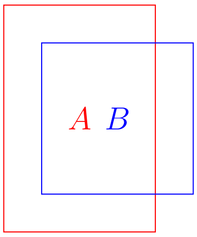
# CO316 - Computer Vision (60006)

## Lecture 1 - Introduction

Computer vision tries to build a system that can understand the world in a similar way to a human. At a higher level, the pipeline for vision consists of sensing an image or video, processing it, and then understanding it. For a human, the sensor is the eyes, and the processor is done by the primary visual cortex. On the other hand, a sensor can be a camera, or some form of medical imaging device, and the processor is the computer itself (and more importantly, the algorithm).

A **classification** problem has the goal of determining the **label** of what is in the picture. Classification is considered to be successful if one of the labels the algorithm predicts matches the true label. On the other hand, object **detection** attempts to draw a bounding box around an object (where are objects in the picture). We can quantify the success of detection based on the following. Consider the following, where the region in red is drawn by a human, and the region in blue is predicted by the algorithm;



We consider it a detection when the intersection over union (IoU) is above 0.5;

$$\text{IoU} = \frac{A \cap B}{A \cup B} > 0.5$$

Another more complex piece of information we can extract is to perform **image segmentation**, allowing us to draw contours for each object.

### Applications

Computer vision is used in our lives daily;

- **face detection**

  This can be noticed in most camera applications on modern smartphones, when a small box is drawn around faces. The algorithm first extracts **Haar** features from an image, and then determines (with these features) whether a region is a face or not.

  One example of these features is checking the contrast between the eyes and nose (horizontally); as the eyes tend to be quite dark in comparison. Another contrast is checked, this time between your eyes, as the nose tends to be brighter.
- **automatic number plate recognition**

  Automated barriers in parking lots can read number plates in order to calculate how long a car stays. Similarly, this can also be used to recognise building numbers, which is overlaid onto *Google Maps*, allowing for a large database of street numbers to be built in an automated fashion.
- **autonomous driving**
- **image style transfer**

  *Choi et al. StarGAN: Unified Generative Adversarial Networks for Multi-Domain Image-to-Image Translation* - used for changing features on inputs. Related to face motion capture (see *Face2Face*). Also see *DeepFake*.

- **Kinect**

  Works by taking a depth image, segmenting it into body parts, locating key points and building a skeleton.

- **design**

  See *OpenAI*'s *DALL-E*, combining NLP and computer vision by generating images based on the concepts of words in a sentence.

- **healthcare**

  Medical image analysis can be used for disease diagnosis. For example, identifying breast cancer lesions from mammograms.

## Lecture 2 - Image Formation

An image, in RGB format, can be represented as pixels, each being three numbers. A digital image is formed from a lighting source being reflected into an optics sensor (eyes, cameras, etc).

### Light

A **point light source** originates from a single location in space, such as a small light bulb, or the sun. This can be described with three properties; location, intensity, and the spectrum.

On the other hand, an **area light source** is more complex. For example, this could be a ceiling light; a rectangle of point lights.

### Reflectance

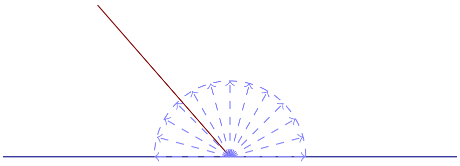
When light emitted from the source hits the surface of an object, it will be reflected. To describe this, we typically use the **bidirectional reflectance distribution function (BDRF)** to model this behaviour (where $\lambda$ is the wavelength, $L_r$ is the output power, and $E_i$ is the input power);

$$f_r(\underbrace{\theta_i, \varphi_i}_{\text{incident}}, \underbrace{\theta_r, \varphi_r}_{\text{reflected}}, \lambda) = \frac{\mathrm{d}L_r}{\mathrm{d}E_i}$$

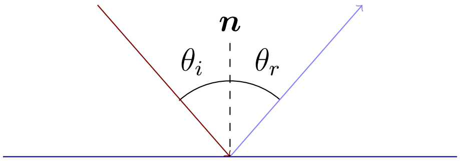
While this is a very general model, it is very complex.

As such, we can use **diffuse reflection**, where light is assumed to be scattered uniformly in all directions. This has a constant BRDF - this says that regardless of the incident or reflected directions, nor the wavelength, the power will be constant;

$$f_r(\theta_i, \varphi_i, \theta_r, \varphi_r, , \lambda) = f_r(\lambda)$$


rough surface

On the other hand, we can use **specular reflection** which performs reflections in a mirror-like fashion. The reflection and incident directions are symmetric with respect to the surface normal $\boldsymbol{n}$, such that $\theta_r = \theta_i$, with the same amount of power;



smooth surface

While these two are the **ideal** cases, the majority of cases, we see a combination of both of those, as well as **ambient** illumination. Ambient illumination accounts for general illumination which could be complicated to model. For example, these could be repeated reflections between walls (which would be very difficult to calculate), and we instead assume that there is some light that exists in the 3D space representing the room. Another example could be a distance source, such as the sky (which has atmosphere).

Combining these, we can use the **Phong** reflection model. This is an empirical model that describes how a surface reflects light as a combination of ambient, diffuse, and specular components.

### 'Duality' with Computer Graphics

Using the game engine to produce example images is useful, as we are able to directly obtain the labels of objects from the engine itself, as well as visual output. As such, we can use these images as training for a model, since we also have an associated label map. This synthetic data is complementary to time-consuming manual annotations.

### Optics and Sensors

Both our eyes and cameras work in similar ways, with a lens governed by the thin lens equation, where $f$ denotes the focal length of the lens, $u$ denotes the distance from the subject to the lens, and $v$ denotes the distance from the lens to the image;

$$\frac{1}{f} = \frac{1}{u} + \frac{1}{v}$$

Our eyes work by light rays being focused by the cornea and lens onto the retina, where vision begins with two neural cells. The **cone** cells are responsible for colour vision, and function in bright light. On the other hand, the **rod** cells have little role in colour vision, but function in dim light.

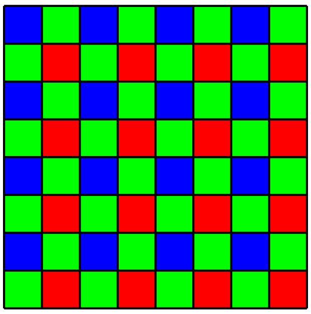
Humans have three types of cone cells (**trichromatic vision**), which have different response curves. The short cone cells respond to short wavelength lights (violet, blue), whereas the medium cone cells respond to medium wavelength lights (green), and long cone cells respond to long wavelength lights (red). Occasionally, there may be two cone cells, or four, which are referred to as **dichromacy** or **tetrachromacy** respectively.

Note that colours are not objective physical properties of light or electromagnetic wave (which have a physical property of wavelength). Colour is a subjective feature, dependent on the visual perception of the observer. Since **rod** cells are more sensitive to light, they are the primary source of visual information at night.

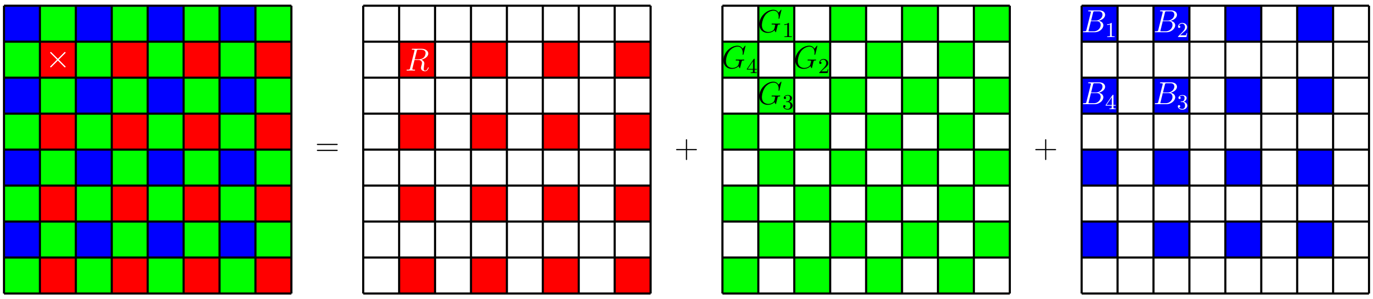
On the other hand, camera sensors have two common types;

- **CCD** (charged-coupled device) often used in handheld cameras
- **CMOS** (complementary metal-oxide semiconductor) used by most smartphone cameras

These sensors convert incoming light into electron charges, which are then read. **Bayer** filter arrays are a way to arrange RGB filters on sensors, half of which are green, and the remaining two quarters are red and blue. This mimics the human eyes, which are most sensitive to green light;



CMOS works by having sensors underneath each of these filtered portions, which can report an electrical signal. However, note that only one colour is available at each pixel (therefore the rest must be interpolated from the neighbours, by using bilinear interpolation; which simply averages the 4 neighbours). For example, consider the following pixel (denoted as a white cross);



$R_\times = R$ $\quad$ $G_\times = \frac{G_1+G_2+G_3+G_4}{4}$ $\quad$ $B_\times = \frac{B_1+B_2+B_3+B_4}{4}$

Note that the use of different filters, and this interpolation, can lead to slightly different colours between cameras.

### Image Representation

The earliest colour space was described in 1931 by CIE, by performing a colour matching experiment. In this experiment, an observer attempts to match different levels of red, green, and blue lights to match a target light. This allows for colours to be represented in 3D space, as $(X, Y, Z)$, corresponding to the different levels. Colours can also be represented on a 2D plane, by normalising brightness;

$$\begin{aligned} x &= \frac{X}{X+Y+Z} \\ y &= \frac{Y}{X+Y+Z} \\ z &= \frac{Z}{X+Y+Z} \\ &= 1 - x - y && \text{therefore redundant} \end{aligned}$$

Here $X, Y, Z$ are primary colours $(R, G, B)$, and $x, y$ are chromacity / colour after removing brightness. This is much easier to draw. However, this colour space, also known as the **gamut** of human vision, was invented before computer screens.

The sRGB (standard RGB) space was created by *HP* and *Microsoft* in 1996 for use on monitors, printers, and the internet.

| sRGB definition | $x$ | $y$ |
|---|---|---|
| red | 0.64 | 0.33 |
| green | 0.30 | 0.60 |
| blue | 0.15 | 0.06 |

This is represented by a triangle (which is a subset) in the gamut of human vision. As this is a subset, it cannot produce all the colours visible by the human eye.

There are other colour spaces, such as HSV, CMYK, and so on. Note that CMYK is a **subtractive** colour model, starting from white, whereas RGB is an **additive** colour model, where we start from black. There can also be an alpha channel in RGB, which represents transparency. In a greyscale image, the three components are equal, hence only require one number.

### Quantisation

Note that this is covered in lecture 3.

**Quantisation** maps a continuous signal to a discrete signal. The pictures from a camera are a continuous signal, but when it is stored on the camera, it is quantised to a discrete signal. Numerical errors can occur during this process, and the magnitude of the errors depends on the number of bits used (less error with more bits; 16 bits can store from 0 to 65535, compared to 8 bits storing from 0 to 255).

Physically, an analog-to-digital convert (ADC) is used to perform the conversion. The energy of photons are converted into voltage, amplified, and then converted.

### Compression

In order to reduce the cost of storage to transmission, compression may be used. Lossy compression loses information after the compression (such as discrete cosine transform (DCT) in JPEG, often used for images or videos). However, lossless compression can also reduce the file size (less efficient compared to lossy), and is preferred for archival purposes or important imaging, where detail needs to be recovered.

## Lecture 3 - Image Filtering I

Note that in this course, most of the examples will be done on greyscale images, but can be applied to the channels individually. Some examples of filters include;

- identity filter

  Does nothing to the image.
- low-pass / smoothing (moving average, Gaussian)

  Removes high-frequency signals, and keep low-frequency signals.
- high-pass / sharpening

  Similar to the previous filter (keeps high-frequency signals, and removes low-frequency).
- denoise (median, non-local means, block-matching and 3D filtering)

### Moving Average Filter

This is commonly used for 1D signal processing (time series), such as stocks, which can be quite noisy. To smooth out a noisy curve, it moves a window across the signal (and calculates the average value within the window) - the larger the window size, the smoother the result.

In a two dimensional case, we can use a **filter kernel** (for example, with a 3 × 3 kernel);

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\1 & 1 & 1\\1 & 1 & 1\end{bmatrix}$$

At each pixel, we apply the kernel centred at the pixel, and take the average of the pixels around it (and itself) to create a new output image. When we blur with a larger kernel, such as $7 \times 7$, we end up with a blurrier image. Note that the output image is smaller than the input image. We can pad the image with zeroes (anything outside of the picture is 0), or by mirroring the pixels (copying the boundary pixels).

Consider an image of size $N \times N$, and a kernel size of $K \times K$. At each pixel, we perform $K^2$ multiplications (by the kernel weights), and then $K^2 - 1$ summations. This has to be done for each pixel, hence $N^2$ times. Therefore this results in $N^2K^2$ multiplications and $N^2(K^2 - 1)$ summations; giving a **complexity** of $O(N^2K^2)$. However, we'd like to reduce this, if possible.

If a big filter can be separated as two filters (**separable filter**) applied consecutively, we can perform the first operation, and then the second. An average in a 2D window can be done as an average across rows (horizontal), and then an average across columns (vertical);

$$\begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix} * \begin{bmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{bmatrix}$$

Doing these two filters, we end up with an equivalent result to the original 2D filter. Note that $*$ is a convolution (see next lecture).

Consider the complexity of separable filtering. The image size remains as $N \times N$, however we have two kernels, of $1 \times K$ and $K \times 1$ respectively. At each pixel we do $K$ multiplications followed by $K - 1$ summations. Again, this is done for $N^2$ pixels, and twice (once for each filter). Therefore, the total number of multiplications is $2N^2K$ multiplications and $2N^2(K-1)$ summations. This is better, in contrast to the original complexity, as we have complexity of $O(N^2K)$ - which will make a difference for large $K$.

A moving average filter removes high frequency signals (noise or sharpness), which results in a smooth but blurry image.

### Gaussian Filter

The kernel is a 2D Gaussian distribution;

$$h(i, j) = \frac{1}{2\pi\sigma^2} e^{-\frac{i^2+j^2}{2\sigma^2}}$$

Here we have $i, j = 0, 0$ as the centre of the kernel. While the support is infinite, small values outside the range $[-k\sigma, k\sigma]$ can be ignored (very small values, such as $k = 3$ or $k = 4$). Note that $\sigma$ is a manually defined parameter. This is a separable filter, which is equivalent to two 1D Gaussian filters with the same $\sigma$, with one along the $x$-axis and the other along the $y$-axis;

$$\begin{aligned} h(i, j) &= h_x(i) * h_y(j) \\ h_x(i) &= \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{i^2}{2\sigma^2}} \end{aligned}$$

### High-pass Filter

One design is to do the following;

$$\underbrace{\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}}_{\text{identity}} + \underbrace{\left( \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix} \right)}_{\text{high-frequency}} = \begin{bmatrix} -\frac{1}{9} & -\frac{1}{9} & -\frac{1}{9} \\ -\frac{1}{9} & \frac{17}{9} & -\frac{1}{9} \\ -\frac{1}{9} & -\frac{1}{9} & -\frac{1}{9} \end{bmatrix}$$

We can add a high frequency signal to the identity, in order to enhance it.

### Median Filter

This is a non-linear filter (not performing an average calculation by multiplication). This moves a sliding window, and replaces the centre pixel with the median value in the window - this is not a linear equation.

# Lecture 4 - Image Filtering II

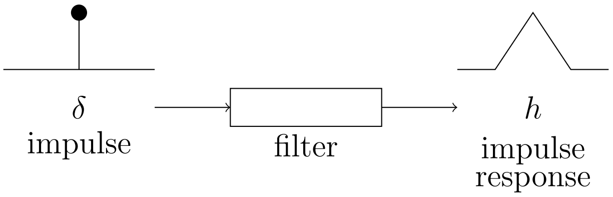
## Mathematical Description

Consider a simple filter, of size 3, in 1D. With an input $f$, and an output $g$, it can be written as the weighted average in a window;

$$g[n] = \frac{1}{3}f[n-1] + \frac{1}{3}f[n] + \frac{1}{3}f[n+1]$$

In general, filtering takes in an input signal $f$, processes it and generates an output signal $g$. A filter is a device (or process) that removes unwanted components or features from a signal (keeps / enhances wanted filters).

In order to mathematically describe a filter, we need the concept of **impulse response**; the output of a filter when the input is an **impulse signal** (only have a signal at a single time point);



For a continuous signal, we treat an impulse as a Dirac delta function $\delta(x)$, whereas for a discrete signal, we treat an impulse as a Kronecker delta function $\delta[i]$;

$$\delta(x) = \begin{cases} \infty & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\int_{-\infty}^{\infty} \delta(x)\,\mathrm{d}x = 1$$

$$\delta[i] = \begin{cases} 1 & \text{if } i = 0 \\ 0 & \text{otherwise} \end{cases}$$

The impulse response $h$ completely characterises a **linear time-invariant** filter. Note that we can consider a filter as time-invariant if, by shifting the input signal some number of time steps $k$, the output signal will remain the same (shape and values) but shifted by the **same** number of time steps. For example;

- $g[n] = 10 \cdot f[n]$ is time-invariant and amplifies the input by a constant
- $g[n] = n \cdot f[n]$ is **not** time-invariant since the amount it amplifies the input depends on the time step $n$

As long as we know $h$, and have an input $x$, we can calculate the output signal $y$. Since it uniquely describes a filter, we often denote a filter by its impulse response function $h$.

