

Using Observations in Tracking:

- Sequence of observations ($\mathbf{z}_1, \mathbf{z}_2, \dots$) improves estimation of true state \mathbf{x} .
- Observation \mathbf{z}_k gives estimate of \mathbf{x}_k , assuming known motion model.
- Estimate of \mathbf{x}_k helps predict:
 - Next state: \mathbf{x}_{k+1}
 - Next observation: \mathbf{z}_{k+1}
- Observe \mathbf{z}_{k+1} and refine estimate of \mathbf{x}_{k+1}

Kalman Filters

$$\mathbf{x} = [x, y, \dot{x}, \dot{y}]^T$$

is a possible state vector, consisting of the position and the velocity in 2D space.

Purpose: Predict and update object states over time using a mathematical model and noisy measurements.

- Kalman filters provide **real-time** tracking, both state and the uncertainty of the object, updating with observed data.
- Only a small, local search is needed around the predicted silhouette position.
- Edge search is conducted along predicted edge normals for max contrast. If no good match is found (low contrast), the state is not updated (no observation).
- Robust to partial occlusion — tracking can still continue.

Limitations:

- Struggles under high noise or clutter.
- Non-linear motion, sudden changes in direction.
- Assumes unimodal, Gaussian-distributed state uncertainty — not always valid.

Kalman Case Study

- Used for vehicle tracking, estimating positions of surrounding objects like pedestrians or cars.
- Help in localization by predicting the vehicle's position and updating with data from GPS, cameras and LiDAR.
- Combine various information from sensors to create a reliable model of the environment around the vehicle.

Particle Filter

Probabilistically derived samples that provide an empirical description of what is and not likely. **Motivation:** Handle complex, non-Gaussian, or multi-modal tracking problems that Kalman filters cannot + real-time.

- Uses a set of weighted samples (particles) to represent the posterior distribution of the state.
- Particles are propagated and weighted according to observations and a motion model.
- Does not assume Gaussian or linear dynamics.

Steps:

- Initialize: Use a weighted sample set $S_{t-1} = \{(s_{t-1}^i, \pi_{t-1}^i)\}$.
- Resample: Select particles based on importance sampling.
- Predict: Propagate particles using motion model (e.g., $s_t^i = A_{t-1} s_{t-1}^i + w_{t-1}$).
- Correct: Use observation z_t to update weights $\pi_t^i = p(z_t | s_t^i)$.
- Normalize: Ensure $\sum_i \pi_t^i = 1$.
- Estimate: Compute state as weighted sum:

$$\hat{x}_t = \sum_{i=1}^N \pi_t^i s_t^i$$

Computer Vision & Deep Learning

Components of a Learning Agent

A learning agent interacts with environment and improves performance over time through feedback and learning.

Learning Loop:

- Agent receives input via sensors from the environment.
- Performance element selects and issues an action via actuators.
- Environment responds and provides new observations.
- Critic evaluates the result and gives feedback.
- Learning element adjusts internal models.
- Problem generator may introduce new goals or exploration.

Artificial Neural Network

- Turing Equivalent Universal Computers
- Massive parallelism makes them very efficient
- Can learn and generalise from training data – no need for subtle design.
- Fault and noise tolerant
- Can do anything a symbolic/logic system can do, good at dealing with semi structured and unstructured data.

Delta Rule (Widrow-Hoff Rule)

Purpose: Update weights to minimize prediction error.

$$\Delta w_i = \eta(y - \hat{y})x_i, \quad w_i \leftarrow w_i + \Delta w_i$$

Where:

- η : learning rate, x_i : input feature
- y : true label, $\hat{y} = \mathbf{w}^T \mathbf{x}$: prediction

Learning

$$\Delta w^{(n)} = -\eta \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} + \alpha \Delta w^{(n-1)} + \beta r$$

- η : learning rate, β : random term, r : random gaussian
- α : momentum coefficient (typically between 0 and 1)
- $\Delta w^{(n-1)}$: previous weight update
- Avoid outputs of 0 or 1 to prevent vanishing gradients.

Intuition:

- Momentum adds inertia to the weight updates.
- Helps accelerate learning in consistent directions.
- Reduces oscillations in areas with steep gradients.
- Can escape shallow local minima.
- If $\alpha = 0$: reduces to standard gradient descent and as $\alpha \rightarrow 1$: update incorporates more past direction history.

Multi Layer Perceptron

A multi-layer perceptron is formed by combining many single-layer perceptrons. This nonlinearity allows the model to learn more complex functions.

Layer Activation

Note: When implementing custom functions/layers, make sure they are **differentiable** (to calculate their gradients). For one hidden layer,

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial f_2} \cdot \frac{\partial f_2}{\partial W}$$

$$\frac{\partial L}{\partial U} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial f_2} \cdot \frac{\partial f_2}{\partial \hat{f}_1} \cdot \frac{\partial \hat{f}_1}{\partial f_1} \cdot \frac{\partial f_1}{\partial U}$$

Bias and Variance Tradeoff

A model with a **high bias** makes more assumptions, and unable to capture the important features of our dataset. A high bias model cannot perform well on new data. To reduce, (1) increase the input features (2) decrease regularization term (3) use more complex models.

A model that shows **high variance** learns a lot and perform well with the training dataset, but does not generalize well with the unseen dataset. To reduce, (1) decrease the input features (2) do not use more complex model (3) add noise/increase training data (4) increase regularization term (for NN, dropout - randomly setting activations to 0).

Activation Functions Overview

- 1. Sigmoid:**

$$\text{Sigmoid}(x) = \frac{1}{1 + e^{-x}} \quad \frac{\partial y}{\partial x} = (1 - \text{Sig}(x)) * \text{Sig}(x)$$

Output in (0,1); binary classification. Smooth, differentiable.

- 2. Tanh (Hyperbolic Tangent):**

$$\tanh(x) = \frac{1 - e^{-x}}{1 + e^{-x}} \quad t(x) = 2\text{Sig}(2x) - 1$$

Output in (-1,1); better for zero-centered data.

- 3. Step Function:**

$$\text{step}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

Non-differentiable; used in early perceptrons.

- 4. ReLU (Rectified Linear Unit):**

$$f(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$$

Efficient but may suffer from dead neurons when majority of activations are 0.

- 5. Piecewise Linear:**

$$f(x) = \begin{cases} 1 & x \geq 0.5 \\ x + 0.5 & -0.5 \leq x \leq 0.5 \\ 0 & x \leq -0.5 \end{cases}$$

Combines step and linear behavior.

- 6. Sign Function:**

$$f(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases}$$

Used in classic perceptrons; non-differentiable.

- 7. Softmax**

$$\text{Softmax}(z)_i = \frac{e^{z_i}}{\sum_j z_j}$$

- Used for clustering/classification.

Common Problems

Vanishing/Exploding gradient: With saturating activations (like sigmoid), the gradient decreases exponentially as we propagate down (due to small derivatives). Consequently, change in weights becomes very small, causing convergence to be slow, if not impossible. To solve this: (1) proper weight initialisation (2) use non-saturating activations e.g. ReLU (Leaky ReLU) (3) Feature scaling (4) gradient clipping (of).

Stochastic Gradient Descent

Gradient descent allows us to compute an estimate of the gradient at each point and move a small amount in the steepest downhill direction, until we converge on a point with minimum loss. If the loss surface is convex, we will always arrive at the global minimum.

$$w_0 = w_0 - \alpha \frac{1}{m} \sum_{j=1}^m (w_0 + w_1 x_1^{(j)} + \dots - y_j)$$

$$w_i = w_i - \alpha \frac{1}{m} \sum_{j=1}^m (w_0 + w_1 x_1^{(j)} + \dots - y_j) \times x_i^{(j)}$$

Stochastic gradient descent updates weights by considering one point at a time, which is faster but does not guarantee global minima due to its randomness.

ADAM Training Algorithm

Core Idea:

- Combines advantages of AdaGrad and RMSProp.
- Individual **adaptive** learning rates for each parameter (vs SGD whose LR is fixed).
- Uses estimates of the first moment (mean) and second moment (uncentered variance) of gradients.

Advantages:

- Straightforward
 - Computationally efficient
 - Low memory usage
 - Invariant to gradient rescaling
 - Works well on noisy/sparse gradients
- Suitable for non-stationary objectives
 - Handles large-scale data/parameters
 - Intuitive hyper-parameters; minimal tuning

Practical Considerations for Training

1. Zero-centering of features helps with efficient training and more balanced gradients.
2. Without normalization, model perform large updates to one weight compared to the others, causing the gradient descent trajectory to oscillate back and forth along one dimension, taking more steps to reach the minimum.
3. Initialize weights randomly. If all units of the net have the same initial parameters, then a deterministic learning algorithm applied to a deterministic cost and model will constantly update both of these units in the same way.
4. Start with large learning rate η and reduce

$$\eta(t) = \frac{\eta(1)}{t}, \quad \eta(t) = \frac{\eta(0)}{1 + t/\tau}$$

5. Online/Stochastic Learning
 - Updates model weights after each data point.
 - Suitable for streaming or large datasets.
 - Fast, low memory usage, but updates may be noisy.
6. Batch Learning
 - Updates weights after seeing the entire dataset.
 - More stable and accurate convergence.
 - Requires full dataset in memory; slower updates.
7. Mini-Batch
 - Combines best of both worlds; less noisy, efficient computation, smooth convergence but requires finding the "correct" size

Loss Function Overview

The “**loss function**” is a function that measures how badly the neural network did by comparing its hypothesis output, \tilde{y}_i , to a reference label \bar{y}_i :

$$\ell(\tilde{y}_i, \bar{y}_i) = \text{“badness” score for the NN output } \tilde{y}_i$$

The average of $\ell(\tilde{y}_i, \bar{y}_i)$ over the entire training corpus tells you how badly the neural net is doing on the whole training corpus.