Additionally, a filter can be **linear**. If it is a linear system, when two input signals are combined linearly, their outputs will also be combined linearly. Let $f_1[n]$ lead to an output $g_1[n]$, and $f_2[n]$ to $g_2[n]$.

$$\text{output}(\alpha f_1[n] + \beta f_2[n]) = \alpha g_1[n] + \beta g_2[n]$$

## Convolution

Most of the filters we've previously covered are linear time-invariant. Since $h$ characterises how the system works (as it is linear time-invariant), it's possible for the output $g$ to be described as the **convolution** between an input $f$ and impulse response $h$;

$$g[n] = f[n] * h[n]$$

We can describe an input signal $f[n]$ as the following, where each time step is a constant multiplied by a spike;

$$f[n] = f[0]\delta[n] + f[1]\delta[n-1] + f[2]\delta[n-2] + f[3]\delta[n-3] + \dots$$

However, since we know the output of $\delta[n]$ is $h[n]$, we can write the output as;

$$g[n] = f[0]h[n] + f[1]h[n-1] + f[2]h[n-2] + f[3]h[n-3] + \dots$$

The output mathematical operation is defined as a convolution, where a signal $f$ and a filter with impulse response / convolution kernel $h$ is defined as;

$$g[n] = f[n] * h[n] = \sum_{m=-\infty}^{\infty} f[m]h[n-m]$$

The continuous form (previously we have only considered the discrete form);

$$g(t) = f(t) * h(t) = \int_{-\infty}^{\infty} f(\tau)h(t-\tau)\,\mathrm{d}\tau$$

Focusing on the discrete case, we notice the following (when we replace $m$ with $n-m$);

$$\sum_{m=-\infty}^{\infty} f[m]h[n-m] = \sum_{m=-\infty}^{\infty} f[n-m]h[m]$$

This shows that the convolution of $f$ and $h$ is equivalent to the convolution of $h$ and $f$ (commutativity);

$$f[n] * h[n] = h[n] * f[n]$$

By expanding the equations, we can also show that convolution satisfies associativity, such that;

$$f * (g * h) = (f * g) * h$$

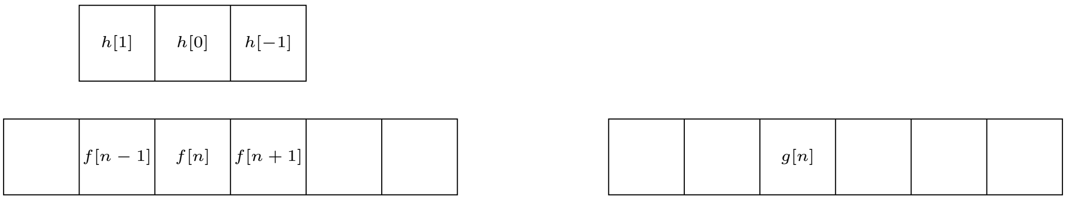
We also have distributivity;

$$f * (g + h) = (f * g) + (f * h)$$

And also differentiation;

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

With a more concrete example, we can visualise it as follows (this is the moving average of size 3);



In our case, we have the kernel $h[n]$, and the values;

$$\begin{aligned}
h[-1] &= \frac{1}{3} \\
h[0] &= \frac{1}{3} \\
h[1] &= \frac{1}{3} \\
h[n] &= \left[\tfrac{1}{3},\ \tfrac{1}{3},\ \tfrac{1}{3}\right]
\end{aligned}$$

This can be expanded into the 2D case, which is used for image filtering;

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

This can also be written as the following, replacing $m-i, n-j$ by $i, j$;

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

By using the property of associativity, if a big filter (call it $f_b$) can be written as the convolution of $g$ and $h$ (smaller filters), we can first convolve $f$ with $g$, then with $h$ - this is used for separable filtering;

$$f * f_b = f * (g * h) = (f * g) * h$$

For example (note that we have padded zeroes, but in code we do not need that);

$$\underbrace{\begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}}_{f_b} = \underbrace{\begin{bmatrix} 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ 0 & 0 & 0 \end{bmatrix}}_{g} * \underbrace{\begin{bmatrix} 0 & \frac{1}{3} & 0 \\ 0 & \frac{1}{3} & 0 \\ 0 & \frac{1}{3} & 0 \end{bmatrix}}_{h}$$

## Lecture 5 - Edge Detection I

### Importance of Edges

In computer vision an edge refers to lines where image brightness changes sharply with discontinuities. This may be due to different reasons such as discontinuities in colour, depth, surface normals, etc. Edges capture important properties of what we see in the world, and they are important features for image analysis (for example, we first capture edges, then facial features, and so on). *Hubel* and *Wiesel* performed vision experiments in 1959, finding that neuron cells in the primary visual cortex are orientation selective, responding strongly to lines or edges of a particular orientation. Humans can also read images even if reduced to simple line drawings. Edges are also heavily used in convolutional networks, with the first layer tending to learn edges, with the later layers learning increasingly complex patterns.

### Detection

An image can be considered as a function of pixel positions (consider plotting the image on the $x-y$ axes, and having the $z$ axis denote intensity). Mathematically, derivatives characterise function discontinuities, which can be used to help edge detection. For a continuous function, the derivative is;

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

On the other hand, for a discrete function (finite difference);

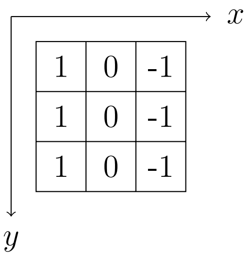
$$\underbrace{f'[x] = f[x+1] - f[x]}_{\text{forward difference}} \qquad \underbrace{f'[x] = f[x] - f[x-1]}_{\text{backward difference}} \qquad \underbrace{f'[x] = \frac{f[x+1] - f[x-1]}{2}}_{\text{central difference}}$$

Notice that these can also be performed with convolutions, using the following kernels;

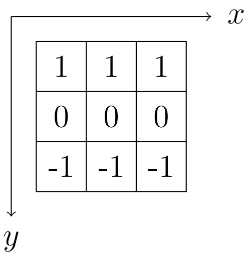
$$\underbrace{h = [1, -1, 0]}_{\text{forward difference}} \qquad \underbrace{h = [0, 1, -1]}_{\text{backward difference}} \qquad \underbrace{h = [1, 0, -1]}_{\text{central difference}}$$

### Prewitt and Sobel Filters

The Prewitt filters, along the $x$-axis (horizontal direction) and along the $y$-axis (vertical direction) are as follows;



| | | |
|---|---|---|
| 1 | 0 | -1 |
| 1 | 0 | -1 |
| 1 | 0 | -1 |

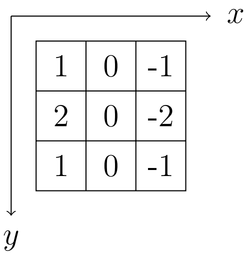


| | | |
|---|---|---|
| 1 | 1 | 1 |
| 0 | 0 | 0 |
| -1 | -1 | -1 |

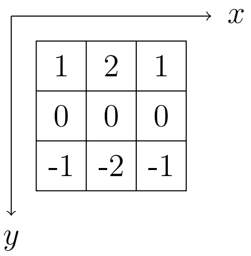
Note that the Prewitt filter is a separable filter ([1] - moving average for smoothing);

$$\underbrace{\begin{bmatrix}1 & 0 & -1\\1 & 0 & -1\\1 & 0 & -1\end{bmatrix}}_{\text{Prewitt filter}} = \underbrace{\begin{bmatrix}0 & 1 & 0\\0 & 1 & 0\\0 & 1 & 0\end{bmatrix}}_{[1]} * \underbrace{\begin{bmatrix}0 & 0 & 0\\1 & 0 & -1\\0 & 0 & 0\end{bmatrix}}_{\text{finite difference}}$$

The Sobel filter is quite similar to the Prewitt filter;



| | | |
|---|---|---|
| 1 | 0 | -1 |
| 2 | 0 | -2 |
| 1 | 0 | -1 |



| | | |
|---|---|---|
| 1 | 2 | 1 |
| 0 | 0 | 0 |
| -1 | -2 | -1 |

Note that the Sobel filter is also a separable filter;

$$\underbrace{\begin{bmatrix}1 & 0 & -1\\2 & 0 & -2\\1 & 0 & -1\end{bmatrix}}_{\text{Sobel filter}} = \underbrace{\begin{bmatrix}0 & 1 & 0\\0 & 2 & 0\\0 & 1 & 0\end{bmatrix}}_{\text{smoothing}} * \underbrace{\begin{bmatrix}0 & 0 & 0\\1 & 0 & -1\\0 & 0 & 0\end{bmatrix}}_{\text{finite difference}}$$

Note that the outputs of the filters are different, with the horizontal filter detecting changes in the horizontal direction; describing discontinuity along the $x$-axis (leading to vertical lines). These can be combined to describe two properties of edges; the magnitude and the orientation.

$$\begin{aligned} g_x &= f * h_x && \text{derivative along } x\text{-axis} \\ g_y &= f * h_y && \text{derivative along } y\text{-axis} \\ g &= \sqrt{g_x^2 + g_y^2} && \text{magnitude of the gradient} \\ \theta &= \text{arctan2}(g_y, g_x) && \text{angle of the gradient} \end{aligned}$$

### Derivative of Gaussian

Note that in both the filters above, there is some smoothing as derivatives are sensitive to noise (therefore the smoothing kernel helps to suppress noise). The Prewitt filters use the mean average kernel, whereas Sobel filters use the kernel $[1, 2, 1]$. Another option is to use the Gaussian kernel for smoothing, before calculating derivatives;

$$h[x] = \frac{1}{\sqrt{2\pi}\sigma}e^{-\frac{x^2}{2\sigma^2}}$$

The derivative of Gaussian filter is an operation, performing Gaussian smoothing, before taking the derivative. However, recalling the rules for the differentiation of convolution, we have $f$ (input signal) convolved with the derivative of the Gaussian kernel;

$$\begin{aligned}\frac{\mathrm{d}}{\mathrm{d}x}(f * h) &= f * \frac{\mathrm{d}h}{\mathrm{d}x} \\ &= f * \frac{-x}{\sqrt{2\pi}\sigma^3}e^{-\frac{x^2}{2\sigma^2}}\end{aligned}$$

In the 2D case, the Gaussian filter is as follows;

$$h[x, y] = \frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}}$$

However, it is a separable filter (and therefore can be sped up) equivalent to the convolution of two 1D Gaussian filters. Notice there's a parameter $\sigma$ in the derivative of Gaussian. With a small $\sigma$, there is more detail in the magnitude map; on the other hand, a large $\sigma$ value suppresses noise and results in a smoother derivative. As such, different $\sigma$ values help to find edges at different scales.

## Lecture 6 - Edge Detection II

### Canny Edge Detection

The results of the filters from the previous lectures (gradient magnitude map) have values ranging from black to white. However, we want either a 0 or 1 (black or white) - a **binary edge map**.

There should be good detection; a low probability of failing too mark real edge points and low probability of falsely marking non-edge points. In addition, there should be good localisation - the points marked as edges should be as close as possible to the centre of the edge. Finally, there should only be a single response to a single edge.

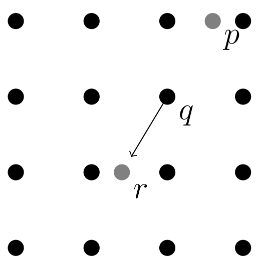
Canny edge detection is done in the following steps;

1. perform Gaussian filtering to suppress noise

   The choice of $\sigma$ depends on the type of edge we desire at the end. If we want large edges (such as buildings), $\sigma$ should be set to a large value (such as 7). However, if we want to pay attention to fine features, $\sigma$ can be set to a small value.

2. calculate gradient magnitudes and directions
3. apply non-maximum suppression (NMS) to get a single response for each edge

   Non-maximum suppression aims to get a single response for each edge, by using the idea that the edge occurs where the gradient magnitude is maximum at the centre of the edge.



   Compare the magnitude at $q$ with the magnitudes of $p$ and $r$, both along the gradient direction (in opposite directions). If $q$ is the local maximum (larger than both $p$ and $r$), we keep the magnitude at pixel $q$.

$$M(x, y) = \begin{cases} M(x, y) & \text{if local maximum} \\ 0 & \text{otherwise} \end{cases}$$

However, notice that there may be issues with $p$ and $r$ not being at exact pixel locations. For example, using pixel $r$ (not at an integer position), we can perform interpolation. It's possible to use linear interpolation (weighted by distance), or a simpler algorithm such as nearest neighbour interpolation (better performance). Note that different interpolation algorithms will lead to different outcomes. Another approach is to round the gradient direction into 8 possible angles, in steps of $45°$ in the range $[0°, 315°]$, allowing us to check only along those directions to give exact pixels (intuitively quite similar to nearest neighbour).

After NMS, there are fewer points with bright values.

4. perform hysteresis thresholding to find potential edges

   Many pixels that are local maxima may still have very low magnitudes; however we only want edges with high magnitudes. A simple thresholding would be to convert an intensity image to a binary image with a threshold $t$;

   $$\text{binary}(x, y) = \begin{cases} 1 & \text{if } I(x, y) \geq t \\ 0 & \text{otherwise} \end{cases}$$

   On the other hand, hysteresis thresholding defines two thresholds $t_{\text{low}}$ and $t_{\text{high}}$. If the magnitude is $\geq t_{\text{high}}$, it is accepted as an edge pixel, and if it is $< t_{\text{low}}$, it is rejected. However, if we have a value between the thresholds, we have a **weak edge** (which may or may not be an edge). If it is connected to existing edge pixels, it is accepted, whereas if it is not connected (adjacent to one strong edge) to an existing edge, it will be rejected.

The initial goals are satisfied as follows;

- **good detection**

  False positives are reduced by using Gaussian smoothing to suppress noise. On the other hand, false negatives are reduced by using hysteresis thresholding to find weak edges.

- **good localisation**

  NMS finds locations based on gradient magnitude and direction.

- **single response** also done with NMS

### Learning-based Edge Detection

Decades of effort have been made to improve detection accuracy. This includes using richer features such as colour and texture, enforcing smoother, as well as using machine learning (by learning mapping from an image to edge directly from data).

This machine learning algorithm assumes paired data (images $x$ and manually defined edge maps $y$). The problem finds a model (with model parameters $\theta$) that maps $x$ to $y$, such that $y = f(x \mid \theta)$. This differs from Canny edge detector, as our example integrates from multiple scales (fine-scale edges) with coarse-scale edges to form a final output. On the other hand, Canny edge detector uses a single scale for edge detection, controlled by a single parameter $\sigma$.

An application for learning-based edge detection is to learn a mapping from a rough sketch to a simplified sketch. However, this isn't just an edge detection problem since it cannot be solved using NMS; when this is done by humans, a high-level understanding about the sketch is required.

### Conclusion

Edge detection is a fundamental problem in image processing and computer vision; aiming to identify where discontinuities occur, or identifying points that are edges. Two solutions are proposed;

- if we know the explicit criteria, we can implement computational criteria

- otherwise, we can collect data pairs representing what we want to achieve and train a machine learning model

Edges provide important low-level features both human vision and computer vision for understanding images. These algorithms provide ideas for other detection algorithms.

We can also go in the other direction, taking edges to images. This aims to train a model that generates an image that looks close to real images, from edges. A model $G$ (generator) learns the mapping from edge to image. For example, we take an edge image $x$ to $G(x)$ (an image). A discriminator $d$ then attempts to assess whether the image is real or fake (GAN).

## Lecture 7 - Hough Transform

In the last lecture, we covered edge detection (1 for edge pixel, 0 for background). If we know these edges form some shape (such as a line), we want to obtain a parametric representation.

### Line Parameterisation

A line can be represented by two parameters (e.g. $m$ (slope) and $b$ ($y$-intercept)), which is much more efficient than a lot of edge points;

- slope intercept form $m$ (slope) and $b$ ($y$-intercept)
$$y = mx + b$$
- double intercept form $(a, 0)$ and $(0, b)$ are on the line
$$\frac{x}{a} + \frac{y}{b} = 1$$
- normal form $\theta$ is angle and $\rho$ is distance
$$x\cos(\theta) + y\sin(\theta) = \rho$$

### Hough Transform

Hough transform transforms from image space (edge map) to parameter space (two parameters of a line). The output is a parametric model, from a n input of edge points. Each edge point 'votes' for possible models in the parameter space. One way is to fit a line model $(m, b)$ to the edge points $(x_1, y_1), (x_2, y_2), \dots$;

$$\min_{m,b} \sum_i (y_i - \underbrace{(mx_i + b)}_{\hat{y}})^2$$

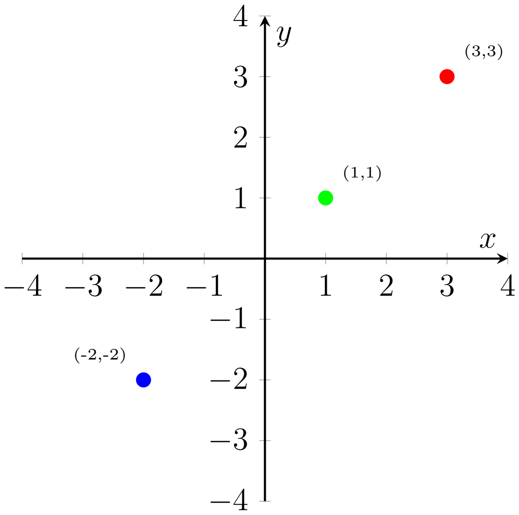
For the Hough transform we will use the slope intercept form.

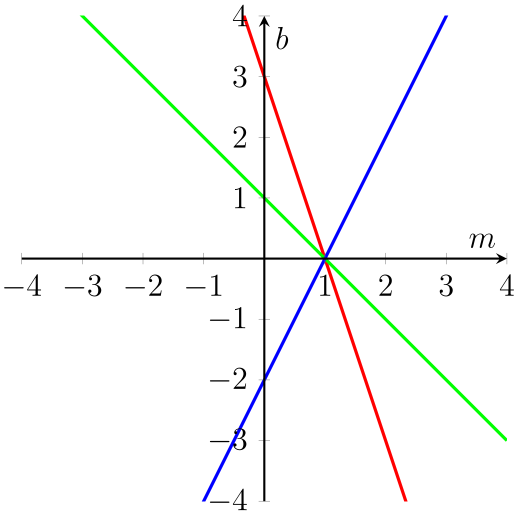
$$y = mx + b \Leftrightarrow b = y - mx$$

Assume we have the edge points $(x_1, y_1), \dots,$ each point will vote for a line model in the parameter space; for example, the first point votes for;

$$b = y_1 - mx_1$$

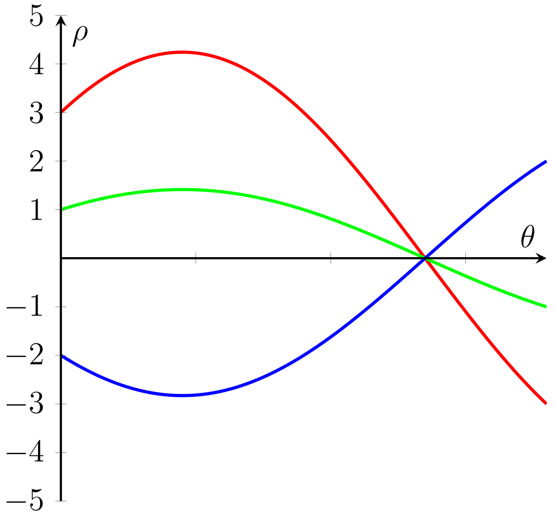
This can be seen graphically as the following (image space on the left, parameter space ($b = y - mx$) on the right);





The intersection will get 3 votes, the other points on the line get 1 vote each, and the empty spaces get 0 votes. This gives a result of $m = 1$ and $b = 0$, thus we get the line $y = x$. However, in practice this is divided into two dimensional bins, with each point incrementing the vote by 1 in one of the bins. One problem is that the parameter space is too large for $m$ and $b$ (both are infinite).

The solution to this is to use the normal form. While $\rho$ can still be infinite (in theory, however we can actually limit this to the image size in practice), we have $\theta \in [0, \pi)$. The transform from the image space to the parameter space ($x \cos(\theta) + y \sin(\theta) = \rho$) will however look different;



This gives the resultant vote $\theta = \frac{3\pi}{4}$ and $\rho = 0$.