$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^n \ell(\tilde{y}_i, \bar{y}_i)$$

How do we choose the loss function?

- We want it to measure how badly we’re doing.
- For an image de-noising task/regression: we want it to measure the difference between the target images, and the neural net outputs.
 - For a classification task: we want it to measure something like the percentage error rate on the training corpus.

Summary

- **Mean Squared Error (MSE):**
 - $\bar{y}_i = [\bar{y}_{i1}, \dots, \bar{y}_{iV}]$ is the expected output.
 - $\tilde{y}_i = [\tilde{y}_{i1}, \dots, \tilde{y}_{iV}]$ is the predicted output.

- Measures squared difference per component:

$$\ell_{\text{MSE}}(\tilde{y}_i, \bar{y}_i) = \frac{1}{V} \sum_{v=1}^V (\tilde{y}_{iv} - \bar{y}_{iv})^2$$

- Overall MSE:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n} \sum_{i=1}^n \ell_{\text{MSE}}(\tilde{y}_i, \bar{y}_i)$$

- **Zero-One Loss:**
 - Assigns a 1 if the prediction is wrong, 0 otherwise:

$$\ell_{\text{ZO}}(\tilde{y}_i, \bar{y}_i) = \begin{cases} 0 & \text{if } \tilde{y}_i = \bar{y}_i \\ 1 & \text{otherwise} \end{cases}$$

- Overall error rate:

$$\mathcal{L}_{\text{ZO}} = \frac{1}{n} \sum_{i=1}^n \ell_{\text{ZO}}(\tilde{y}_i, \bar{y}_i)$$

- Not differentiable — limits use in gradient-based optimization.
- **Cross Entropy:** Let:

$$y_{ci} = \begin{cases} 1 & \text{if } c \text{ is correct class for token } i \\ 0 & \text{otherwise} \end{cases}$$

and \tilde{y}_{ci} = Estimated $P(\text{class } c \mid \vec{x}_i)$
Cross-entropy loss:

$$\ell_{\text{CE}}(\tilde{y}_i, \bar{y}_i) = - \sum_{c=1}^V y_{ci} \log \tilde{y}_{ci}$$

Averaged over dataset:

$$\mathcal{L}_{\text{CE}} = \frac{1}{n} \sum_{i=1}^n \ell_{\text{CE}}(\tilde{y}_i, \bar{y}_i)$$

Differentiable and works well for classification.

- **Binary Cross Entropy (BCE):** Used when the output class is binary (0 or 1). Let:

$$y_i = \begin{cases} 1 & \text{if token } i \text{ is of class } +1 \\ 0 & \text{otherwise} \end{cases}$$

where \tilde{y}_i = Estimated $P(\text{class of token } i \text{ is } +1 \mid \vec{x}_i)$
Binary cross-entropy loss:

$$\ell_{\text{BCE}}(\tilde{y}_i, y_i) = -y_i \log \tilde{y}_i - (1 - y_i) \log(1 - \tilde{y}_i)$$

Alternatively:

$$\ell_{\text{BCE}}(\tilde{y}_i, y_i) = \begin{cases} -\log \tilde{y}_i & \text{if } y_i = 1 \\ -\log(1 - \tilde{y}_i) & \text{if } y_i = 0 \end{cases}$$

Overall BCE loss:

$$\mathcal{L}_{\text{BCE}} = \frac{1}{n} \sum_{i=1}^n \ell_{\text{BCE}}(\tilde{y}_i, y_i)$$

Interpretation:

- Measures how far predicted probability is from actual binary label.
- A low loss means the predicted probability is confident and correct.

Applicability:

- Can be applied in both binary and multi-class (via one-vs-rest) classification tasks.

ANN Parameters and Overfitting

Key factors influencing overfitting: (1) Number of training epochs (2) Number of hidden neurons (3) Training algorithm (4) Constraints (i.e., training data)
To avoid overfitting, apply the constraint:

$$\#C \geq kF^0, \quad k = 4, 5, \dots, 10$$

- $\#C$: number of constraints (training samples)
- F^0 : degrees of freedom (number of trainable parameters)

Notes

- Universal approximation: A single hidden layer can approximate any function, but may require an exponentially large number of neurons.
- Deeper networks approximate the same functions more compactly, enabling better generalization and scalability.
- Deep learning does automatic feature extraction (better) but complicated and computationally heavy.
- Having more features may help you to have a capable pattern recognition system using simpler classifiers. However, features should be meaningful, effective, and not correlated.
- Curse of dimensionality, number of features above a threshold will decrease the performance of your model.

Regularization

Two ways to address overfitting: (1) reduce the number of features (2) reduce magnitude of weights (regularization)
L1-norm, Lasso Regression

$$J(w) = \frac{1}{m} \sum_{i=1}^m E(\hat{y}^i, y^i) + \frac{\lambda}{2m} \sum_{i=1}^n |w_i|$$

L2-norm, Ridge Regression

$$J(w) = \frac{1}{m} \sum_{i=1}^m E(\hat{y}^i, y^i) + \frac{\lambda}{2m} \sum_{i=1}^n |w_i|^2$$

where E is any difference function.

Note:

1. Too large λ can cause your model to underfit. Regularization assure that your model performs better on unseen data, sacrificing performance on training data. (higher bias but low variance)
2. Lasso shrinks less important feature’s coefficient to zero, removing some altogether. Works well for feature selection in case we have a huge number of features.
3. Ridge penalizes larger parameters, attempting to pull all parameters towards small values.
4. **Dropout:** because nodes are randomly dropped, the network cannot depend heavily on any single neuron or input feature to make predictions. Instead, it must distribute the learning across all neurons and features, shrinking the model.

Batch Normalization

Purpose:

- Normalize activations across each mini-batch to stabilize learning before passing to the next layer H_{m+1}
- Reduces internal covariate shift — i.e., shifting of activation distributions.

- Accelerates training and allows higher learning rates. Makes deep networks easier to train by smoothing the optimization landscape.
- Helps coordinate updates across deep layers.

Formula:

$$A'(H_m) = \frac{A(H_m) - \mu[A(H_m)]}{\sigma[A(H_m)]}$$

- A : activation function; H_m : input to layer m
- μ : batch mean; σ : batch standard deviation

Fine-Tuning and Transfer Learning

Transfer learning involves using a pre-trained model’s features and retraining only the final layers for a new task. **Fine-tuning** adjusts the weights of the entire model (or a subset of layers) to better adapt it to the new task. Fine-tuning provides more flexibility but requires more data and computation.

- Earlier layers are reused, and later layers are fine-tuned. Early layers capture **general, transferable features** (e.g., edges, textures). Later layers are more task-specific and often need more fine-tuning.
- The extent to which the parameters of early layers versus the last layers are changed depends on the task, data size, and model architecture. In practice, earlier layers change less than later layers during fine-tuning.

Clustering

At the beginning, weights, W , are random. We present the 1st data sample/ feature vector to the network, then one of the neurons/ cluster has the maximum, and will be the winner.

- Then weight vector of that neuron/ cluster would be updated and replaced by the scaled average of the old weight vector and the feature vector of the 1st data sample:

$$W_i^{new} = kW_i^{old} + (1 - k)X_j$$

- We assumed that the i_{th} neuron has won and shown the maximum output for the j_{th} data sample, X_j , and now its weights, W_i should be updated. $1 \geq k \geq 0$, can be constant or variable but usually $k \gg 0$.

LSTM: Long Short-Term Memory Networks

Purpose: Designed to learn long-term dependencies and avoid vanishing gradient problems in sequential data.

Key Components:

- **Forget gate:** Controls what to discard from the cell state.

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

- **Input gate:** Decides what new information to store.

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i), \quad \tilde{C}_t = \tanh(W_C[h_{t-1}, x_t] + b_C)$$

- **Cell state update:**

$$C_t = f_t C_{t-1} + i_t \tilde{C}_t$$

- **Output gate:** Controls what to output.

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o), \quad h_t = o_t \cdot \tanh(C_t)$$

Use LSTM when:

- Data has sequential dependencies.
- You need to capture long-term context.
- Standard RNNs fail due to vanishing/exploding gradients.

AlexNet Architecture

AlexNet is a deep convolutional neural network.

Architecture:

- **Input:** $227 \times 227 \times 3$ RGB image
- **Conv1:** 96 filters, 11×11 , stride 4
- **Max Pool:** 3×3 , stride 2
- **Conv2:** 256 filters, 5×5
- **Max Pool:** 3×3 , stride 2
- **Conv3–5:** 384, 384, 256 filters, 3×3
- **Max Pool:** 3×3 , stride 2
- **FC1, FC2:** 4096 units each, with ReLU + dropout
- **FC3:** 1000-way softmax (ImageNet classes)

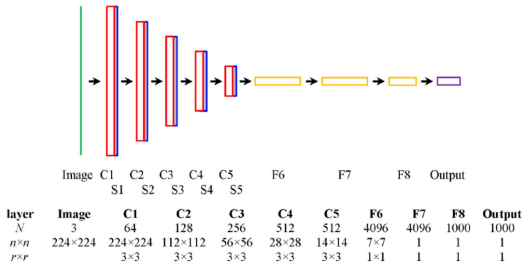
Notable Features:

- Uses ReLU activations, Dropout in fully connected layers
- Max pooling (dimension reduction, translation invariance, noise robustness and controls overfitting). More stable, efficient and generalizable.

When to Use:

- As a baseline for image classification
- For moderate-scale datasets (e.g., CIFAR, Caltech101)
- For transfer learning or feature extraction
- When lower compute cost is needed

VGGNet Architecture (VGG-16)



- Uses only 3×3 convolutions and 2×2 max pooling.
- Depth: 16 or 19 layers; VGG-16 is most common.