To perform this algorithm, we do the following;

1. initialise the bins $H(\rho, \theta)$ to zero in the parameter space
2. for each edge point $(x, y)$;

   a. for $\theta$ from 0 to $\pi$

      i. calculate $\rho = x \cos\theta + y \sin\theta$

      ii. accumulate $H(\rho, \theta) = H(\rho, \theta) + 1$

3. find $(\rho, \theta)$ where $H(\rho, \theta)$ is a local maximum and larger than a threshold

   This is done in a similar way to the previous lecture. The local maximum allows multiple solutions to be detected, and the threshold reduces false positives.

4. the detected lines are given by $\rho = x \cos\theta + y \sin\theta$

In contrast to model fitting (minimisation), Hough transform can simultaneously detect multiple lines (as long as they are local maximums and above a threshold) whereas the former can only detect a single line. Hough transform is robust to noise for two reasons. First, the initial edge map is generated after image smoothing (already suppressing noise). Furthermore, the broken edge maps are still able to vote and contribute to line detection. Similarly, it is also robust to object occlusion (objects overlapping, such as a tree obstructing part of a face).

However, the computational complexity is high; we have to vote in a 2D or 3D parameter space for each edge point. We also need to set parameters carefully, such as parameters for the edge detector, the threshold for the accumulator, or the radius range (in the case of circles).

This can be generalised to other shapes (which can be analytically represented), such as ellipses or planes in a 3D space. We can also weigh the votes, by taking the gradient magnitude (such that stronger edge points are weighted higher).

In general, if it is a simple shape (such that there is no analytical equation), we can still vote as long as we have a model. For example, consider the case of pedestrian detection; we can vote for points above a patch where a foot is detected, and below where a head is detected.

### Circles

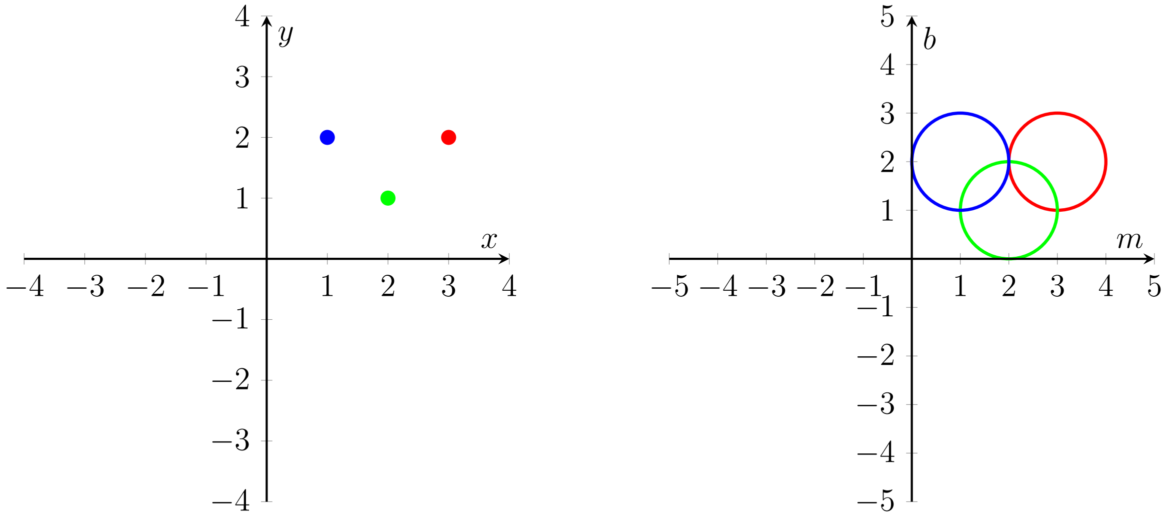
We can parameterise a circle as;

$$(x-a)^2 + (y-b)^2 = r^2$$

However, this is a very large parameter space (hence many bins). If we have some prior knowledge (such as the radius $r$), we can make the problem easier by reducing the search space - we can just vote for $a, b$ (the centre of the circle). The votes are still circles in the parameter space $H(a, b)$ (assuming $r = 1$);

$$(a-x)^2 + (b-y)^2 = 1$$

This can be seen graphically as the following (image space on the left, parameter space ($(a-x)^2 + (b-y)^2 = 1$) on the right);

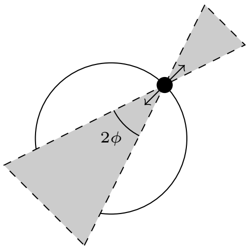


If we don't know the radius, we set a range $r \in [r_{\text{min}}, r_{\text{max}}]$, and then perform a similar algorithm to before.

Another representation of a circle can be done with trigonometric functions (parametric form);

$$x = a + r\cos\theta$$
$$y = b + r\sin\theta$$

However, if we know $\theta$, we can vote along a direction. Since we know the direction of an edge point (obtained from edge detection), we can narrow it down (along the gradient or opposite the gradient). We assume an accuracy of $\pm\phi$, and vote within that area.



The algorithm can be done as follows;

```
1. initialise all bins $H(a, b, r)$ to zero
2. for each possible radius $r \in [r_{\min}, r_{\max}]$
    a. for each edge point $(x, y)$;
        i. let $\theta$ be gradient direction / opposite gradient direction
        ii. calculate $a = x - r \cos\theta$ and $b = y - r \sin\theta$
        iii. accumulate $H(a, b, r) = H(a, b, r) + 1$
3. find $(a, b, r)$ where $H(a, b, r)$ is a local maximum and larger than a threshold
```

## Lecture 8 - Interest Point Detection I

An interest point is a point we are interested in, and are useful for subsequent processing and analysis (classification, matching, retrieval, etc). These are commonly corners or blobs, where local image structure is rich. These are also known as keypoints, landmarks, or low-level features.

### Applications

We can define landmarks, on a face, which are most representative. Once we detect these landmarks, the locations can be used for purposes such as mood analysis, AR, etc.

Another application is **image matching**; detecting relationships between images (such as orientation) and finding spatial correspondence between two images. This can be done by pixels (by using all information), by edges (using some information), or by interest points (using only important information). Matching with pixels often works by optimising a similarity metric (based on all pixels), with a spatial transformation $T$;

$$\max_T \text{Similarity}(I_A, I_B(T))$$

A more efficient approach is to find interest points and see how they are transformed between images (consider different perspectives of an art piece).

### Harris Detector

Corners are intersection of edges (which are high magnitude of gradient). For corner detection, by just looking at the gradient magnitude of a single pixel, we can't tell if it is a corner or edge pixel. However, if we look at the small windows, we can tell a difference;

| | |
|---|---|
| • flat | change of intensity in **neither** direction |
| • edge | change of intensity along just **one** direction |
| • corner | change of intensity along **both** directions |

We want to define a window $W$ and a window function $w$ (1 if it is in the window, 0 otherwise), where we have the sum of squared difference, intensity in the shifted window and the original intensity;

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

Note that the window function $w(x, y)$ is used as it isn't necessarily 1 inside, and 0 outside, we can use also use a Gaussian to put more focus on the middle of the window.

We can use the following Taylor expansion (note that $I_x$ and $I_y$ denote the image derivative along $x$ and $y$ respectively);

$$I(x+u, y+v) = I(x, y) + uI_x(x, y) + vI_y(x, y) + \dots$$

Using just the first order approximation;

$$\begin{aligned}
E(u, v) &= \sum w(x, y)[I(x+u, y+v) - I(x, y)]^2 \\
&\approx \sum w(x, y)[u\,\underbrace{I_x(x, y)}_{I_x} + v\,\underbrace{I_y(x, y)}_{I_y}])^2 \\
&= \sum w(x, y)(u^2 I_x^2 + 2uvI_xI_y + v^2 I_y^2) \\
&= \sum w(x, y) \begin{bmatrix} u & v \end{bmatrix} \begin{bmatrix} I_x^2 I_x I_y & \\ I_x I_y & I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} && \text{written as matrix formula} \\
&= \begin{bmatrix} u & v \end{bmatrix} \sum w(x, y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}
\end{aligned}$$

Therefore, for a small shift $(u, v)$, we have the approximation (note that $E(u, v)$ will be large when image derivatives are large);

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

Where $\boldsymbol{M} \in \mathbb{R}^{2\times 2}$ is computed from the window function and image derivatives (this matrix will also provide the direction of changes);

$$\boldsymbol{M} = \sum_{x,y} w(x, y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

We have the following simple cases (and examples) - note that these are **only** diagonals;

- **flat region** small values on diagonal

  $$\boldsymbol{M} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

- **edge** diagonal, large on one of diagonal

  $$\boldsymbol{M} = \begin{bmatrix} 10 & 0 \\ 0 & 0.1 \end{bmatrix}$$

  In this case, we have a large change if shifting along $u$, but a small change if we shift along $v$, hence we have an edge.

- **corner** diagonal, large on both diagonal

  $$\boldsymbol{M} = \begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix}$$

  If we move along either direction, there will be a large change, hence it is a corner.

However, in the case that $\boldsymbol{M}$ is more complex (such that it isn't a diagonal matrix), we can simplify it by performing eigen decomposition. Because $\boldsymbol{M}$ is a real symmetric matrix, we can decompose it as;

$$\boldsymbol{M} = \boldsymbol{P}\boldsymbol{\Lambda}\boldsymbol{P}^\top$$

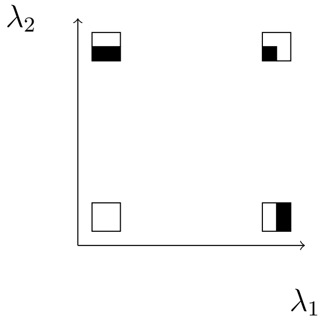
In the above, $\boldsymbol{P}$ has eigenvectors (orthogonal to each other) as columns (and therefore $\boldsymbol{P}^\top$ has eigenvectors as rows), and $\boldsymbol{\Lambda}$ is a diagonal matrix with eigenvalues;

$$\boldsymbol{\Lambda} = \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$$

Referring back to $E(u, v)$, we have the following;

$$E(u, v) \approx \begin{bmatrix} u & v \end{bmatrix} \boldsymbol{P}\boldsymbol{\Lambda}\boldsymbol{P}^\top \begin{bmatrix} u \\ v \end{bmatrix}$$

The same rules as above apply; however instead of moving along $u$, we instead move along the first eigenvector (and instead of moving along $v$, we move along the second eigenvector). The edges are no longer horizontal or vertical, they are in the directions of the eigenvectors. The eigenvalues can be interpreted as follows;



Here we have the following cases;

- $\lambda_1 \sim 0$ and $\lambda_2 \sim 0$ flat
- $\lambda_1 >> \lambda_2$ vertical edge
- $\lambda_2 >> \lambda_1$ horizontal edge
- $\lambda_1$ and $\lambda_2$ both large corner

However, we want to define a single number for 'cornerness' (high when both eigenvalues are large). This can be defined in multiple ways;

- *Harris and Stephens (1988)*

  $$R = \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2$$

  Here $k$ is a small number, such as 0.05. The small $k$ ensures that the number does not become too small (when subtracted) when both eigenvalues are large.
- *Kanade and Tomasi (1994)*

  $$R = \min(\lambda_1, \lambda_2)$$

- *Noble (1998)*

  $$R = \frac{\lambda_1\lambda_2}{\lambda_1 + \lambda_2 + \epsilon}$$

However; notice that we only want the values of $\lambda_1\lambda_2$ and $\lambda_1 + \lambda_2$, therefore we do not need to perform eigen decomposition. Using the following properties for matrix determinant and trace, we can obtain the following;

$$\begin{aligned}
\det(\boldsymbol{M}) &= \det(\boldsymbol{P}\boldsymbol{\Lambda}\boldsymbol{P}^\top) \\
&= \det(\boldsymbol{\Lambda}) \\
&= \lambda_1\lambda_2 \\
\text{trace}(\boldsymbol{M}) &= \text{trace}(\boldsymbol{P}\boldsymbol{\Lambda}\boldsymbol{P}^\top) \\
&= \text{trace}(\boldsymbol{\Lambda}) \\
&= \lambda_1 + \lambda_2
\end{aligned}$$

Therefore, we only need the determinant and trace for the original matrix to calculate $R$;

$$R = \det(\boldsymbol{M}) - k(\text{trace}(\boldsymbol{M}))^2$$

This detector can also find strong responses at blobs and textures, as well as corners. The algorithm is as follows;

1. compute $x$ and $y$ derivatives of an image $\qquad$ $G$ can be the Sobel filter

$$\begin{aligned}
I_x &= G_x * I \\
I_y &= G_y * I
\end{aligned}$$

2. compute the matrix $\boldsymbol{M}$ at each pixel

$$\boldsymbol{M} = \sum_{x,y} w(x, y) \begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix}$$

3. calculate the detector response

$$R = \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2$$

4. detect interest points which are local maxima (and $R$ above a certain threshold)

Note that the Harris detector is **rotation-invariant**. We will still get the same change of intensities when shifting along a rotated direction - the eigenvalues of $\boldsymbol{M}$ remain the same, but the eigenvectors will change. However, it is not invariant to scale. At a different zooming scale, an edge may be detected as a corner or vice versa (for example if the circle arc is large it will be detected as an edge, but if it is small, it will be detected as a corner).

## Lecture 9 - Interest Point Detection II

### Scale

The intuitive ideal for scaling is to check whether the Harris detector gives the highest response at a given scale. If the region looks most like a corner at scale $\sigma$, there should be a high response.

$$\boldsymbol{M} = \sum_{x,y} w(x, y) \begin{bmatrix} I_x^2(\sigma) & I_x(\sigma)I_y(\sigma) \\ I_x(\sigma)I_y(\sigma) & I_y^2(\sigma) \end{bmatrix}$$

The response is determined by the eigenvalues of $\boldsymbol{M}$, which are determined by the derivatives $I_x(\sigma)$ and $I_y(\sigma)$. These are inversely proportional to scale $\sigma$; the larger the scale, the smaller the derivative magnitude.

Consider signals which only differ by scale $s$ (and have same peak magnitude);

$$f(x) = g(sx)$$

However, when we look at the derivatives of the function, we have a different peak magnitude; since the derivative looks at how quickly the functions change. Since we have the relation above, we have;

$$f(x + \Delta x) - f(x) = g(sx + s\Delta x) - g(sx)$$

Looking at the Taylor expansion for the above, we have;

$$\begin{aligned} f(x + \Delta x) &= f(x) + \Delta x \cdot f'(x) + \dots \\ g(x + \Delta x) &= g(x) + \Delta x \cdot g'(x) + \dots \\ g(sx + s\Delta x) &= g(sx) + s\Delta x \cdot g'(sx) + \dots \end{aligned}$$

Substituting back into the first equation, it gives us the following

$$\begin{aligned} \Delta x \cdot f'(x) &= s\Delta x \cdot g'(sx) \\ f' &= sg'(sx) \end{aligned}$$

This gives us the following result;

$$\frac{\mathrm{d}f}{\mathrm{d}x} = s \cdot \frac{\mathrm{d}g}{\mathrm{d}x}\bigg|_{sx}$$

Therefore, we can multiply derivative by its scale $s$ to allow for magnitude comparison across scales, with the scale adapted Harris detector;

$$\boldsymbol{M} = \sum_{x,y} w(x, y)\sigma^2 \begin{bmatrix} I_x^2(\sigma) & I_x(\sigma)I_y(\sigma) \\ I_x(\sigma)I_y(\sigma) & I_y^2(\sigma) \end{bmatrix}$$

We can apply the scale adapted detector at multiple scales - at each pixel we determine the scale with the largest response (at this scale, the region looks most like a corner). Something is detected as an interest point if it is a local maximum both along the scale dimension (most appropriate scale) and across space, therefore we can use $(x, y, \sigma)$ for the interest point. This gives the following adapted algorithm;

1. for each scale $\sigma$

   a. perform Gaussian smoothing with $\sigma$
   b. calculate $x$ and $y$ derivatives of the smoothed image $I_x(\sigma)$ and $I_y(\sigma)$, respectively
   c. at each pixel, compute the matrix $\boldsymbol{M}$ (see above)
   d. calculate the detector response $R = \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2$

2. detect interest points which are local maxima across both scale and space, with $R$ above a threshold

### Laplacian of Gaussian (LoG)

This first performs Gaussian smoothing, followed by the Laplacian operator. The Laplacian is the sum of second derivatives, for a 2D image;

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

We have the following filter;

$$\underbrace{\begin{bmatrix} 0 & 0 & 0 \\ 1 & -2 & 1 \\ 0 & 0 & 0 \end{bmatrix}}_{\frac{\partial^2 f}{\partial x^2}} + \underbrace{\begin{bmatrix} 0 & 1 & 0 \\ 0 & -2 & 0 \\ 0 & 1 & 0 \end{bmatrix}}_{\frac{\partial^2 f}{\partial y^2}} = \underbrace{\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}}_{\text{Laplacian filter}}$$

The second derivative is even more sensitive to noise than the first derivative.

The Laplacian of Gaussian filter is as follows (where $h$ is the Gaussian kernel);

$$\Delta(f * h) = \frac{\partial^2 (f * h)}{\partial x^2} + \frac{\partial^2 (f * h)}{\partial y^2} = f * \left( \frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2} \right)$$

Since we know the formation of a 2D Gaussian, we can derive the following;

$$\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2} = -\frac{1}{\pi \sigma^4} \left( 1 - \frac{x^2 + y^2}{2\sigma^2} \right) e^{-\frac{x^2+y^2}{2\sigma^2}}$$

LoG can also be a good interest point detector. We need to ensure it is comparable between scales (note that $I_{xx}$ denotes the second derivative, we need to multiply by $\sigma^2$ rather than just $\sigma$; without the multiplication we have the LoG response at scale $\sigma$);

$$\text{LoG}_{\text{norm}}(x, y, \sigma) = \sigma^2 \left( I_{xx}(x, y, \sigma) + I_{yy}(x, y, \sigma) \right)$$

### Difference of Gaussian (DoG)

This is defined as; the difference of Gaussians with different scales (note that $I$ is the image, Lowe suggests $k = \sqrt{2}$);

$$\text{DoG}(x, y, \sigma) = I * G(k\sigma) - I * G(\sigma)$$

This approximates the normalised Laplacian of Gaussian, which provides convenience in calculating the response across different scales;

$$\text{DoG}(x, y, \sigma) \approx (k - 1)\sigma^2 \nabla^2 G(x, y, \sigma)$$

DoG filters are used in SIFT, which is a pipeline for detecting and describing interest points.

### Interest Point Detectors

All of the interest point detectors are scale-invariant (other than the Harris detector, before normalisation);

| detector | response |
|---|---|
| scale adapted Harris detector | $\lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2$ |
| normalised Laplacian of Gaussian | $\sigma^2 \left( I_{xx}(x, y, \sigma) + I_{yy}(x, y, \sigma) \right)$ |
| difference of Gaussian | $I * G(k\sigma) - I * G(\sigma)$ |

All of these follow similar procedures, where we calculate the detector responses across values, providing a three-dimensional response map ($x$ and $y$, as well as $\sigma$ for scale). From here, we find local extrema across both space and scale.

## Lecture 10 - Feature Description I

### Pixel Intensity

A simple method for describing a feature is o use the intensity of a pixel. However, this is sensitive to absolute value (the intensity of the same point will change during different lighting conditions). This descriptor is not discriminative; many pixels can have the same intensity within an image. Furthermore, a single pixel cannot represent any local content (such as an edge, or an object).

### Patch Intensities

This improves on a single pixel, as we can take some window representing the local pattern. If the images are of similar intensity range, and are roughly aligned, it can perform well. However, it will still be sensitive to absolute intensity values as before. Furthermore, it is not rotation-invariant; for example if we look at an image from a different rotation we can still tell what the image is.

### Gradient Orientation

While gradient magnitude is sensitive to intensity changes, orientation isn't. However, this still isn't rotation-invariant.

### Histogram

We can take the intensity histogram of a patch. This is done by binning the intensity values of a patch, and generating a histogram from these values. This has several benefits, where it is robust to rotation, as well as being robust to scaling (if the entire image is enlarged). However, it is still sensitive to intensity changes.

As such, we can consider combining the advantages of gradient orientation and the advantages of this histogram.

### SIFT (Scale-invariant Feature Transform)

This is an algorithm for detecting and describing local features. It transforms an intensity image to a set of interest points, each of which is described by a feature vector.

1. detection of scale-space extrema

   The first step is to search across scales and pixel locations, looking for interest points. This uses the difference of Gaussian filter, as previously encountered (in the SIFT paper, the different scales are multiplied by $\sqrt{2}$ continuously; $\sigma, \sqrt{2}\sigma, 2\sigma, 2\sqrt{2}\sigma, 4\sigma, \dots$);

   $$\text{DoG}(x, y, \sigma) = I * G(k\sigma) - I * G(\sigma)$$

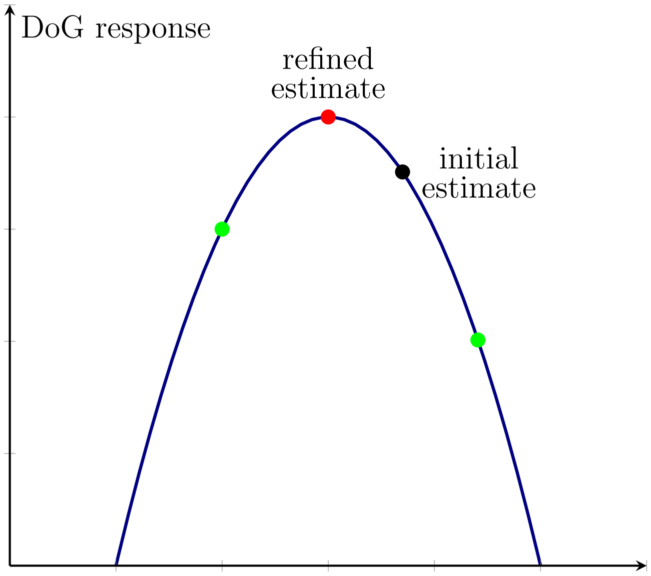
   As before, a point is an interest point if it is a local extremum both along the scale dimension (most appropriate scale) and across space.

   Note that extrema means both minima and maxima. If the input interest point is a very bright blob, DoG filter gives a negative response, and vice versa (a dark blob will lead to a positive response).

2. keypoint localisation

   The initial version of SIFT simply locates the keypoints at the scale-space extrema (which we have from above). An improved version fits a model onto nearby data, improving accuracy for image matching.

   A quadratic function can be fitted onto the DoG response of neighbouring pixels, and the location and scale of extremum for this (quadratic) function can be estimated.



Denote the DoG response as $D(x, y, \sigma)$ or $D(\boldsymbol{x})$, where $\boldsymbol{x} = (x, y, \sigma)^\top$ (containing both location and scale). By Taylor expansion we have (omitting higher order terms);

$$D(\boldsymbol{x} + \Delta\boldsymbol{x}) = D(\boldsymbol{x}) + \underbrace{\frac{\partial D}{\partial \boldsymbol{x}}}_{(1)}^\top \Delta\boldsymbol{x} + \frac{1}{2}\Delta\boldsymbol{x}^\top \underbrace{\frac{\partial^2 D}{\partial \boldsymbol{x}^2}}_{(2)} \Delta\boldsymbol{x}$$

(1) first derivatives can be estimated with finite difference

$$\frac{\partial D}{\partial \boldsymbol{x}} = \frac{D(x+1, y, \sigma) - D(x-1, y, \sigma)}{2}$$

(2) second derivatives / Hessian can also be estimated with finite difference

This gives us a quadratic function for $\Delta\boldsymbol{x}$, the shift to the refined estimate, from the initial estimate $\boldsymbol{x}$. Therefore, to find a refined extrema for the function;

$$\frac{\partial D(\boldsymbol{x} + \Delta\boldsymbol{x})}{\partial \Delta\boldsymbol{x}} = \frac{\partial D}{\partial \boldsymbol{x}} + \frac{\partial^2 D}{\partial \boldsymbol{x}^2}\Delta\boldsymbol{x} = 0 \Rightarrow \Delta\boldsymbol{x} = -\frac{\partial^2 D}{\partial \boldsymbol{x}^2}^{-1} \frac{\partial D}{\partial \boldsymbol{x}}$$

Shifting the initial estimate by $\Delta\boldsymbol{x}$ gives us the refined estimate.

Recall that $\boldsymbol{x}$ is a 3D vector, and gives both location and scale. This means that we can get refined estimates for the location (sub-pixel accuracy), as well as a refined scale, which gives values between the scales of $\sigma, \sqrt{2}\sigma, 2\sigma, 2\sqrt{2}\sigma, 4\sigma, \dots$

3. orientation assignment

   This step attempts to assign a consistent orientation to each keypoint, based on image properties. Feature descriptors can then be represented relative to this orientation, to achieve rotation-invariance. To do this, we need to know the orientation of the feature, and once we have this, we can perform sampling in a rotated coordinate system when we calculate features, giving us the same features.

   We can calculate the gradient orientation for pixels in some neighbourhood and vote for the dominant orientation. We can create an orientation histogram with 36 bins (10° per bin), and each pixel in the neighbourhood votes for an orientation bin (weighted by the gradient magnitude) - the keypoint will be assigned an orientation, which would be the majority bin.

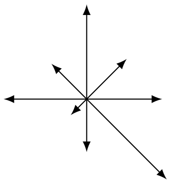
   We now have the location $(x, y)$, scale $\sigma$, and dominant orientation $\theta$. Using this, we an draw samples using a window size proportional to $\sigma$ rotated by $\theta$.

4. keypoint descriptor

   Now that we have sample points, we need to describe the local image content. SIFT uses a histogram of gradient orientations.

   A histogram of gradient orientations is made from the calculated gradient magnitudes and orientation for the sampling points. These orientations are calculated relative to the dominant orientation $\theta$. In practice, subregions are used, with each subregion having $4 \times 4$ samples - each subregion will have an orientation histogram, which (when combined) describes what it looks like around a given keypoint.

   For each subregion, we can construct an orientation histogram with 8 bins (each bin representing 45°). This can also be represented as 8 arrows (seen below), where the length of the arrow denotes the sum of the gradient magnitude along a given direction;



In Lowe's implementation, 16 subregions were used. This gives the descriptor a dimensionality of $128 = 16 \times 8$ (since each subregion has 8 bins); meaning each keypoint is described with a feature vector of 128 elements.

This descriptor is robust to rotation due to a consideration of a dominant orientation, similarly it is robust to scaling as we draw samples from a window proportional to scale. It is also robust to changes in illumination since we use gradient orientations for feature description.

### Keypoint Matching

Keypoints between two images are matched by identifying the nearest neighbours (which is defined by the Euclidean distance of SIFT descriptors). Each keypoint in image $A$ identifies its nearest neighbour in the database of keypoints for image $B$. However, we may not need to find the exact nearest neighbours for the sake of efficiency.

Suppose we find a keypoint $(x, y)$ in $A$ that corresponds to a keypoint $(u, v)$ in $B$. We assume they are related with an affine transformation (rotation, scaling, etc. and translation);

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} m_1 & m_2 \\ m_3 & m_4 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

However, with many pairs of corresponding keypoints, we can write the equation as;

$$\begin{array}{r} \text{keypoint 1} \\ \\ \text{keypoint 2} \\ \\ \\ \end{array} \begin{bmatrix} x_1 & y_1 & 0 & 0 & 1 & 0 \\ 0 & 0 & x_1 & y_1 & 0 & 1 \\ x_2 & y_2 & 0 & 0 & 1 & 0 \\ 0 & 0 & x_2 & y_2 & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ t_x \\ t_y \end{bmatrix} = \begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \\ \vdots \end{bmatrix}$$

This can also be written as a linear system, where only $\boldsymbol{m}$ is unknown;

$$\boldsymbol{A}\boldsymbol{m} = \boldsymbol{b}$$

However, we can obtain the least-square solution to the linear system with the Moore-Penrose inverse, minimising the squared difference $||\boldsymbol{A}\boldsymbol{m} - \boldsymbol{b}||^2$;

$$\boldsymbol{m} = \left(\boldsymbol{A}^\top \boldsymbol{A}\right)^{-1} \boldsymbol{A}^\top \boldsymbol{b}$$

Once this is solved, we have the spatial transformation between the two images.

### RANSAC

However, the squared difference can be sensitive to outliers (noise) - in our case, this would be points that are deemed to be corresponding but actually aren't. To ensure our solution is robust to the outliers we can use methods which consider outliers in the model fitting such as **RANSAC (Random Sample Consensus)**.

RANSAC aims to find the best fitting line to a given set of points. The algorithm to perform this is as follows (this is done repeatedly until we get a good fitting line);

1. randomly sample some points (the example shows 2 points in a 2D illustration)
2. fit a line along the sampled points
3. find the number of **inliers** within a threshold to the line
4. terminate if enough inliers have been found, or we have reached a certain number of iterations

## Lecture 11 - Feature Description II

### SIFT

As mentioned before, the SIFT descriptor at each interest point is formed by the concatenation of the gradient orientation histograms for 16 subregions (each having $128 = 16 \times 8$ values). Calculating magnitudes and orientations can be slow, especially if we want real-time performance (such as stitching on a phone camera, or for robotics).

One solution is to implement SIFT on FPGAs, which is faster than a CPU (once configured to perform this function). Another approach is to improve the algorithm, where we decompose the pipeline into several steps, where we evaluate how to improve each step. The pipeline for image matching is as follows;

1. **feature detection** (e.g. DoG)

   There are a number of approaches for faster feature detection, including separable filtering, downsampling the image, or cropping the Gaussian kernel.

2. **feature description** (e.g. SIFT)

   Here we can consider whether we can evaluate the gradient orientation faster, or if we can use a different method of describing the local content.

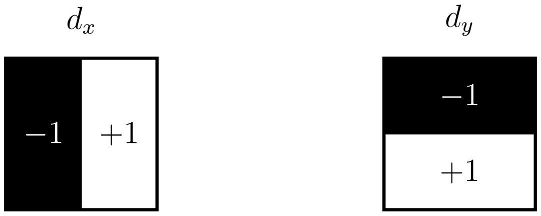
3. **interest point matching**

   Here we can approximate the nearest neighbour search, as well as use a lower-dimensional feature vector.

### SURF (Speeded-Up Robust Features)

To accelerate computation, SURF only computes gradients along horizontal and vertical directions (rather than all of them) using **Haar wavelets**. Recall that the SIFT descriptor needs to calculate a histogram of gradient orientations, for each of the 16 subregions centred at an interest point.

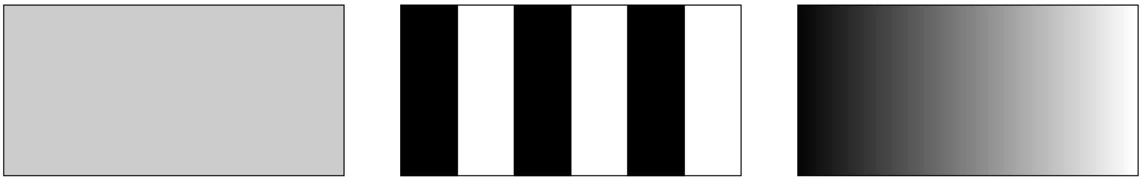
SURF still uses the same subregions and sample points, however we do not need to calculate gradients for arbitrary orientations. This applies two very simple filters (Haar wavelets, of size $2 \times 2$ or $4 \times 4$), $d_y$ and $d_x$ onto the sample points;



Summing the pixel intensities with a weight of 1 or $-1$ is very fast, since it is either an addition or subtraction. For each subregion, we sum up the Haar wavelet responses over the sample points - the descriptor for a given subregion is defined by 4 elements (the sum of values and the sum of absolute values);

$$\left(\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|\right)$$

We can visualise the meanings of these descriptors as follows. If all four values are low, then the subregion is mostly homogeneous. If there are patterns similar to zebra stripes, a change from black to white, and then from white back to back would give a sum of 0 when we look at $d_x$ (since they cancel out). However, if we look at $|d_x|$, the value doubles (since they will not cancel).



| | homogeneous region | zebra pattern | gradually increasing intensities |
|---|---|---|---|
| $\sum d_x$ | low | low | high |
| $\sum \lvert d_x \rvert$ | low | high | high |
| $\sum d_y$ | low | low | low |
| $\sum \lvert d_y \rvert$ | low | low | low |

The SURF descriptor still uses $16 = 4 \times 4$ subregions, each with 4 elements. This leads to a dimensionality of 64 for each interest point. Compared to SIFT, it is approximately 5 times faster due to the use of simple Haar wavelets.

If we know the ground truth (in this case, the transformation between image $A$ and $B$), we can perform quantitative analysis on the performance of algorithms. The paper shows that SURF performs as well as SIFT in matching.

### BRIEF (Binary Robust Independent Elementary Features)

In both SIFT and SURF, each dimension is a floating-point number (represented in 4 bytes), and our comparison of feature vectors are done with Euclidean distance. Ideally, we'd want to make feature description and matching even faster.

We can further shorten the descriptor by performing quantisation (converting a continuous number into a discrete number, hence 8 bits for the range $[0, 255]$), or even performing binarization (converting each into a binary number).

Haar wavelets in SURF compare a local region toe another, giving a difference (which is a float). In BRIEF, comparing points $p, q$ will result in a binary value as an output (1 if $q$ is brighter than $p$);

$$\tau(p, q) = \begin{cases} 1 & \text{if } I(p) < I(q) \\ 0 & \text{otherwise} \end{cases}$$

BRIEF randomly sample $n_d$ pairs of points for binary tests; this random pattern is only determined once, and the same pattern will be applied to all interest points. If we set $n_d = 256$, we will perform 256 tests in total, each giving us a single bit (with a total of 32 bytes; SIFT uses 128 bytes, and SURF uses 64 bytes). The descriptor is then a $n_d$-dimensional bitstring.

Note that this is also fast to compute, as we can perform bit shifting (`<< n` denotes a bit shift by `n` places). For example, if we performed 8 binary tests, we could do the following;

```
descriptor = ((I(p1) < I(q1)) << 7) + ((I(p2) < I(q2)) << 6)
           + ((I(p3) < I(q3)) << 5) + ((I(p4) < I(q4)) << 4)
           + ((I(p5) < I(q5)) << 3) + ((I(p6) < I(q6)) << 2)
           + ((I(p7) < I(q7)) << 1) + ((I(p8) < I(q8)) << 0)
```

Not only does it use less memory, the computation only compares two numbers, without calculating gradient orientation like in SIFT, or the intensity difference like in SURF. We also avoid calculating the Euclidean distance, instead we use the Hamming distance. This allows for an efficient calculation by performing a bitwise XOR between two descriptors, and then taking a bit count - both of which can be done extremely quickly on modern CPUs in very few instructions. For example;

`10001001 XOR 11000011 = 01001010`, thus a Hamming distance of 3

However; BRIEF isn't rotation-invariant nor scale-invariant. It assumes that the images taken are from a camera where the movement **only involves translation**. By combining all of the above, BRIEF is approximately 40 times faster than SURF, hence being around 200 times faster than SIFT. Furthermore, if there is no rotation or scaling, BRIEF is comparable to SURF in terms of matching accuracy.

### Image Similarity

Image similarity can be defined by the number of matched point. Intuitively, if two images are similar, there should be a higher number of interest points that can be matched together. See *VisualRank*.

### HOG (Histograms of Oriented Gradient)

We can also extend the idea of feature descriptors (which describe local content centred at a point) to describe the feature of a large region, or even a whole image.

While HOG and SIFT both use gradient orientation histograms for feature descriptions, HOG differs in that it describes features for a large image region (rather than a point). HOG divides a large region into a (dense) grid of cells, describes each cell, and concatenates the local descriptions to form a global description.

In the example, we divide the image into equally spaced cells (each of $8 \times 8 pixels$). 4 of these cells form a block; and we calculate the gradient orientation histogram for a block (thus forming a description of that block). Note that the description vector $\boldsymbol{v}$, which is the concatenation of the 4 histograms (from each cell), is normalised to form a locally normalised descriptor - note the inclusion of a small value $\epsilon$ to improve numerical stability. This is done to prevent issues between differing brightnesses;

$$\boldsymbol{v}_{\text{norm}} = \frac{\boldsymbol{v}}{\sqrt{||\boldsymbol{v}||_2^2 + \epsilon^2}}$$

The block is then shifted along horizontally by one cell (8 pixels in our case), note that there will be some overlap between the blocks. The HOG descriptor is then formed by concatenating all the normalised local descriptors for all blocks.

We can use HOG to perform feature extraction from an input image to a feature representation. Feature extraction transforms an input image into low-dimensional vectors for easier comparison or matching. A classifier (trained with a dataset consisting of images, and often labels) can be used to give an output label to the feature representation.

## Lecture 12 - Image Classification I

This lecture starts with some motivation on image classification. The ImageNet project aims to collect $80,000$ classes, with each class having between 500 to $1,000$ images. The ILSVRC uses a subset of this, with only $1,000$ (simpler) classes used; there are also no overlap between classes, since pairs of classes cannot be ancestors of each other. In ImageNet classes, all classes are organised as a tree. Below the root node, ImageNet has 12 subtrees (with each subtree expanding further into many classes);

- mammal
- bird
- fish
- reptile
- amphibian
- vehicle

- furniture
- musical instrument
- geological formation
- tool
- flower
- fruit

These classes are defined with concepts from our natural language defined by WordNet (a lexical database of English). WordNet groups nouns into sets of cognitive synonyms (synsets), each expressing a **distinct** concept - these synsets will form the leaves of the tree, with the concepts already being organised as a tree.

### Classifier

This lecture is literally just going over stuff in **CO395**. For example, drawing a line as a classifier for data points in 2D space. When labels are not available for training data, we perform unsupervised learning; a common technique is to perform clustering for classification.