VGG-16 Layer Structure:

- Input: $224 \times 224 \times 3$ RGB image
- Conv Block 1: $2 \times \text{Conv}(64, 3 \times 3) + \text{MaxPool}$
- Conv Block 2: $2 \times \text{Conv}(128, 3 \times 3) + \text{MaxPool}$
- Conv Block 3: $3 \times \text{Conv}(256, 3 \times 3) + \text{MaxPool}$
- Conv Block 4: $3 \times \text{Conv}(512, 3 \times 3) + \text{MaxPool}$
- Conv Block 5: $3 \times \text{Conv}(512, 3 \times 3) + \text{MaxPool}$
- FC: $\text{FC}(4096) \rightarrow \text{FC}(4096) \rightarrow \text{FC}(1000\text{-way softmax})$

Features:

- All conv layers use ReLU activation, unit stride.
- Max pooling: 2×2 , stride 2
- Dropout applied in FC layers

When to Use:

- Image classification and transfer learning
- When depth and simplicity are preferred
- Note: computationally heavy due to many parameters

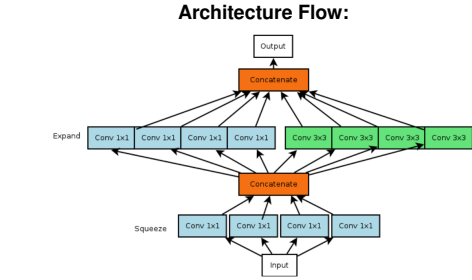
SqueezeNet Architecture

- Designed for model efficiency with fewer parameters.
- Achieves AlexNet accuracy with **fewer parameters** (x50).
- Enables deployment on mobile and embedded devices.

Key Component: Fire Module

- **Squeeze layer:** 1×1 convolutions to reduce input depth.

- **Expand layer:** Mixture of 1×1 and 3×3 filters to increase output depth.



- Conv(96 filters, 7×7 , stride 2) + MaxPool
- **Fire modules:** Fire2 to Fire9
- MaxPool after Fire4 and Fire8
- Final Conv(1000 filters, 1×1) + Global Avg Pooling + Softmax

Efficiency Strategies:

- Replace 3×3 filters with cheaper 1×1 where possible.
- Downsample late to retain spatial resolution in early layers.
- Use fewer filters in squeeze layers to reduce input size to expand layers.

When to Use:

- Mobile, embedded, or edge deployment.
- Real-time inference with limited compute.
- Applications needing small models without sacrificing too much accuracy.

Inception V3 Architecture

- Deep CNN model for efficient image classification.
- Optimized for both accuracy and computational cost.
- Used in large-scale image datasets and real-world classification tasks.

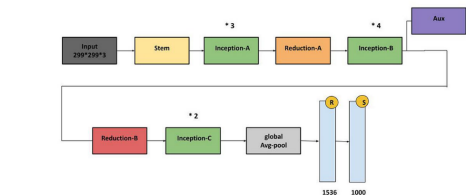
Architecture Highlights:

- Utilizes multiple **Inception modules**:
 - Parallel convolutional branches with 1×1 , 3×3 , and 5×5 filters.
 - Reduces width and depth bottlenecks with efficient filter use.
- Includes **factorized convolutions** (e.g., $n \times n \rightarrow 1 \times n + n \times 1$) to reduce computation.

Improvement Features:

- Factorized convolutions
- Auxiliary classifiers (for regularization)
- Label smoothing (to reduce overconfidence)

Architecture Flow:



- Initial Convolution + Pooling Layers
- Factorized Convolutions
- Multiple Inception Modules
- Auxiliary Classifier Branch
- Global Average Pooling

- Dense + Softmax Output (e.g., 1000 classes)

When to Use:

- When both accuracy and speed matter (e.g., on GPUs/TPUs)
- For large-scale classification tasks (e.g., ImageNet)
- When training time needs to be balanced with model depth

CNN Architecture Comparison

| Model | Pros | Cons |
|------------|-----------------------|------------------|
| AlexNet | Simple, fast to train | Outdated, bulky |
| VGG-16 | Deep, uniform design | Very large, slow |
| SqueezeNet | Tiny, efficient | Lower accuracy |
| Inception | Efficient, accurate | Inflexible |

Object Detection

Characterized as (1) Recognition (2) Localization.

Challenges in Object Detection

- Severe Lighting Conditions (overexposure, shadows, glare, contrast changes, color variations, and unpredictable environmental variations.)
- Scale/Aspect Ratio (Objects vary in size and orientation, harder to detect and localize objects at different scales and orientations in images)
- Occlusion/Background (Targets hidden or confused with clutter)

Intersection over Union (IoU)

- IoU measures the overlap between a predicted bounding box b and a ground truth box g .
-

$$\text{IoU}(b, g) = \frac{b \cap g}{b \cup g}$$

Usage:

- Classification: predict the class of the object.
- Bounding Box Regression: tightest box position that enclose object.
- IoU is used to assign a label:

$$y_u = \begin{cases} x & \text{if } \text{IoU}(b, g) \geq u \\ 0 & \text{otherwise} \end{cases}$$

where u is the IoU threshold (e.g., 0.5).