### MNIST Example

The example we go through will use the handwritten digit recognition dataset from MNIST - there are $60,000$ training samples and another $10,000$ reserved for testing. Each sample is a $28 \times 28$ image, which is labelled with a class from 0 to 9.

However, we need to perform some pre-processing. We need to first find where the digit is in a large image, normalise the size of the digit to $28 \times 28$, as well as normalising the location (place the mass centre in the image centre). Sometimes slant correction may be performed, shifting each row to force the principal axis to become vertical. MNIST provides both datasets.

There are two ways to extract features, either with hand-crafted techniques (such as HOG or pixel intensities), or learnt features such as CNNs. After which, we perform classification, with classifiers like KNN (once again, see **CO395**).

To define the neighbours, we need a distance metric between two data points $\boldsymbol{x}$ and $\boldsymbol{y}$;

- **Euclidean distance**

  $$D(\boldsymbol{x}, \boldsymbol{y}) = \sqrt{(x_1 - y_1)^2 + \cdots + (x_n - y_n)^2}$$

  In our case, we have $784 = 28 \times 28$ dimensions. If each dimension has different scales, it can be better to normalise the feature vector (for example, weights and heights have different value ranges depending on the units). An example is to normalise each dimension to a Gaussian distribution $\mathcal{N}(0, 1)$, allowing for the dimensions to be treated fairly.

- **cosine distance**

  $$D(\boldsymbol{x}, \boldsymbol{y}) = \frac{\boldsymbol{x} \cdot \boldsymbol{y}}{||\boldsymbol{x}||\ ||\boldsymbol{y}||} = \frac{x_1 y_1 + \cdots + x_n y_n}{\sqrt{\sum\limits_{i=1}^{n} x_i^2}\sqrt{\sum\limits_{i=1}^{n} y_i^2}}$$

- **generalised**

  The Manhattan distance uses the $\ell_1$-norm;

  $$D_1(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{n} |x_i - y_i|$$

  A more general form is the $\ell_p$-norm (note that the Euclidean distance is the $\ell_2$-norm);

  $$D_p(\boldsymbol{x}, \boldsymbol{y}) = \left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{\frac{1}{p}}$$

Using $K = 1$, it can already work quite well. However, we may still have some inaccuracies, where a 7 may be classified as a 3, a 7 being classified as 0, or an 8 being classified as a 5, and so on.

The lecture then goes over hyperparameter tuning for $K$, finding that $K = 3$ is ideal for the MNIST dataset, with a 2.4% error rate. KNN has no training step, is simple (but quite effective), and can also handle multi-class classification easily. However, it is expensive to use KNN in terms of both storage of training data and searching (for an extremely large set, the computational cost can become very expensive).

Suppose we have $N$ training images, and $M$ test images. While we have no training time, the computational cost to classify all $M$ test images is $O(MN)$.

However, we are typically fine with slow training (since we can perform the training on high-powered hardware), but we want the test time to be fast (since it may be deployed on weaker hardware).

### Features

For simple pre-processed images (like in our example), it might be fine to use the $28 \times 28$ pixel intensities as the feature vector. However, generally it may not be ideal to use pixel intensities, since it isn't invariant to scale nor rotation (and is generally a much higher dimension).

## Lecture 13 - Image Classification II

### Linear Classifier

A linear classifier, in 2D, has the form;

$$w_1x_1 + w_2x_2 + b = 0$$

The rule of the classifier assigns a class $c$ on $x$ based on the following;

$$c = \begin{cases} 1 & \text{if } w_1x_1 + w_2x_2 + b \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

For a linear classifier, once we know $\boldsymbol{w}$ and $b$, we can discard the training data (unlike in KNN). This is much faster in test time.

For a more general case, the linear model is as follows (where we have the weights / normal, data / feature, and bias);

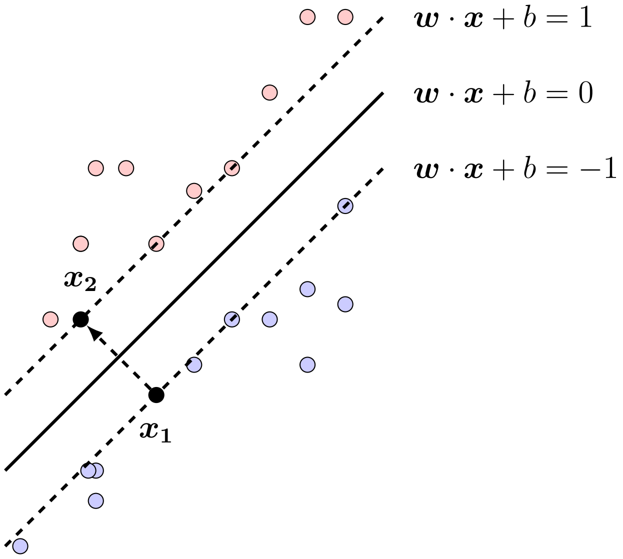
$$\boldsymbol{w} \cdot \boldsymbol{x} + b = 0$$

Similarly, we have the same rule to assign class $c$;

$$c = \begin{cases} 1 & \boldsymbol{w} \cdot \boldsymbol{x} + b \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

### Linear SVM (Support Vector Machine)

To determine the maximum margin hyperplane, we only need to look at the inner most points, which are the **support vectors**. We would like the support vectors to fulfil the equations $\boldsymbol{w} \cdot \boldsymbol{x} + b = -1$ (blue points) and $= 1$ (red points), to allow other points to be easily classified. We want to maximise the margin between the two dashed lines.


$\boldsymbol{w} \cdot \boldsymbol{x} + b = 1$
$\boldsymbol{w} \cdot \boldsymbol{x} + b = 0$
$\boldsymbol{w} \cdot \boldsymbol{x} + b = -1$
$\boldsymbol{x_2}$
$\boldsymbol{x_1}$

We want the margin between the two following lines to be as large as possible, and we can also derive the distance as follows;

$$\begin{aligned}
\boldsymbol{w} \cdot \boldsymbol{x} + b &= 1 \\
\boldsymbol{w} \cdot \boldsymbol{x} + b &= -1 \\
\boldsymbol{w} \cdot \boldsymbol{x_1} + b &= -1 && \text{let this be a point on the } -1 \text{ plane} \\
\boldsymbol{w} \cdot (\boldsymbol{x_1} + m\boldsymbol{n}) + b &= 1 && \text{move the point along the normal by the size of the margin to } \boldsymbol{x_2} \\
m\boldsymbol{w} \cdot \boldsymbol{n} &= 2 && \text{from the previous two equations} \\
\boldsymbol{n} &= \frac{\boldsymbol{w}}{||\boldsymbol{w}||}
\end{aligned}$$

SVM then aims to solve optimise the following;

$$\max_{\boldsymbol{w},b} \frac{2}{||\boldsymbol{w}||}$$

This is subject to the following constraints, for all $i = 1, \dots, N$ (note that $y_i$ denotes the label of the point);

$$\begin{aligned}
\boldsymbol{w} \cdot \boldsymbol{x_i} + b &\geq 1 && \text{if } y_i = 1 \\
\boldsymbol{w} \cdot \boldsymbol{x_i} + b &\leq -1 && \text{if } y_i = -1
\end{aligned}$$

However, we've assumed that all the points are linearly separable, which isn't always the case. In this case, we cannot solve the optimisation problem, since the constraints require the data to be linearly separable. This can be done with slack variables, $\xi_i \geq 0$ (for each data point). This formulates the optimisation problem as

$$\min_{\boldsymbol{w},b} ||\boldsymbol{w}||^2 + C \sum_{i=1}^{N} \xi_i$$

Subject to the following, for all data points ($\forall i$);

- $\xi_i \geq 0$
- $y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) \geq 1 - \xi_i$

Note that $\xi_i$ will be 0 when the point is correctly classified or $0 < \xi_i \leq 1$ for points within the margin (but on the correct side of the hyperplane), otherwise it represents the distance from the margin. In the above, $C$ is a regularisation term (a small $C$ will allow constraints to easily be fulfilled, whereas a large $C$ would make constraints difficult to ignore - typically leading to a tighter margin).

### Optimisation

Note that we can formulate the slack variable as;

$$\xi_i = \underbrace{\max(0, 1 - y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b))}_{\text{hinge loss}}$$

Therefore, we aim to optimise the following function;

$$\min_{\boldsymbol{w},b} ||\boldsymbol{w}||^2 + C \sum_{i=1}^{N} \max(0, 1 - y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b))$$

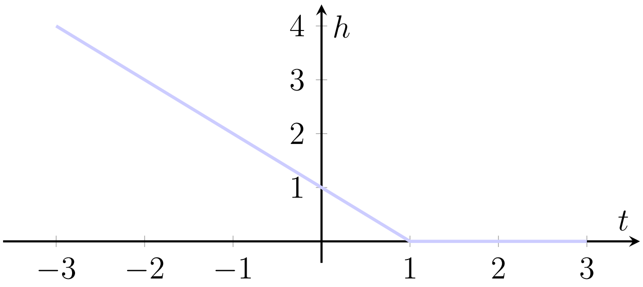
However, note that both of these functions (a quadratic term and a hinge loss function) are **convex** functions (hence we know there is a minimum, and that a local minimum is also a global minimum). Note that the non-negative weighted sum of two convex functions is also a convex function. A convex function $f : X \to R$ satisfies the following;

$$\forall x_1, x_2 \in R,\ \forall t \in [0, 1]\ [f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)]$$

A line joining $(x_1, f(x_1))$ and $(x_2, f(x_2))$ will lie **above** the function curve. There are two approaches to optimise this;

- **gradient descent** used in large scale-optimisation

  Note that the hinge loss function looks like the following;

  While it is not differentiable, we can use a more generalised concept for a gradient, the **subgradient**;

  $$\nabla_{\boldsymbol{w}} h = \begin{cases} -y_i\boldsymbol{x_i} & \text{if } y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b) < 1 \\ 0 & \text{otherwise} \end{cases}$$

  This would be similar for $\nabla_b h$, however, we mainly focus on $\boldsymbol{w}$ for our derivation. For each iteration, we move along the negative of the gradient by some step length $\eta$;

  $$\begin{aligned} \boldsymbol{w}^{(k+1)} &= \boldsymbol{w}^{(k)} - \eta \nabla_{\boldsymbol{w}} \left( ||\boldsymbol{w}||^2 + C \sum_{i=1}^{N} \max(0, 1 - y_i(\boldsymbol{w} \cdot \boldsymbol{x_i} + b)) \right) \\ &= \boldsymbol{w}^{(k)} - \eta \left( 2\boldsymbol{w}^{(k)} + C \sum_{i=1}^{N} \nabla_{\boldsymbol{w}} h \right) \end{aligned}$$

- **Lagrangian duality** (solves the dual problem) often used for SVM libraries

  This is beyond the scope of the course, but works by solving the **dual** problem (given the **primal** problem). See **CO343** for more details about duality.

  In the dual problem, the result of many of the $a_i$ are 0, except for the points located on the margin (the support vectors). The solution of the primal problem is then given by;

  $$\boldsymbol{w} = \sum_{i=1}^{N} a_i y_i \boldsymbol{x_i}$$

  $b$ can also be estimated with the support vectors by substituting back into the equation.

### Classification

At test time, classification can be done according to;

$$c = \begin{cases} 1 & \text{if } f(\boldsymbol{x}) = \boldsymbol{w} \cdot \boldsymbol{x} + b \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

This works quite well if the data can be linearly separable. However, if the points are not linearly separable in the **original** feature space, it may be easier to transform the features. Consider a dataset which has two classes, the first being a circle of points centred around some point $\boldsymbol{x_c}$, and another class being a ring of points around the first class. Using the transformation $\boldsymbol{Phi}(\boldsymbol{x}) = (\boldsymbol{x} - \boldsymbol{x_c})^2$, the first class would have a lower value (since it is closer to $\boldsymbol{x_c}$, which is then squared) after the transformation, whereas the second class would have a much higher value, thus separating the points.

Therefore, if we transform the feature vector using a function $\boldsymbol{\Phi}$, the classifier also must change;

$$f(\boldsymbol{x}) = \underbrace{\left(\sum_i a_i y_i \boldsymbol{\Phi}(\boldsymbol{x_i})\right)}_{\text{transformed } \boldsymbol{w}} \cdot \boldsymbol{\Phi x} + b$$

We can also define the kernel (different from image filtering) as;

$$\boldsymbol{k}(\boldsymbol{x_i}, \boldsymbol{x_j}) = \boldsymbol{\Phi}(\boldsymbol{x_i}) \cdot \boldsymbol{\Phi}(\boldsymbol{x_j})$$

The classifier (which is now a non-linear function) only contains the kernel;

$$f(\boldsymbol{x}) = \sum_i a_i y_i \boldsymbol{k}(\boldsymbol{x_i}, \boldsymbol{x}) + b$$

The common kernels used for SVM are;

- **linear kernel**

  $$\boldsymbol{k}(\boldsymbol{x_i}, \boldsymbol{x_j}) = \boldsymbol{x_i} \cdot \boldsymbol{x_j}$$

- **polynomial kernel**

  $$\boldsymbol{k}(\boldsymbol{x_i}, \boldsymbol{x_j}) = (\boldsymbol{x_i} \cdot \boldsymbol{x_j})^d \text{ or } (1 + \boldsymbol{x_i} \cdot \boldsymbol{x_j})^d$$

- **Gaussian kernel / radial basis function (RBF) kernel**

  $$\boldsymbol{k}(\boldsymbol{x_i}, \boldsymbol{x_j}) = e^{-\frac{||\boldsymbol{x_i} - \boldsymbol{x_j}||^2}{2\sigma^2}}$$

Note that we only have binary classification at the moment. For multi-class classification, we can take several approaches (considering the MNIST digit recognition example);

- **one versus rest**

  In this, we train a classifier for each of the 10 digits, the first classifies between a digit that is 1 and not 1, the second being a classifier between 2 and not 2, and so on. The classifier which produces the highest response at test time determines the result;

  $$c = \underset{k=1,\dots,K}{\operatorname{argmax}} f_k(\boldsymbol{x})$$

- **one versus one**

  In contrast to before, we have a classifier for each pairing of digits - for example, we'd have a classifier for 1 versus 2, 1 versus 3, 2 versus 3, and so on. At test time, each classifier votes for the digit, and the majority vote wins. The number of classifiers we need for $K$ classes is;

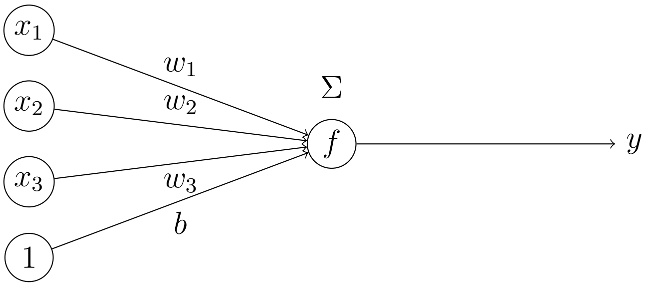
  $$\frac{K(K-1)}{2}$$

## Lecture 14 - Image Classification III

Note that a lot of this is also covered in **CO395**.

### Neural Networks

Neural networks are an algorithm to learn a model from the data for classification / regression problems. Artificial neural networks are **inspired** by biology, but does not exactly model how a neuron works. In the past few decades, neural networks have improved with more layers of connections (deeper), as well has having better hardware for faster computation and larger datasets for training.



$$y = f\left(\sum_{i=1}^{3} w_i x_i + b\right)$$

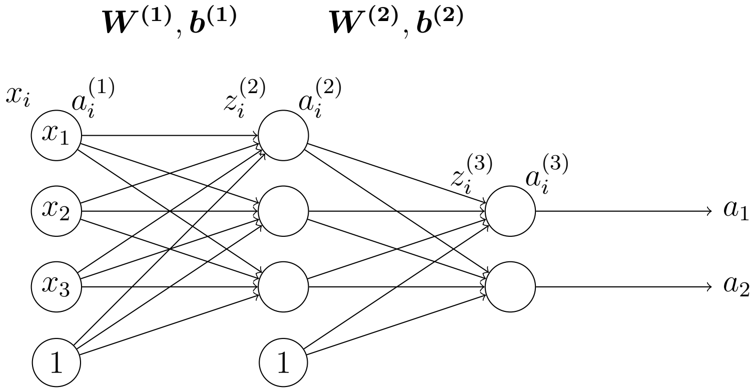
The simplest form of a neural network is a perceptron, using which has a single layer and uses the **Heaviside** step function as an activation function (where $\boldsymbol{w}$ and $b$ are optimised to match the ground truth);

$$y = \begin{cases} 1 & \text{if } \boldsymbol{w} \cdot \boldsymbol{x} + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

Note that the difference (compared to SVMs) is that the non-linearity is added with an activation function $f(\boldsymbol{w} \cdot \boldsymbol{x} + b)$, whereas SVMs add non-linearity through feature transforms; $\boldsymbol{w} \cdot \boldsymbol{\Phi}(\boldsymbol{x}) + b$. A common activation function is the logistic function / sigmoid;

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

A neural network is formed by connecting neurons, where the outputs of a neuron can be the inputs of another;



$\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}$
$\boldsymbol{W}^{(2)}, \boldsymbol{b}^{(2)}$
$x_i$
$a_i^{(1)}$
$z_i^{(2)}$
$a_i^{(2)}$
$z_i^{(3)}$
$a_i^{(3)}$
$x_1$
$x_2$
$x_3$
1
1
$a_1$
$a_2$

The output $a$ can be compared with the ground truth $y$ in a loss function $J$ (which we aim to minimise). A training set is denoted as pairs of data $\boldsymbol{x}$ and ground truth labels;

$$\{(x_1, y_1), (x_2, y_2), \dots, (x_M, y_M)\}$$

An example of a loss function is MSE;

$$J(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{M} \sum_{m=1}^{M} \frac{1}{2} ||a_m - y_m||^2$$

### Optimising

To calculate the output of a network $a$ given some input $x$, we use **forward propagation**. The parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ can be found with gradient descent and **backpropagation** can be used to calculate the gradient.

In the diagram above, we can perform a calculation as follows (applying the activation function to the input of each neuron) - this can also be more compactly written in matrix notation (where the activation function is generally applied element-wise);

$$\begin{aligned}
z_i^{(2)} &= W_{i,1}^{(1)} a_1^{(1)} + W_{i,2}^{(1)} a_2^{(1)} + W_{i,3}^{(1)} a_3^{(1)} + b_i^{(1)} \\
a_i^{(2)} &= f\left(z_i^{(2)}\right) \\
\boldsymbol{z^{(2)}} &= \boldsymbol{W^{(1)}} \boldsymbol{a^{(1)}} + \boldsymbol{b^{(1)}} \\
\boldsymbol{a^{(2)}} &= f\left(\boldsymbol{z^{(2)}}\right)
\end{aligned}$$

More generally, for layer $l + 1$, we do the following;

$$\begin{aligned}
\boldsymbol{z^{(l+1)}} &= \boldsymbol{W^{(l)}} \boldsymbol{a^{(l)}} + \boldsymbol{b^{(l)}} \\
\boldsymbol{a^{(l+1)}} &= f\left(\boldsymbol{z^{(l+1)}}\right)
\end{aligned}$$

Gradient descent can be done as follows, given a learning rate $\alpha$ (note the equals here denotes reassignment, rather than equality);

$$\begin{aligned}
\boldsymbol{W} &= \boldsymbol{W} - \alpha \frac{\partial J}{\partial \boldsymbol{W}} \\
\boldsymbol{b} &= \boldsymbol{b} - \alpha \frac{\partial J}{\partial \boldsymbol{b}}
\end{aligned}$$

The backpropagation algorithm is based on the chain rule. We can work out the derivative of a composition as follows;

$$\begin{aligned}
z &= g(f(x)) \\
y &= f(x) \\
z &= g(y) \\
\frac{\mathrm{d}z}{\mathrm{d}x} &= \frac{\mathrm{d}z}{\mathrm{d}y} \frac{\mathrm{d}y}{\mathrm{d}x}
\end{aligned}$$

Using the running example, we can quite easily derive $\frac{\partial J}{\partial a}$, as well as $\frac{\partial a}{\partial \boldsymbol{W^{(2)}}}$. Therefore, we can use the chain rule;

$$\frac{\partial J}{\partial \boldsymbol{W^{(2)}}} = \frac{\partial J}{\partial a} \frac{\partial a}{\partial \boldsymbol{W^{(2)}}}$$

### Stochastic Gradient Descent

SGD addresses the computational expense of performing forward and backward propagation for all samples in one go. In SGD, a batch of $B$ samples are selected for performing the propagation. For each iteration in the optimisation, we do the following;

1. randomly select a batch of $B$ samples
2. calculate the gradients $\frac{\partial J_B}{\partial \boldsymbol{W}}$ and $\frac{\partial J_B}{\partial \boldsymbol{b}}$ for **only** this batch
3. update the parameters (note that $J_B$ denotes the loss for the $B$ samples)

$$\begin{aligned}
\boldsymbol{W} &= \boldsymbol{W} - \alpha \frac{\partial J_B}{\partial \boldsymbol{W}} \\
\boldsymbol{b} &= \boldsymbol{b} - \alpha \frac{\partial J_B}{\partial \boldsymbol{b}}
\end{aligned}$$

### Classification

Note that MSE works for regression problems, where $y$ is a continuous variable. However, this isn't optimal for classification problems, which can be binary ($y$ is 1 or 0) or multi-class (where $y$ is one of a number of classes).

In binary classification, we only need a single neuron in the output layer. The activation function could be the sigmoid function, which gives an output $\in (0, 1)$, which is in the range of probability. Consider the case where we predict a probability of 0.9 for class 1 (hence 0.1 for class 0). The ground truth has $y = 1$ (therefore the probability of class 1 is 1, and class 0 is 0). A distance metric between the predicted probability and true probability can be used, namely cross entropy.

The cross entropy between a true probability distribution $p$ and an estimated probability distribution $q$ is;

$$H(p, q) = -\sum_i p_i \log(q_i)$$

In the general case (for two classes), we have $p = [y, 1 - y]$ and $q = [f(z), 1 - f(z)]$, hence;

$$H(p, q) = -[y \log(f(z)) + (1 - y) \log(1 - f(z))]$$

For $K$ classes, there are $K$ neurons in the output layer. The activation function used would be softmax, to create a probability distribution;

$$f(z_i) = \frac{e^{z_i}}{\sum\limits_{k=1}^{K} e^{z_k}}$$

The true probability is represented with one-hot (only one element is 1) encoding;

$$p = [y_1, \dots, y_i, \dots, y_K] = [0, \dots, 1, \dots, 0]$$

### Performance

To create more training samples, data augmentation can be performed. This applies affine transformations to the original samples, by performing translation, scaling, squeezing, and shearing.

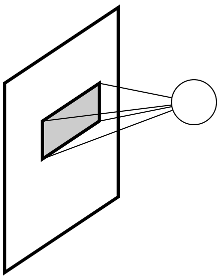
Note that a key limitation of MLP is the high number of parameters required. As such, it's likely to scale poorly for larger images (and even more so if more channels are considered). This is caused by the flattening of a 2D image into a single vector, rather than performing operators on 2D images.

## Lecture 15 - Image Classification IV

### Convolutional Neural Networks

In contrast to MLPs, CNNs assume inputs of 2D images and encode properties such as local connectivity and weight sharing into the architecture, reducing the number of parameters and making the computation more efficient.

When we look at an image, we first look at local regions, process this information, and then combine information from different regions to form a global understanding. In CNNs, each neuron will only see a **receptive field** (small local region) in the layer before it - this is also called local connectivity (reducing the number of parameters).



Note in the example above, a neuron only depends on a small region in the previous layer, whereas a fully connect layer in an MLP depends on **all** the neurons in the previous layer.

### Convolutional Layer

In a CNN, convolutional layers are the basic building blocks. Consider an input image of size $X \times Y \times C$ (note that $C$ denotes the number of channels, 3 for RGB, and 1 for greyscale). If a neuron required a $5 \times 5 \times C$ cuboid of the input, it would have $5 \times 5 \times C + 1$ parameters (including one for bias).

The input and output of the neuron are defined as follows (note that here we sum over 3 axes);

$$\begin{aligned} z &= \sum_{i,j,k} \boldsymbol{W}_{i,j,k} \boldsymbol{x}_{i,j,k} + b \\ a &= f(z) \end{aligned}$$

However, we can add more neurons to form a $D \times 1 \times 1$ cuboid of output (where $D$ represents the depth). Each of these neurons has its own set of weights and biases;

$$a_n = f\left(\sum_{i,j,k} \boldsymbol{W}_{\boldsymbol{n}i,j,k} \boldsymbol{x}_{i,j,k} + b_n\right)$$

This window can be moved across the input image, which gives an output cuboid of size $X \times Y \times D$ - note that all of the windows use the same weights, allowing us to have a total of $5 \times 5 \times C \times D$ parameters for connection weights, and another $D$ for bias. The convolutional kernel $\boldsymbol{W}$ has four dimensions - output depth, kernel width, kernel height, input depth;

$$a_d = f\left(\sum_{i,j,k} \boldsymbol{W}_{d,i,j,k} \boldsymbol{x}_{i,j,k} + b_d\right)$$

The output feature map has three dimensions, the output depth, as well as the width and height.

We can set a number of parameters for the layer;

- **padding**

  Recall that applying a $3 \times 3$ kernel on an image of shape $X \times Y$ will result in an output of $(X - 2) \times (Y - 2)$. Adding a padding of $p = 1$ gives a border of zeroes around the image. Given an input shape of $X_{\text{in}} \times Y_{\text{in}}$, a $k \times k$ kernel, $p \times p$ padding, we obtain the following output shape;

  $$X_{\text{out}} = X_{\text{in}} + 2p - k + 1$$

- **stride**

  In order to move the window faster and obtain a downsampled grid, we can use a stride. For example, with $s = 2$, it the window to every other pixel. Consider the following; note that we have a padding $p = 1$ and stride $s = 2$ (crosses denote the centre of the kernel);

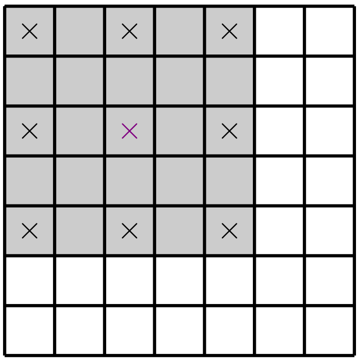
| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | × | | × | | × | 0 |
| 0 | | | | | | 0 |
| 0 | × | | × | | × | 0 |
| 0 | | | | | | 0 |
| 0 | × | | × | | × | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

This can be combined with the padding shape to obtain the following (given a stride of $s \times s$);

$$X_{\text{out}} = \left\lfloor \frac{X_{\text{in}} + 2p - k}{s} \right\rfloor + 1$$

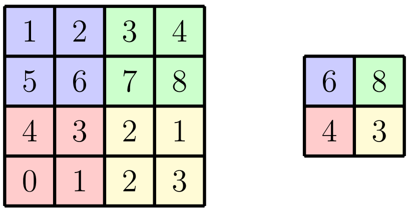
- **dilation**

  For a neuron to have a larger receptive field, we can either keep the kernel size the same (but downsample the input image), or increase the kernel size (thus also increasing the number of parameters). Dilation aims to increase the region size without downsampling or adding more parameters. Consider the following - note that a violet cross denotes the centre of the kernel, and regular crosses denote points which are taken into account (this is a $3 \times 3$ kernel, with dilation $d = 2$, hence we look at every other pixel);



### Pooling Layer

A pooling layer is another building block of CNNs, which make feature maps / representations smaller (similar to image downsampling). Consider the following example, which performs max pooling with a $2 \times 2$ window and stride of 2;



| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 4 | 3 | 2 | 1 |
| 0 | 1 | 2 | 3 |

| | |
|---|---|
| 6 | 8 |
| 4 | 3 |

Another form of pooling is average pooling.

### LeNet-5

This uses 7 layers in total (including the output layer but excluding the input layer). A layer beginning with C denotes a convolutional layer, S denotes a pooling (subsampling) layer, and F denotes a fully connected layer. The input image is of size $32 \times 32$.

| | |
|---|---|
| • C1 | $28 \times 28$ with depth of 6, using $5 \times 5$ convolutions |
| • S2 | $14 \times 14$ with depth of 6, max pooling |
| • C3 | $10 \times 10$ with depth of 16, using $5 \times 5$ convolutions |
| • S4 | $5 \times 5$ with depth of 16 |
| • C5 | 120 neurons |
| • F6 | 84 neurons |
| • output | 10 classes |

The same techniques can be used to optimise a loss function, as convolutional layers are mathematically similar to layers in MLP (just with more dimensions).

**Deep Neural Networks**

A factor in the improvement of deep neural networks is hardware; namely GPUs. They are ideal for training neural networks as convolutional operations can be performed in parallel using thousands of simpler (relative to CPU) cores on a GPU.

Another factor is improvements in optimisation. There are two main problems with previously mentioned approach; exploding gradient (where the gradient becomes too large for the algorithm to converge) and vanishing gradient (where the gradient becomes too small for the weights to change values). An example of exploding gradient is when the current point is very close to a 'cliff' - the gradient descent may overshoot up the cliff, causing it to not converge. One solution is to perform gradient clipping (by value), which does the following for each element of the gradient $\boldsymbol{g}$;

$$g_i = \begin{cases} v_{\text{min}} & \text{if } g_i < v_{\text{min}} \\ v_{\text{max}} & \text{if } g_i > v_{\text{max}} \\ g_i & \text{otherwise} \end{cases}$$

Another approach is to clip by the $\ell_2$-norm;

$$\boldsymbol{g} = \begin{cases} \frac{\boldsymbol{g}}{||\boldsymbol{g}||}v & \text{if } ||\boldsymbol{g}|| > v \\ \boldsymbol{g} & \text{otherwise} \end{cases}$$

On the other hand, the vanishing gradient problem can be caused on either extremes of the sigmoid function. Recall that gradients are multiplied, therefore if any are zero (or close to zero), the propagated derivative will also be small;

$$\begin{aligned} f(z) &= \frac{1}{1 + e^{-z}} && \text{sigmoid function} \\ f'(z) &= \frac{e^{-z}}{(1 + e^{-z})^2} \\ &= f(z)(1 - f(z)) && \text{small if either are close to 0 or 1} \\ \frac{\partial J}{\partial \boldsymbol{z}^{(\boldsymbol{l})}} &= \left( \left(\boldsymbol{W}^{(\boldsymbol{l})}\right)^\top \frac{\partial J}{\partial \boldsymbol{z}^{(\boldsymbol{l+1})}} \right) \circ f'\left(\boldsymbol{z}^{(\boldsymbol{l})}\right) && \text{propagated derivative} \end{aligned}$$

This can be addressed with different activation functions, for example;

- **rectified linear unit (ReLU)**

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

  This gives the following subgradient;

  $$f'(z) = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases}$$

  This prevents gradients from vanishing when $z$ is large, however it becomes 0 with a negative $z$. This outperforms sigmoid as an activation function for deep neural networks.

- **leaky ReLU**

  $$f(z) = \begin{cases} 0.01z & z < 0 \\ z & z \geq 0 \end{cases}$$

  Similar to ReLU, but gives some gradient for negative values,

- **parametric ReLU (PReLU)**

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

  This is similar to leaky ReLU, however the parameter $a$ is learnt in training.

- **exponential linear unit (ELU)**

$$f(z) = \begin{cases} a(e^z - 1) & z < 0 \\ z & z \geq 0 \end{cases}$$

### AlexNet and VGG

AlexNet is generally quite similar to LeNet, but is deeper. It also uses ReLU for activation, and performs data augmentation by cropping random $224 \times 224$ regions from the original $256 \times 256$ images, as well as performing horizontal reflection and perturbing RGB values.

VGG only uses $3 \times 3$ convolutional kernels, rather than larger kernels. The convolution of two $3 \times 3$ kernels is equivalent to a $5 \times 5$, three kernels is equivalent to $7 \times 7$, and so on - this is deeper and reduces the number of parameters. In addition, more non-linearity can be added as activation functions can be used after every small kernel.

## Lecture 16 - Object Detection

When we formulate the problem for image classification, we develop a classifier which analyses the input image and gives a class label as an output (which may give some probability of it being a certain class). However, with object detection, we want to develop a detector which tells us the label, but also a bounding box $(x, y, w, h)$, which specifies the top left (or centre) and the size of the box.

We've built a neural network that can perform classification. If we move a sliding window across a large image, and move a sliding window across the image (treating the windows as smaller images), we can find regions. At each window, we perform two tasks;

- classification (e.g. whether this is an animal or not)
- localisation (bounding box coordinates) - the sliding window already gives some localisation, however we want to refine this with CNN features

The CNN gives an output with two branches; the first is fully connected to $K$ classes (which provides a class score) compared against the ground truth class label (with cross entropy), and another fully connected to the four variables of the bounding box, which is compared against the ground truth bounding box. These are then added together to be optimised.

However, it may be expensive to apply a CNN to each pixel location on the image. Object detection is split into two-stage detection;

- **region proposal** initial guesses to propose some possible interesting regions
- **detector** apply network to classify and predict bounding boxes of these regions

One-stage detection skips guessing, but rather divides the image into grid cells, and then performs classification and bounding on these cells.

### Selective Search

This is rarely used nowadays and is a traditional approach before the development of CNNs. We can look at image features such as greyscale intensities or gradients to separate the image into regions with similar features.

### Faster R-CNN

A region proposal network (RPN) is used instead (which is much faster compared to selective search). The input image is fed into a convolutional network, which gives a feature map describing the image. This then goes into a region proposal network which gives interesting regions to be looked at. In stage

2, there is a classifier / detector which looks closely at the proposed regions, and can also refine the shape of the bounding box.

AlexNet / VGG-16 are known as backbone networks (and are often pretrained), with the last convolutional layer being used as a convolutional feature map. Both stages rely on this feature map. Each pixel of the feature map is a high-dimensional feature vector, describing the content of a small region in the input image (which may tell us whether a region is interesting).

Using this feature map (such as `conv5` in VGG), we can use a $3 \times 3$ sliding window to perform binary classification at each location, giving it 0 if it isn't interesting, and 1 if it is. It handles objects of different sizes / aspect ratio by making $k$ predictions (bounding boxes). For example, this is done in 3 scales $128^2, 256^2, 512^2$, and 3 aspect ratios $1:1, 1:2, 2:1$ - these bounding boxes are called anchors. For a convolutional feature map of $W \times H$, $W \times H \times k$ anchors are predicted (in our case $k = 9 = 3 \times 3$) - only the highest scoring boxes are kept (based on objectness score, generated by the network).

The loss is defined as;

$$L(p, t) = \underbrace{\sum_{i=1}^{n_\text{anchor}} L_\text{cls}(p_i, p_i^*)}_{\text{classification loss}} + \underbrace{\lambda \sum_{i=1}^{n_\text{anchor}} 1_{y=1} L_\text{loc}(t_i, t_i^*)}_{\text{localisation loss}}$$

The classification loss function $L_\text{cls}$ is can be calculated with cross entropy, by comparing the predicted objectness score $p_i$ with the ground truth $p_i^*$ (a binary value, 1 for a interesting location and 0 otherwise). The localisation loss attempts to refine the bounding box. We can either directly predict the values for a bounding box (the centre coordinates $x, y$ and size $w, h$), or the transformation parameters $\boldsymbol{t}$ from the anchor into the ground truth bounding box (note that if $\boldsymbol{t} = \mathbf{0}$, then the predicted bounding box is equivalent to the anchor);

$$\begin{aligned}
\text{anchor} &= (x_a, y_a, w_a, h_a) \\
\text{predicted bounding box} &= (x, y, w, h) \\
\text{predicted transformation} &= (t_x, t_y, t_w, t_h) \\
t_x &= \frac{x - x_a}{w_a} \\
t_y &= \frac{y - y_a}{h_a} \\
t_w &= \log\left(\frac{w}{w_a}\right) \\
t_g &= \log\left(\frac{h}{h_a}\right)
\end{aligned}$$

It may be easier to predict the transformation than the bounding box directly, since the predicted bounding box components may be arbitrary. Note that the localisation loss compares vectors of continuous numbers, therefore MSE can be used as the loss function.

Now that we have the locations and sizes of the objects, we can look closely at them in the feature map. The features from the region (which is no longer just a $3 \times 3$ window) can be used in a classifier (RoI pooling). In the RoI (region of interest) pooling layer, the location and size is calculated on the feature map - the features in this region are then converted into a fixed size to be provided into the classifier.

For each RoI, the classifier predicts the label class and refines the bounding box estimate;

$$L(p, t) = L_\text{cls}(p, y) + \lambda \cdot 1_{y \geq 0} L_\text{loc}(t, t^*)$$

Note that the classification loss is now a multi-class classification problem.

RPN and detection network are fairly similar. However, the input to a RP is a $3 \times 3$ window, whereas the input to a detection network is a proposed region (leading to more accurate features). RPN is also class-agnostic (a binary value for whether the region is interesting or not) whereas the detection network classifies the region into a number of classes. The detection network doesn't need anchors (since we have a rough size from the proposal), whereas RPN does since we do not know the shape of the object.

### One-Stage Object Detection

Region proposal and classification can be done in a single go with one-stage object detection methods, such as YOLO and SSD. The region proposal network, in two-stage, uses a binary value for the classification loss. This is changed into a multi-class classifier, which predicts the object for each anchor.

Note that Faster R-CNN is more accurate but slower, and vice versa for SSD. Faster R-CNN is more accurate as it first roughly estimates the region size, before looking closely at features and then refining it.

### Performance

Note that backbone networks can be changed out to improve performance. The meta architecture (Faster RCNN, R-FCN, SSD, etc.) combined with the backbone network can be used to observe trade-offs between speed and accuracy.