Regression Sample Selection:

- Training pairs are selected with:

$$\mathcal{G} = \{(g_i, b_i) \mid \text{IoU}(b_i, g_i) \geq u\}$$

Interpretation:

- Higher IoU thresholds = more accurate box predictions.
- Lower thresholds = more matches, poor localization.

Region-Based CNN

Problem: Object localization in an image. **Brute Force**

Solution: Sliding window with various rectangle sizes.

Drawbacks: Inefficient, slow, generates millions of windows.

R-CNN Solution:

- Uses category-independent **region proposals** (e.g., Selective Search) to narrow search space ($\sim 2k$ regions from millions). Over-segment + recursively merge based on texture, color, size and geometry.

- Extracts CNN features from each region (using pretrained CNNs like AlexNet or VGGNet). Penultimate network layer is used as feature representation for each proposal.
- Feeds CNN output to class-specific **SVMs** for classification.
- Refines bounding boxes using regressors.

Training:

- CNN is pretrained on ImageNet, then fine-tuned on proposal regions.
- Classification is a $(K + 1)$ -way problem: K object classes + 1 background class.
- Proposals with $\text{IoU} > 0.5$ vs ground truth = positives; rest = negatives. Not sufficient to just use ground truth, need to augment more data since CNN are data hungry.
- Training batch follows 1:3 positive:negative ratio.
- Loss function (cross-entropy):

$$L_{\text{cls}}(h(x), y) = -\log h_y(x)$$

where x is the region, y is the true label, $h(x)$ is the classifier output.

- After finetuning, the category-specific linear SVM classifiers and BB regressors are trained on the proposal features generated by the CNN.

Conclusion:

- **Multi-stage pipeline:** region proposal + CNN feature extraction + SVM + BB regressor.
- Accurate but slow, storage-heavy, and computationally inefficient. Each region is processed independently, even if the regions overlap.

SPP-Net

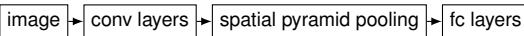
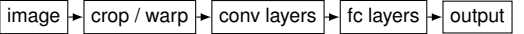
SPP-Net improved upon R-CNN by making the CNN model agnostic of **input image size**, drastically improving the bounding box prediction speed. However, SPP-Net inherits the **tedious multistage training procedure** of R-CNN.

Why the need for fixed-size input images?

The fixed size constraint is not because of the convolution layer but rather the Fully Connected (FC) layer. Convolution layers takes an image and produces a feature map proportional to a particular ratio (called the sub-sampling ratio) w.r.t the input image. For a FC layer, the input has to be a fixed-length vector. **Solution:** Replace the last pooling layer with a Spatial Pyramid Pooling (SPP) layer

Spatial Pyramid Pooling Layer

Unlike R-CNN, which crops proposals before CNN computation, the SPP-Net forwards the whole image through the CNN. Spatial pyramid pooling is then used to extract fixed-length features of the cropped feature maps associated with each proposal, which is then used as input to the FC layers for final prediction. This enables the sharing of expensive CNN computations among proposals, and the **whole image is only processed once**.



Maps the instance-wise features of arbitrary scale and size into a fixed-length vector. This was the first time that features from a convolutional feature map can be pooled over a spatial region to produce an instance-wise feature representation with good properties for instance recognition.

Fast R-CNN

Unlike the R-CNN and SPP-Net, the Fast R-CNN is trained end-to-end with a multitask loss, avoiding the tedious multistage training procedure.

- Forwards the whole image through the CNN to generate feature maps.
- Region of interest (RoI) pooling layer is used to extract a vector of fixed-length features for each object proposal.
- Two fully connected (FC) layers are used to make the final predictions:
 - classification probabilities for $K + 1$ classes
 - regression of four bounding box coordinates.

SPP vs Simplicity

RoI pooling operation is a simplified version of SPP. Instead of pooling a RoI to different spatial resolutions (1x1, 2x2, and 4x4), RoI pooling uses a single HxW resolution.

RoI Pooling Process

Given a RoI window of size h_{xw}, RoI pooling divides it into HxW subwindows, each of size h/H x w/W. Max-pooling is applied independently inside each subwindow to extract the largest feature value.

Channel Independence

The RoI pooling process is applied independently to each feature map channel. Although RoI pooling is simpler than SPM, it can still effectively extract a powerful feature representation for proposals of arbitrary size and scale from the precomputed convolutional feature maps.

Critical for Object Detection

The ability to adapt to proposals of different sizes and scales is crucial for object detection. **RoI pooling plays a vital role.**

Multitask Loss

Fast R-CNN jointly optimizes two tasks:

- Classification:** Predicts class y using a softmax function over $K + 1$ categories (including background).
- Bounding Box Regression:** Refines object localization with a loss applied only to foreground objects.

The combined multitask loss is:

- $$L = L_{\text{cls}}(h(x), y) + \lambda[y \geq 1]L_{\text{loc}}(f(x), \mathbf{b}), \mathbf{g})$$
- $L_{\text{cls}}(h(x), y) = -\log h_y(x)$ is the cross-entropy loss. $h(x)$ is $K + 1$ dimensional estimate of the posterior distribution over classes, i.e., $h_k(x) = p(y = k|x)$, where y is the class label and x is the image patch.

- L_{loc} is the bounding box regression loss (e.g., Smooth L1).
- λ balances classification and localization losses.
- The indicator $[y \geq 1]$ ensures localization loss is applied only for object classes, not background.

Bounding Box Regression

Bounding box $\mathbf{b} = (b_x, b_y, b_w, b_h)$ contains the coordinates of an image patch. The goal is to regress \mathbf{b} to the ground truth box \mathbf{g} using a function $f(x, \mathbf{b})$. The regression loss is:

$$L_{\text{loc}}(\mathbf{a}, \mathbf{b}) = \sum_{i \in \{x, y, w, h\}} \text{smooth}_{L_1}(a_i - b_i)$$

Smooth L1 Loss:

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2, & \text{if } |x| < 1 \\ |x| - 0.5, & \text{otherwise} \end{cases}$$

This combines L_1 and L_2 behavior, with less sensitivity to outliers and a stable gradient. To ensure **scale and location invariance**, the regression target is the distance vector:

$$\Delta = (\delta_x, \delta_y, \delta_w, \delta_h)$$

where

$$\delta_x = \frac{g_x - b_x}{b_w}, \quad \delta_y = \frac{g_y - b_y}{b_h}$$
$$\delta_w = \log\left(\frac{g_w}{b_w}\right), \quad \delta_h = \log\left(\frac{g_h}{b_h}\right)$$

Since bounding box regression usually performs minor adjustments on \mathbf{b} , the numerical values can be very small. This usually makes the regression loss much smaller than the classification loss. To improve the effectiveness of multitask learning, δ_x is normalized

$$\delta'_x = \frac{\delta_x - \mu_x}{\sigma_x}$$

where μ_x and σ_x are dataset-level mean and std.

Finetuning Strategy

Sampling Fast R-CNN samples N images, from which it extracts R RoIs per image. Choosing $N \ll R$ reduces the number of full CNN forward passes required. Despite concerns of **correlated** RoIs slowing training, the approach has proven effective. RoIs are typically sampled with a 25% positive and 75% negative ratio.

Backpropagation through RoI Pooling

Fast R-CNN improves over SPP-Net by enabling gradient backpropagation through the RoI pooling layer. Without this, convolutional layers below the RoI layer would remain untrained for detection, as is the case for SPP-Net.

- In RoI pooling, each output feature is the max-pooling of the corresponding sub window on the feature map, the back-propagation computations reduce to those of the max-pooling operation. Namely, the output gradient is only back-propagated to the position of largest max feature value in the subwindow.
- Applied to every RoI feature of every region proposal.

Conclusion

R-CNN \rightarrow **SPP-Net** \rightarrow **Fast R-CNN** illustrates the evolution of efficient object detection pipelines.

- Fast R-CNN improves detection speed and simplifies the training pipeline. Despite improvements, it still relies on external region proposal methods (e.g., selective search), limiting real-time efficiency.

YOLO: You Only Look Once

Single-stage object detection algorithm

- The input image is divided into $S_x S_y$ cells, and B predictions are made per cell x . A cell is considered responsible for an object if and only if the center of the ground truth bounding box of the object is located inside it.
- Each prediction consists of four bounding box coordinates x, y, w , and h and an objectness score $p(o = 1|x)$. The latter reflects the confidence that the predicted box includes an object and is ideally equal to the IoU between the predicted box and the object ground truth box.
- If no object exists in the cell, the objectness score should be 0. The confidence prediction for class k is then defined as

where $p(y = k|o = 1, x)$ is the class conditional probability of class k appearing in cell x given that the cell contains an object. However, a single set of C class conditional probabilities is shared by the B predictions of the cell, i.e., all predictions in a cell have the same conditional probabilities.

YOLOv3 Procedure

- Preprocessing:** The input image is resized to a fixed size to ensure consistency, only a single scale.