## Lecture 17 - Image Segmentation

This is an even more detailed understanding than a bounding box; where there is a label class for each pixel (0 for background, etc.) - this can generate a segmentation mask. Instance segmentation gives each instance (such as each person) a unique label.

### Thresholding

This is one of the simplest methods for segmentation, and converts a greyscale image into a binary label map. At each pixel, the label is defined as follows;

$$f(x) = \begin{cases} 1 & \text{if } I(x) \geq \text{threshold} \\ 0 & \text{otherwise} \end{cases}$$

This requires no training data, and the only parameter we require is the threshold. This relies on the intensity histogram being split into two classes, which can easily be split with a threshold.

### K-means

Generally, there can be more than two classes. K-means can provide a simple method for clustering by estimating cluster parameters. Each cluster can be represented by its centre, with each data point (pixel intensity) being associated to the nearest centre. The optimal centres minimise the intra-class variance (note $C_k$ denotes data points which are associated with cluster $k$);

$$\min \sum_{k=1}^{K} \sum_{x \in C_k} (x - \mu_k)^2$$

This can be reformulated with $\delta_{x,k}$, which denotes membership (whether $x$ is assigned to cluster $k$ or not) - the two unknowns are the membership and the cluster centre.

$$\min \sum_{k=1}^{K} \sum_{x} \delta_{x,k} (x - \mu_k)^2$$

If we know $\delta_{x,k}$ (the association between data and clusters), we can work out $\mu_k$, and vice versa. However, we don't know either of these initially.

This can be done iteratively, by taking some initial guess for **either** of these parameters. The algorithm is already covered in **CO395** (take initial guess for cluster centres, compute $\delta_{x,k}$, update $\mu_k$, and so on).

In order to determine the number of clusters ($K$), we can calculate the intra-class variance for each value of $K$. The clustering can also be performed on other features, such as colour similarity (possibly combined with position). $x$ becomes a feature vector, rather than a scalar.

### Gaussian Mixture Model (GMM)

K-means performs a hard assignment of a data point $x$ to a cluster $k$. GMMs allow for soft assignment by assuming each cluster is a Gaussian distribution. The probability that $x_j$ is in cluster $k$ is;

$$P(y_j = k \mid x_i, \pi_k, \mu_k, \sigma_k) = \pi_k \cdot \frac{1}{\sqrt{2\pi}\sigma_k} e^{-\frac{(x-\mu_k)^2}{2\sigma_k^2}}$$

The process for updating this is very similar to K-means (algorithm is detailed in **CO395** again). The parameters are updated as follows ($j \in [1, N]$);

$$\begin{aligned}
\pi_k &= \frac{\sum\limits_j P(y_j = k)}{N} \\
\mu_k &= \frac{\sum\limits_j P(y_j = k) \cdot x_j}{\sum\limits_j P(y_j = k)} \\
\sigma_k^2 &= \frac{\sum\limits_j P(y_j = k) \cdot (x_j - \mu_k)^2}{\sum\limits_j P(y_j = k)}
\end{aligned}$$

The class is then assigned by maximum probability;

$$c = \underset{c}{\operatorname{argmax}}\, P(y_j = c \mid x_j, \pi_c, \mu_c, \sigma_c)$$

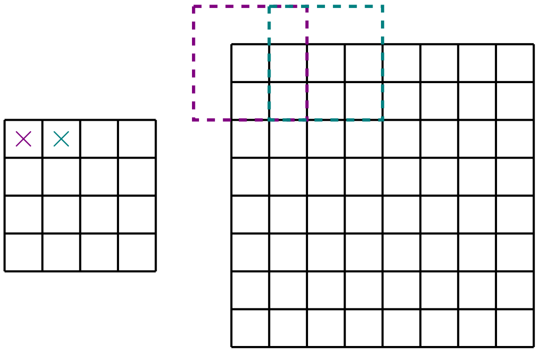
This method is used in *Microsoft PowerPoint*, which as two clusters ($K = 2$). Segmentation is then performed with constraints (such as smoothness, or user editing using GrabCut).

### CNNs

The pixel-wise segmentation map will have to manually be annotated by humans (which can be used as ground truth). For simplicity, we assume we have this data (pairs of images with their respective segmentation maps). The ideas are the same as before; moving a window across an image and applying the classification network to each pixel. However, applying the network to many windows is expensive - we would rather develop a network which provides a pixel-wise probability map (rather than single probability vectors).

The last feature map (before the fully connected layers) of AlexNet is of size $13 \times 13$, downsampled by a factor of 16 from $224 \times 224$. Each pixel in this feature map encodes information regarding a $16 \times 16$ region in the input image. Instead of using fully connected layers to obtain probability vectors, we can obtain pixel-wise classification results from the feature map. A process called **convolutionalization** replaces the fully connected layers with convolutional layers (hence creating a fully convolutional network) to enable the network to produce a pixel-wise probability map (each class will have a heat map, whereas before a class would be a single value in the probability vector). Note that the depth of the final convolutional network represents the number of classes (the ground truth would be one-hot encoded heat maps).

Note that we will need to apply an upsampling operation to obtain a pixel-wise prediction (since the feature map is downsampled). A transposed convolution can be used to achieve this. Consider the following, which has a stride of 2 (the windows jump by 2) - each pixel in the input contributes to a $3 \times 3$ window in the output, with some weights;



A convolution in 1D can be represented as follows (note that the input is $\boldsymbol{x}$, which denotes a vertical region, and the kernel is a $3 \times 1$ region);

$$\begin{bmatrix} w_1 & w_2 & w_3 & 0 \\ 0 & w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

A transposed convolution can then be represented as;

$$\begin{bmatrix} w_1 & 0 \\ w_2 & w_1 \\ w_3 & w_2 \\ 0 & w_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

A classification network can be transformed into a segmentation network by replacing the fully connected layers with convolutional and transposed convolutional layers.

Segmentation is a pixel-wise classification problem; therefore at each pixel we can define a classification loss (such as cross entropy) - the segmentation loss would then be the average classification loss. The segmentation loss can be optimised to train the network.

It is possible for the same network to perform image segmentation as well as object detection. The input is the convolutional feature map, which is fed into two branches - detection (combining classification and localisation) and segmentation (which is fully convolutional).

## Lecture 18 - Motion I

### Optic Flow

Optic flow methods estimate optic flow. Optic flow (motion) is how brightness / intensity moves in videos. The output of this is a flow field, with the same dimension as the video, and describes displacement at each pixel.

A video is a 2D-$t$ image sequence captured over time, hence it is a function of three parameters; space $(x, y)$, and time $t$. For each point $(x, y)$ at time $t$, we want to estimate its corresponding position $(x + u, y + v)$ at time $t + 1$. To do this, we make three assumptions;

- **brightness constancy**

  If we have two time frames from a video, we assume that two corresponding points will have the same greyscale intensities.

- **small motion**

  The motion between two successive frames is very small.

- **spatial coherence**

  For pixels which are connected to each other (neighbours), the motion should be similar.

### Optic Flow Constraint

We have the following, based on the brightness constancy assumption (where $I$ denotes intensity, $(x, y, t)$ denotes spatial and temporal coordinates, and $(u, v)$ denotes displacement);

$$\underbrace{I(x+u, y+v, t+1)}_{\text{new position}} = I(x, y, t)$$

We can perform first-order Taylor expansion (disregarding the higher-order terms) on the left-hand term, based on the small motion assumption;

$$I(x+u, y+v, t+1) \approx I(x, y, t) + \frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \frac{\partial I}{\partial t}$$

Combining those two equations leads to the following, the **optical flow constraint equation**;

$$\frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \frac{\partial I}{\partial t} = 0$$

Note that if we use $I_x, I_y, I_t$ to denote partial derivatives, this can be written as the following, where we have the spatial gradient, displacement, and temporal gradient;

$$I_x u + I_y v + I_t = 0$$

### Lucas-Kanade Method

Since there are two unknowns $(u, v)$, this is an underdetermined system, which requires another constraint. The Lucas-Kanade method addresses this with the spatial coherence assumption (where we assume the flow is constant within a small neighbourhood). The optic flow constraint equation at each pixel $p$;

$$\begin{bmatrix} I_x(p) & I_y(p) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = -I_t(p)$$

We assume that $u, v$ are the same for a small neighbourhood, such as a $3 \times 3$ window, giving a system of linear equations;

$$\underbrace{\begin{bmatrix} I_x(p_1) & I_y(p_1) \\ I_x(p_2) & I_y(p_2) \\ \vdots & \vdots \\ I_x(p_N) & I_y(p_N) \end{bmatrix}}_{\boldsymbol{A}} \underbrace{\begin{bmatrix} u \\ v \end{bmatrix}}_{\boldsymbol{x}} = - \underbrace{\begin{bmatrix} I_t(p_1) \\ I_t(p_2) \\ \vdots \\ I_t(p_N) \end{bmatrix}}_{\boldsymbol{b}}$$

However, this system is now overdetermined (more equations than unknowns) - the unknowns can be estimated with the least square method;

$$x = \operatorname*{argmin}_x ||\boldsymbol{Ax} - \boldsymbol{b}||^2$$

The least square solution can be obtained as follows;

$$\begin{aligned}
\boldsymbol{x} &= \left(\boldsymbol{A}^\top \boldsymbol{A}\right)^{-1} \boldsymbol{A}^\top \boldsymbol{b} && \text{Moore-Penrose / pseudo inverse} \\
&= \begin{bmatrix} u \\ v \end{bmatrix} \\
\boldsymbol{A}^\top \boldsymbol{A} &= \sum_p \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \\
\boldsymbol{A}^\top \boldsymbol{b} &= -\sum_p \begin{bmatrix} I_x I_t \\ I_y I_t \end{bmatrix}
\end{aligned}$$

Note that the matrix $\boldsymbol{A}^\top \boldsymbol{A}$ also appears in the Harris detector. Optic flow (estimating motion) is related to the cornerness response (estimating corners). In linear algebra, we know that the condition number is the ratio between its minimum and maximum eigenvalues;

$$\text{cond}\left(\boldsymbol{A}^\top \boldsymbol{A}\right) = \frac{|\lambda_{\text{max}}|}{|\lambda_{\text{min}}|}$$

For flat regions, we are likely to have a large condition number due to a small $\lambda_{\text{min}}$, hence calculating the inverse becomes sensitive to small changes. On the other hand, due to larger $\lambda_{\text{min}}$ in corners, we are likely to have small condition numbers - this leads to the inverse being more numerically stable and more robust flow estimation. Intuitively, the motion of a corner is easier to track than the motion on an edge or a corner.

This is known as the **aperture problem**. The motion of a line is ambiguous when looking through an aperture, since the motion component parallel to the line cannot be inferred based on visual input. However, the motion of a corner is clearer to define, even through an aperture.

To perform the Lucas-Kanade method, we first need to calculate the image gradients $I_x, I_y$, either with finite differences (between neighbouring pixels), or Sobel / Prewitt filters, as well as $I_t$ - calculated with the finite difference between neighbouring time frames. Then, for each pixel, we apply the least square solution as mentioned above.

Note that in the small motion assumption, we use $\approx$, rather than $=$, since we are ignoring higher-order terms. This can occasionally influence the performance. This assumption no longer holds when the motion is large (when $(u, v)$ are large). An approach to estimate this is to introduce a multi-scale / multi-resolution framework. The idea is that the motion will look small in a downsampled resolution, even when it is large in the original resolution. For example, a displacement of 8 pixels becomes a displacement of 1 pixel at scale 4 (where the image has been downsampled by a factor of 8). An image pyramid is a multi-scale representation of an image, where the first layer (scale 1) is the original resolution. The next layer above is downsampled by a factor of 2 (scale 2), and the image after is another factor of 2 (hence scale 3 being downsampled by a factor of 4 overall).

Using two pyramids, for two consecutive frames, the flow is estimated from the coarsest resolution (highest scale) downwards. After this is done, incremental flow is estimated on the the next resolution; the final flow field is obtained by summing all incremental flows.

For example, assume we obtain an estimate $u^{(4)}, v^{(4)}$ for images $I$ and $J$ at scale 4. This becomes $2u^{(4)}, 2v^{(4)}$ at scale 3 (the initial values). Then incremental flow is calculated for the two following images;

- $I(x, y)$ at time $t$
- $J_{\text{warped}} = J(x + 2u^{(4)}, y + 2v^{(4)})$ at time $t + 1$

This is then accumulated.

The multi-scale Lucas-Kanade method is as follows, where the following is done for each scale $l$ (from coarse to fine);

1. upsample flow estimate from previous scale

$$\begin{aligned}u^{(l+1)} &= 2u^{(l+1)} \\ v^{(l+1)} &= 2v^{(l+1)}\end{aligned}$$

2. compute warped image

$$J^l_{\text{warped}} = J^l(x + u^{(l+1)}, y + v^{(l+1)})$$

3. compute $I_x, I_y, I_t$ for image $I^l$ and $J^l_{\text{warped}}$ at this scale; $I_x, I_y$ are computed from $I^l$

$$I_t = J^l_{\text{warped}}(x, y) - I^l(x, y)$$

4. estimate incremental flow

$$\begin{bmatrix} u^\delta \\ v^\delta \end{bmatrix} = \left(\boldsymbol{A}^\top \boldsymbol{A}\right)^{-1} \boldsymbol{A}^\top \boldsymbol{b}$$

5. update flow at this scale

$$\begin{aligned}u^{(l)} &= u^{(l+1)} + u^\delta \\ v^{(l)} &= v^{(l+1)} + v^\delta\end{aligned}$$

**Horn-Schunck Method**

This is an optimisation-based method, defining a global energy functional (a cost function) for the flow. Note that we still have the optic flow constraint equation, however this defines a global energy functional for all the pixels, rather an solving the constraint for each pixel as in Lucas-Kanade;

$$E(u, v) = \iint\limits_{(x,y)\in\Omega} \underbrace{(I_x u + I_y v + I_t)^2}_{(1)} + \underbrace{\alpha(|\nabla u|^2 + |\nabla v|^2)}_{(2)} \, \mathrm{d}x\mathrm{d}y$$

Note that (1) relates to the optic flow constraint equation, as we want to minimise it (thus bringing it close to 0). On the other hand (2) is a smoothness term; minimising the gradients for $u$ and $v$, thus minimising the change between adjacent vectors in both $u$ and $v$ (leading to neighbouring pixels having similar values). Together, this aims to enforce the optic flow constraint, and achieve a smooth flow field. This is integrated over all pixels in image $\Omega$. Note that $u = u(x, y)$ and $v = v(x, y)$, which are unknown **functions** defined on the image pixels.

A function is a mapping from a variable to a value, such as $f(x) = x^2$ (mapping a variable $x$ to a value). This can be minimised with regard to $x$;

$$\min_x f(x)$$

On the other hand, a functional is a mapping from a function to a value, such as $E(f) = \int f(x)\mathrm{d}x$. This can be minimised with regard to $f$;

$$\min_f E(f)$$

If $L(x, y, u, u_x, u_y)$ is the integrand (term inside integral) of the energy, minimising the functional comes down to solving the associated Euler-Lagrange equation (outside the scope of this course);

$$\begin{aligned}\frac{\partial L}{\partial u} - \frac{\partial}{\partial x}\frac{\partial L}{\partial u_x} - \frac{\partial}{\partial y}\frac{\partial L}{\partial u_y} &= 0 \\ \frac{\partial L}{\partial v} - \frac{\partial}{\partial x}\frac{\partial L}{\partial v_x} - \frac{\partial}{\partial y}\frac{\partial L}{\partial v_y} &= 0\end{aligned}$$

Therefore we have the following (note that $\Delta$ is the Laplace operator, $\Delta := \partial_{xx} + \partial_{yy}$ - the sum of the second order derivatives);

$$\begin{aligned}
&(I_x u + I_y v + I_t)I_x - \alpha\Delta u = 0\\
&(I_x u + I_y v + I_t)I_y - \alpha\Delta v = 0
\end{aligned}$$

However, the Laplacian can be approximated using finite differences, where we have $\bar{u}$ being the local average in a small region (this is shown in the slides);

$$\Delta u = \bar{u} - u$$

Plugging this into the above, we can obtain the following (after reorganising), giving an equation system for $u$ and $v$;

$$\begin{aligned}
&(I_x^2 + \alpha)u + I_x I_y v = \alpha\bar{u} - I_x I_t\\
&I_x I_y u + (I_y^2 + \alpha)v = \alpha\bar{v} - I_y I_t
\end{aligned}$$

This can be solved to obtain the following, which allows us to estimate $u, v, \bar{u}, \bar{v}$ iteratively (with $u$ starting at 0, giving an estimate for $\bar{u}$, which can be used to estimate $u$, and so on);

$$\begin{aligned}
u &= \bar{u} - \frac{I_x(I_x\bar{u} + I_y\bar{v} + I_t)}{I_x^2 + I_y^2 + \alpha}\\
v &= \bar{v} - \frac{I_y(I_x\bar{u} + I_y\bar{v} + I_t)}{I_x^2 + I_y^2 + \alpha}
\end{aligned}$$

The implementation of the method is as follows;

1. compute the gradients $I_x, I_y, I_t$
2. initialise the flow field $u = 0$ and $v = 0$
3. for each iteration $k$

   a. calculate the average flow field
   $$\bar{u}^{(k)}, \bar{v}^{(k)}$$

   b. update the flow field
   $$\begin{aligned}
   u^{(k+1)} &= \bar{u}^{(k)} - \frac{I_x(I_x\bar{u}^{(k)} + I_y\bar{v}^{(k)} + I_t)}{I_x^2 + I_y^2 + \alpha}\\
   v^{(k+1)} &= \bar{v}^{(k)} - \frac{I_y(I_x\bar{u}^{(k)} + I_y\bar{v}^{(k)} + I_t)}{I_x^2 + I_y^2 + \alpha}
   \end{aligned}$$

4. terminate if change of the value is smaller than a threshold / maximum number of iterations reached

### Learning-Based Flow Methods

Recently, learning-based flow methods have become popular, where a CNN takes in an input (which is two frames of an image, at times $t$ and $t + 1$), and outputs a flow field. A flow field can be visualised with a displacement vector (an arrow) at each pixel. Another approach is to use the colour wheel, where the hue denotes the flow orientation, and the saturation denotes the flow magnitude (less saturated for low magnitude).

The ground truth for optic flow fields can be generated with external data, or by generating synthetic data. The former can be done by projecting movements between 3D point clouds onto a 2D image plane. Another approach is to generate images by moving and rendering 3D objects, where we know the ground truth flow field.

The average end-point error ($EE$) can be used to evaluate the performance of a method, if the ground truth is available, by using the difference between displacement vectors;

$$EE = \frac{1}{N}\sum_{x,y}\sqrt{(u(x,y)-u_{GT}(x,y))^2+(v(x,y)-v_{GT}(x,y))^2}$$

Another method is to warp image $I$ to frame $J$, using the estimated flow field. If the field is accurate, the images should be almost identical (which can be quantified by calculating the MSE between $I$ and $J$).