- Pass the Image through the CNN:** A variant of the Darknet architecture known as Darknet-53, which consists of 53 convolutional layers.
- Feature Extraction:** Extracts features at different scales. The network learns to identify various features of objects, like edges, textures, and shapes.
- Detecting Objects:** YOLOv3 makes detections at *three different scales*, improving its ability to detect small objects compared to its predecessors. It does this by applying detection kernels on three different sizes of feature maps at three different places in the network
- Using Anchor Boxes:** For each scale, YOLOv3 uses pre-defined anchor boxes (bounding box templates) to predict the presence of objects. The network predicts bounding box coordinates relative to these anchor boxes.
- Predicting Bounding Boxes and Class Probabilities:** For each object in an image, predict multiple BB and their corresponding objectness scores (likelihood of object's presence) and class probabilities (likelihood of object belonging to particular class).
- Applying Thresholds:** To reduce the number of detections, apply threshold to the objectness score.
- Non-Maximum Suppression (NMS):** To eliminate redundant boxes. NMS keeps only the boxes with the highest objectness scores and discards boxes that overlap significantly with higher-scoring boxes.
- Output:** Each bounding box prediction includes: (1) Coordinates (x, y, w, h) , (2) Confidence score $p(\text{object})$, (3) Class conditional probabilities $p(\text{class}_i|\text{object})$
The final score for class i is given by:

$$p(\text{class}_i|o = 1) = p(\text{class}_i|o = 1, x) \cdot p(o = 1|x)$$

Notes

- Throughout this process, YOLOv3 operates in a single forward pass of the neural network, making it remarkably fast and efficient for real-time object detection tasks. This single-pass approach is a key differentiator from other object detection methods that might require multiple passes or complex pipelines. It outperforms other object detection models like SSD and Faster R-CNN in terms of speed, though they might be a slightly more accurate
- The loss function in YOLOv3 is a combination of mean squared error for bounding box prediction and binary cross-entropy for class prediction and objectness prediction

Snippets

Listing 1: Ideal Low Pass Filter Implementation

```
def idealLowPass(M, D0):
    # Initializing the filter with zeros
    filter = np.zeros((M, M))

    # Normalized cut-off frequency is mapped to real index
    D0 = int(D0 * M)

    # Scanning through each pixel and setting values within D0
    filter[0:D0, 0:D0] = 1.0

    return filter

def highPass(M, D0):
    return 1 - lowPass(M, D0)
```

Listing 2: Optical Flow w/ Feature Points

```
# Building and showing the frames
t1 = np.zeros([5,5])
t1[4,0]=1
t2 = np.zeros([5,5])
t2[2,2] = 1
# Take difference
dtime = np.abs( t1-t2 )
# Roberts operator/filter
rob = np.array([ [ 1 , -1] , [-1 , 1] ])
# Filtering frames t1 and t2 with the Roberts operator, rob
edt1 = sig.convolve2d(t1, rob, mode="same")
edt2 = sig.convolve2d(t2, rob, mode="same")
# Results - Compute vector from (4, 0) to (2, 2)
result = np.abs(edt1-edt2) * dtime
```

Listing 3: Optical Flow w/o Feature Points

```
f1 = np.zeros( [5,5] )
f1[2:4,1] = 1
f2 = np.zeros( [5,5] )
f2[1:3,3] = 1
# configure h and v filters
h = np.array([ [ 1 , 1] , [-1 , -1] ])
v = h.T
# compute the time difference
dtime = np.abs( f1 - f2 )
# apply both filters on f1 and calculate the edge map
hedf1 = sig.convolve2d(f1, h, mode="same")
vedf1 = sig.convolve2d(f1, v, mode="same")
edgf1 = (hedf1**2 + vedf1**2)**0.5
# do the same for f2
hedf2 = sig.convolve2d(f2, h, mode="same")
vedf2 = sig.convolve2d(f2, v, mode="same")
edgf2 = (hedf2**2 + vedf2**2)**0.5
# calculate dt*|edgf1-edgf2| , that will show the edge displacement
dtime * np.abs( edgf1 - edgf2 )
# one vector from [3,1] to [2,3] , or 2 vectors from [3,1] to [2,3], and
# from [2,1] to [1,3]
```

Listing 4: Wavelet Implementation

```
# 4 line (f1,f2) and edge (f3,f4) detector filters
# to detect vertical (f1,f3) and horizontal (f2,f4) patterns
f1 = np.array([[0.5 , -1 , 0.5] , [1, -1, 1] , [0.5, -1 , 0.5]])
f2 = f1.T
f3 = np.array([[1 , 0.0 , -1] , [2, 0, -1] , [1, 0, -1]])
f4 = f3.T
levels = 3
for i in range(levels):
    M=img.shape
    # printing the pyramid level size
    print(M)
    # lowpass filtering by downsampling by 2
    ltmp = cv2.resize(img, (int(M[1]/2) , int(M[0]/2)))
    imgl.append( ltmp )
    # highpass filtering by image - lowpass filtered
    imgh.append( img - cv2.resize(ltmp, (M[1], M[0])) )
    # filtering the lowpass filtered image by f1 to f4
    fl1.append( cv2.filter2D(ltmp, cv2.CV_8UC3, f1) )
    fl2.append( cv2.filter2D(ltmp, cv2.CV_8UC3, f2) )
    fl3.append( cv2.filter2D(ltmp, cv2.CV_8UC3, f3) )
    fl4.append( cv2.filter2D(ltmp, cv2.CV_8UC3, f4) )
    img = ltmp
```