## Lecture 19 - Motion II

For object tracking, we don't care about the movement for each pixel, but rather the displacement vector for each bounding box.

### Lucas-Kanade Tracker

We use the same assumptions we made in optic flow, of constant brightness and small motion. It aims to estimate the motion from the **template image** $I$ (time $t$) to image $J$ (in the next time frame, time $t+1$).

$$\min_{u,v} E(u,v) = \overbrace{\sum_x \sum_u \underbrace{(I(x,y)-J(x+u,y+v))^2}_{\text{squared difference between images}}}^{\text{sum over all pixels}}$$

We assume the template moves by a vector $(u,v)$, and assume that it will become the same as image $I$. Our goal is to find how the template moves from frame to frame, hence finding the position of the template in successive time frames.

To optimise this, we can differentiate $E$ with respect to both of the variables, and set the derivatives to be 0;

$$\frac{\partial E}{\partial u} = -2\sum_x\sum_y[I(x,y)-J(x+u,y+v)]\frac{\partial J}{\partial x} = 0$$

$$\frac{\partial E}{\partial v} = -2\sum_x\sum_y[I(x,y)-J(x+u,y+v)]\frac{\partial J}{\partial y} = 0$$

Note that we apply the chain rule to obtain the $\frac{\partial J}{\partial x}$ term (and same for the second equation);

$$\frac{\partial(-J)}{\partial u} = -\frac{\partial J(x+u)}{\partial(x+u)}\cdot\underbrace{\frac{\partial(x+u)}{\partial u}}_{=1}$$

Note that we can use Taylor expansion, since we have the small motion assumption, which gives us the following;

$$\frac{\partial E}{\partial u} = -2\sum_x\sum_y[I(x,y)-J(x,y)-\frac{\partial J}{\partial x}u-\frac{\partial J}{\partial y}v]\frac{\partial J}{\partial x} = 0$$

$$\frac{\partial E}{\partial v} = -2\sum_x\sum_y[I(x,y)-J(x,y)-\frac{\partial J}{\partial x}u-\frac{\partial J}{\partial y}v]\frac{\partial J}{\partial y} = 0$$

Note that we also have $\frac{\partial I}{\partial t} = I(x,y)-J(x,y)$ (temporal gradient), and due to small motion we can approximate $\frac{\partial J}{\partial x}$ with $\frac{\partial I}{\partial x}$ (similarly for $\frac{\partial J}{\partial y}$ with $Iy$).

This can therefore be rewritten as;

$$\frac{\partial E}{\partial u} = -2\sum_x\sum_y[-I_t-I_xu-I_yv]I_x = 0$$

$$\frac{\partial E}{\partial v} = -2\sum_x\sum_y[-I_t - I_x u - I_y v]I_y = 0$$

This can also be written in the following matrix form;

$$-\sum_x\sum_y\begin{bmatrix}I_xI_t\\I_yI_t\end{bmatrix} - \sum_x\sum_y\begin{bmatrix}I_x^2 & I_xI_y\\I_xI_y & I_y^2\end{bmatrix}\begin{bmatrix}u\\v\end{bmatrix} = 0$$

By rearranging, we can solve for the motion as follows;

$$\begin{bmatrix}u\\v\end{bmatrix} = \left(\sum_x\sum_y\begin{bmatrix}I_x^2 & I_xI_y\\I_xI_y & I_y^2\end{bmatrix}\right)^{-1}\sum_x\sum_y\begin{bmatrix}I_xI_t\\I_yI_t\end{bmatrix}$$

Compared to Lucas-Kanade optic flow, we are summing over pixels within the **template image** (Lucas-Kanade tracker), whereas the former sums over a small neighbourhood. In the general case, where the motion is more complex, we can use a parametric model for this motion, for example (note that if we have the identity matrix for $\boldsymbol{M}$, we only have translation as before);

$$\begin{aligned}W(x, y; p) &= \begin{bmatrix}m_1 & m_2\\m_3 & m_4\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix} + \begin{bmatrix}t_x\\t_y\end{bmatrix}\\ \min_{u,v} E(u, v) &= \sum_x\sum_y(I(x, y) - J(W(x, y; p))))^2\end{aligned}$$

The Lucas-Kanade method is a classical method, which has assumptions that may not hold. For example, the brightness constancy assumption may not always hold, and it doesn't learn discriminative features for this template (since it only uses pixel intensities).

### Correlation Filter

In a correlation filter, we aim to maximise the correlation between template features (in the current time frame) and image features (in the next time frame).

$$(f \star h)[n] = \sum_{m=-\infty}^{\infty} f[m]h[n+m]$$

The correlation can be more robust to illumination changes, compared to the sum of squared differences.

In 2D images, we have a search window (in the next time frame) and a box with the same size as the template. The box is moved around in the search window, and we can calculate the correlation for each of these positions (taking the maximum). We can use more discriminative features, such as those learnt from CNNs. A recent approach uses Siamese (twin / two) networks to compare features. The networks takes in a template at $t$ (size $127 \times 127 \times 3$), and a search window at $t + 1$ (size $255 \times 255 \times 3$), which is passed through convolutional layers to obtain feature maps ($6 \times 6 \times 128$ and $22 \times 22 \times 128$ respectively), which goes into a correlation function giving a score map ($17 \times 17 \times 1$). Each pixel in the score map relates to a position of the search window.

These can be trained with a supervised learning approach (the convolutional layers can be imported from a backbone network). Paired images can be extracted from videos and annotated (we can annotate the centre of objects in successive frames, and draw a region where the score is 1 for high correlation, and $-1$ outside) - this defines a ground truth for the score map.

### Tracking by Detection

Note that object tracking can be thought of as performing object detection at each frame. In tracking, we have typically have an initial bounding box, and we only care about how a single object moves, without considering objects in the background.

Since we are only interested in the object in the initial bounding box, we can learn specific features. We can perform online learning (using information from the video as it's being streamed), compared to offline (which uses data from an existing training set). For object tracking, we are more interested in online learning (where we want to show results in real time).

At each sliding window, we perform two tasks (for object detection);

- **classification**

  Note that in detection, this is connected to $K$ classes (one for each class), whereas in object tracking we only perform binary classification.

- **localisation**

  The localisation branch remains connected to 4 variables, and this enables us to vary the size of the bounding box if needed.

Positive and negative samples are extracted from the first time frame (manually drawn positive samples). Note that **hard samples** are objects that look quite similar to the object we want to track. We can perform data augmentation to obtain more training data (translation, rotation, scaling, etc.). As we get more samples from more time frames, we can also use them for training. The majority of the network can be pre-trained with large datasets, only the last few layers need to be trained online. There are fewer parameters in the classification and localisation branches, which can be trained with fewer samples.

### Action Recognition

This is closely related to motion, but we want to classify videos into classes (similar to how images are classified in object detection). It is typically difficult to recognise actions from a single static image, whereas we can understand actions better with videos. The pipeline is quite similar to image classification.

We can extract features from both the image (static), as well as from motion (dynamic). The former could give information about the edges / objects in the image, whereas the latter could give us temporal features such as optic flow fields. These can be combined to train a classifier. Two-Stream 3D-ConvNet (2D-$t$, one temporal dimension) takes images in one 3D ConvNet, and optic flow in another, which is then combined to give an action class.

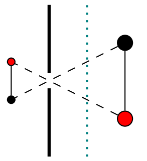
## Lecture 20 - Camera Model

The camera model describes the mapping from the 3D world to a 2D image. We denote the 3D coordinates as $\boldsymbol{X}$ and the 2D coordinates as $\boldsymbol{x}$. The mapping is described with a camera matrix, $\boldsymbol{P}$;

$$\boldsymbol{x} = \boldsymbol{P}\boldsymbol{x}$$

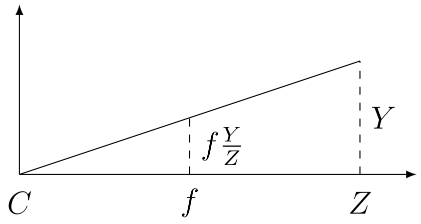
### Pinhole Camera

This is the simplest case, but can be generalised to other cameras. A pinhole is simply a very small hole (for example, in a board). Note we have a **dark room** on the left side of the board, and that the image is flipped;



We want to model how $\boldsymbol{X} = (X, Y, Z)^\top$ is mapped to $\boldsymbol{x} = (x, y)^\top$ on the image plane. The image plane is rotated by 180 degrees and put as a **virtual** image plane in front of the pin hole (see the teal line above). This is more convenient, as the image is no longer flipped.

The pinhole is treated as the camera origin, and the distance from the camera origin to the image plane is referred to as the focal length $f$. In one dimension, we can use **similar triangles**;



This therefore gives us the following mapping;

$$(X, Y, Z) \to \left(f\frac{X}{Z}, f\frac{Y}{Z}\right)$$

Homogeneous coordinates provide some convenience in geometry, as $(x, y, 1)$ represents the same point as $(kx, ky, k)$ for any non-zero $k$ (and similarly for 3D). In homogeneous coordinates, this becomes the following (see **CO317**);

$$(X, Y, Z, 1) \to \left(f\frac{X}{Z}, f\frac{Y}{Z}, 1\right) \text{ or } (fX, fY, Z)$$

This is a perspective projection, where the object will appear smaller as the distance increases. Pinhole cameras can be generalised to lens cameras, which also has similar triangles along the central ray. The mapping for a pinhole camera can be represented in the following matrix notation;

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \to \begin{bmatrix} fX \\ fY \\ Z \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

### Principal Point Offset

For example, in a CCD array, the image origin is typically defined at the bottom / top left, hence the image coordinate system's origin my differ from the **principal point** $\boldsymbol{p} = (p_x, p_y)$ (which would be in the centre of the image plane). To account for this shift, $(X, Y, Z, 1)$ should be mapped to $(f\frac{X}{Z} + p_x, f\frac{Y}{Z} + p_y, 1)$;

$$\begin{bmatrix} fX + p_xZ \\ fY + p_yZ \\ Z \end{bmatrix} = \begin{bmatrix} f & 0 & p_x & 0 \\ 0 & f & p_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

This can be written more concisely as follows;

$$\boldsymbol{K} = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix}$$

$$\boldsymbol{x} = \left[\boldsymbol{K} \mid \boldsymbol{0}\right] \boldsymbol{X}_{\text{cam}}$$

### World Coordinate System

Note that we're currently working in the camera's coordinate system (which is defined within the camera itself). However, the camera is actually moving in the **world coordinate system**. We can simply subtract the translation $\boldsymbol{C}$, if only translation is involved. However, if there is also rotation, we need to factor in a 3D rotation matrix $\boldsymbol{R}$;

$$\boldsymbol{X}_{\text{cam}} = \boldsymbol{R}(\boldsymbol{X} - \boldsymbol{C})$$

Note that we are using **inhomogeneous coordinates** (no '1' at the end), hence $\boldsymbol{X} = (X, Y, Z)^\top$. Using homogeneous coordinates, we can write the transformation as;

$$\underbrace{\begin{bmatrix} X_{\text{cam}} \\ Y_{\text{cam}} \\ Z_{\text{cam}} \\ 1 \end{bmatrix}}_{\boldsymbol{X}_{\text{cam}}} = \begin{bmatrix} \boldsymbol{R} & -\boldsymbol{RC} \\ 0 & 1 \end{bmatrix} \underbrace{\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}}_{\boldsymbol{X}}$$

### Camera Matrix

We first perform a mapping from the (homogenous) world coordinate ($\boldsymbol{X}$) to the camera coordinate ($\boldsymbol{X}_{\text{cam}}$), which is then mapped to the image coordinate ($\boldsymbol{x}$). Hence we have the following (note that the matrices in violet, when multiplied together, is the **pinhole camera matrix**);

$$\boldsymbol{x} = \begin{bmatrix} f & 0 & p_x & 0 \\ 0 & f & p_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \underbrace{\begin{bmatrix} \boldsymbol{R} & -\boldsymbol{RC} \\ 0 & 1 \end{bmatrix} \boldsymbol{X}}_{=\boldsymbol{X}_{\text{cam}}}$$

Note that the camera matrix can be written more concisely as the following (the column of 0s removes the bottom row of the world transformation);

$$\boldsymbol{P} = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{R} & -\boldsymbol{RC} \end{bmatrix} = \boldsymbol{K} \begin{bmatrix} \boldsymbol{R} \mid -\boldsymbol{RC} \end{bmatrix} = \boldsymbol{KR} \begin{bmatrix} \boldsymbol{I} \mid -\boldsymbol{C} \end{bmatrix}$$

We have a total of 9 degrees of freedom (3 intrinsic (internal to the camera) parameters, $(f, p_x, p_y)$ and 6 extrinsic (world to camera, outside of the camera) parameters to describe the 3 for 3D rotation and 3 for 3D translation).

Note that we're still representing the image coordinates in the same units as the camera coordinates (such as millimetres), however we want to have some conversion ratio $k_x, k_y$ (such as pixels per millimetre);

$$\left(f\frac{X}{Z} + p_x, f\frac{Y}{Z} + p_y, 1\right) \text{ converted to } \underbrace{\left(k_x\left(f\frac{X}{Z} + p_x\right), k_y\left(f\frac{Y}{Z} + p_y\right), 1\right)}_{(\alpha_x X + x_0, \alpha_y Y + y_0, 1)}$$

This converts the camera matrix to the following;

$$\boldsymbol{P} = \begin{bmatrix} \alpha_x & 0 & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{R} & -\boldsymbol{RC} \end{bmatrix}$$

However, $\alpha_x$ and $\alpha_y$ may differ, such that the pixels are **anisotropic** (such as rectangular pixels), leading to 10 degrees of freedom. This can be increased to 11 degrees of freedom when we account for skew (note that $s = 0$ for most cameras), when the axes are not orthogonal, giving us the most general camera matrix;

$$\boldsymbol{P} = \begin{bmatrix} \alpha_x & s & x_0 \\ 0 & \alpha_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{R} & -\boldsymbol{RC} \end{bmatrix}$$

### Calibration

Assume that we know the 3D structure, we want to estimate the camera matrix (photographer pose) from a 2D image. Given a set of points $\{\boldsymbol{X_i}, \boldsymbol{x_i}\}$, we want to estimate the matrix $\boldsymbol{P}$, which is similar to many problems previously encountered. We can either solve this as an (overdetermined) equation system, or to formulate this as an optimisation problem.

This mapping is formulated as;

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} p_1 & p_2 & p_3 & p_4 \\ p_5 & p_6 & p_7 & p_8 \\ p_9 & p_{10} & p_{11} & p_{12} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{p_1}^\top \\ \boldsymbol{p_2}^\top \\ \boldsymbol{p_3}^\top \end{bmatrix} \boldsymbol{X}$$

The second form allows us to have the following;

$$\begin{aligned} x &= \boldsymbol{p_1}^\top \boldsymbol{X} \\ y &= \boldsymbol{p_2}^\top \boldsymbol{X} \\ z &= \boldsymbol{p_3}^\top \boldsymbol{X} \end{aligned}$$

Converted into inhomogeneous image coordinates, we have;

$$\begin{aligned} x &= \frac{\boldsymbol{p_1}^\top \boldsymbol{X}}{\boldsymbol{p_3}^\top \boldsymbol{X}} \\ y &= \frac{\boldsymbol{p_2}^\top \boldsymbol{X}}{\boldsymbol{p_3}^\top \boldsymbol{X}} \end{aligned}$$

Rearranged, we obtain the following two equations;

$$\boldsymbol{p_1}^\top \boldsymbol{X} - \boldsymbol{p_3}^\top \boldsymbol{X} x = 0$$

$$\boldsymbol{p_2}^\top \boldsymbol{X} - \boldsymbol{p_3}^\top \boldsymbol{X} y = 0$$

By transposing both sides;

$$\begin{aligned} \boldsymbol{X}^\top \boldsymbol{p_1} - \boldsymbol{X}^\top \boldsymbol{p_3} x = 0 \\ \boldsymbol{X}^\top \boldsymbol{p_2} - \boldsymbol{X}^\top \boldsymbol{p_3} y = 0 \end{aligned}$$

This can now be written in matrix form, where $\boldsymbol{p_i}$ is a $4 \times 1$ vector;

$$\underbrace{\begin{bmatrix} \boldsymbol{X}^\top & 0 & -\boldsymbol{X}^\top x \\ 0 & \boldsymbol{X}^\top & -\boldsymbol{X}^\top y \end{bmatrix}}_{2\times 12} \underbrace{\begin{bmatrix} \boldsymbol{p_1} \\ \boldsymbol{p_2} \\ \boldsymbol{p_3} \end{bmatrix}}_{12\times 1} = \boldsymbol{0}$$

For $n$ pairs of coordinates, the overdetermined system can be formed as follows;

$$\underbrace{\begin{bmatrix} \boldsymbol{X_1}^\top & 0 & -\boldsymbol{X_1}^\top x_1 \\ 0 & \boldsymbol{X_1}^\top & -\boldsymbol{X_1}^\top y_1 \\ \vdots & \vdots & \vdots \\ \boldsymbol{X_n}^\top & 0 & -\boldsymbol{X_n}^\top x_n \\ 0 & \boldsymbol{X_n}^\top & -\boldsymbol{X_n}^\top y_n \end{bmatrix}}_{2n\times 12} \underbrace{\begin{bmatrix} \boldsymbol{p_1} \\ \boldsymbol{p_2} \\ \boldsymbol{p_3} \end{bmatrix}}_{12\times 1} = \boldsymbol{0}$$

Due to this form, the solution $\boldsymbol{p}$ is the null space of $\boldsymbol{A}$ (the large matrix) - this can be solved with the following optimisation problem, note that we add a constraint $||\boldsymbol{p}||^2 = 1$ as we want to exclude trivial solutions (where $\boldsymbol{p} = \boldsymbol{0}$);

$$\boldsymbol{p} = \underset{\boldsymbol{p}}{\operatorname{argmin}} \, ||\boldsymbol{A}\boldsymbol{p}||^2$$

This can be solved by performing singular value decomposition for the matrix $\boldsymbol{A}$ (see **CO233**), where $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthogonal, and $\boldsymbol{\Sigma}$ is diagonal. $\boldsymbol{p}$ is the column of $\boldsymbol{V}$ corresponding to the **smallest singular value**;

$$\underbrace{\boldsymbol{A}}_{m\times n} = \underbrace{\boldsymbol{U}}_{m\times m}\underbrace{\boldsymbol{\Sigma}}_{m\times n}\underbrace{\boldsymbol{V}^\top}_{n\times n}$$

Once we have this solution, w can rearrange the elements to form the camera matrix, establishing the mapping from 3D to 2D.

We can obtain these data points (pairs of camera and image coordinates) with references objects (such as a checkerboard), which we can use to identify correspondences between points in the image and scene via a manual process.

This can be used to relate information from the 3D world to 2D images, which is utilised in AR or object tracking for drones.

### Multiple View Geometry

We only consider a single perspective. However, if the camera is moved, multiple view geometry is used (where the image coordinates differ based on the perspective of the camera).