Computer Robot Vision: Lecture 4

Sam Barrett

February 22, 2021

1 Image Texture

Texture is a very important factor in understanding properties of an object however they can be difficult to define due to potential near random patterns.

We can use texture to:

- estimate surface orientation or shape.
- classify/ segment an image. Allowing us to group image regions with consistent texture.
- synthesise new texture patches or entire images given some example set.

By analysing texture we can often identify properties of the material it is made from. Texture acts as a *feature* that is one step abstracted from the basics of filters and edges.

We can think of textures as **repeated local patterns**. This reduces the first stage of our task of analysing texture to finding patterns. Patterns can include: spots, bars, raw patches etc. We describe textures with statistics related to their local window including mean, standard deviation etc.

We can use our x and y derivative filters to find the texture *responses* in each section of an image. We can then find local statistics by moving a arbitrarily large *window* across the image.

Having calculated this information we can identify all windows in the image with high vertical or horizontal (or both) activity, these areas are likely to have *texture* following these directions. Colour-coding these 4 sets (no intensity, high x, high y, high x, y) we can roughly visualise textures present in the image.

On a plot of mean $\frac{\delta}{\delta x}$ against mean $\frac{\delta}{\delta y}$ the further apart 2 windows are, the more dissimilar their textures. We measure the distance between two points on this plot simply as Euclidean distance ($D(a,b) = \sqrt{\sum_{i=1}^{2}(a_i - b_i)^2}$)

When performing this form of analysis, the window size we use is very important. If our window is close ins size to an object of interest, made up of multiple textures, we will lose a lot of information.

So far, we have discussed using 1st order derivative filters to find texture information. We can more generally say that we aapply a collection of multiple,

d, filters called a *filter bank*. This results in feature vectors in d dimensions. We can still consider *nearness* as the Euclidean distance but now we consider it in d dimensional space. This is a simple extension to the previous equation:

$$D(a,b) = \sqrt{\sum_{i=1}^{d}(a_i - b_i)^2}$$

1.1 Filter Banks

We design filter banks so that they cover most bases of possible textures. They contain filters specific to edges, bars, spots or any other textural feature that we are interested in.

Many of these different filters are **multivariate Gaussian** filters. Multivariate Gaussian filters can be stretched and contorted to respond to different shapes through **manual** tuning of its parameters. Multivariate Gaussian filters have the form:

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

By changing the parameter Σ we can change the direction and distribution of the filter.

More recent work has been done to *learn* filter banks using Convolutional Neural Networks. Here highly specialised filters are learnt from a training set in order to closely match the domain in which the model will be deployed.

CNNs use different levels of filter banks, low level features include basic edges and textures, mid-level filters detect more precise but still abstract features and high-level filters are more concrete still, detecting exact patterns or even objects.

1.2 Texture Synthesis

There are many applications for being able create new samples of a given texture:

- Virtual environments (games, VR, etc.)
- Hole-filling on images
- Texturing surfaces

The challenge with this sort of task the spectrum of textures we are trying to replicate, ideally with a single general solution. We want to model both repeated and stochastic (random) textures.

1.2.1 Markvov Chains

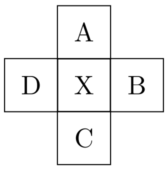
A Markov chain is a sequence of random variables $x_1, x_2, \ldots, x_n$ where x_t is the state of the model at a given time t.

They have been used for text synthesis, with the most basic implementations building a probability histogram or table featuring the probabilities of a word being present given the previous words seen $(p(x_t|t_{t-1}, \ldots, x_{t-(n-1)}))$

More modern approaches use machine learning models such Long-short-term-memory networks or LSTMs. The results of these models are better but still often nonsensical.

We can extend the approach of Markov chains to help us to predict and synthesise textures in images. We do this using **Markov random fields** (MRF), these are generalisations of Markov chains to be in 2 or more dimensions.

A first order MRF (works on 2 dimensions) computes the probability that pixel X takes a certain value **given** the values of neighbours A, B, C, D, $P(X|A, B, C, D)$



1.2.2 Single pixel synthesis

In order to synthesise a value for a single pixel x we need to calculate $P(x|\text{neighbourhood of pixels around } x)$.

To do this we:

- first find all windows **elsewhere in the image** that match the neighbourhood of x.
- Pick 1 matching window at random, W
- assign x the centre pixel of W.

Often an exact neighbourhood match will not exist, so we find the best matches using sum of squared differences (SSD) error and choose between them with a weighted probability, preferring *better* matches.

The size of our neighbourhood window effects the predictions we make. It must be set according to the texture being synthesised. Brickwork, for example, should not be synthesised using a window bigger than a single brick etc.

1.2.3 Efros & Leung algorithm

This is the algorithm for synthesising texture pixel-by-pixel, it is **simple**, yields **good results** but is **very slow**.

1.2.4 Efros & Freeman algorithm

This is an improved version of the Efros and Leung algorithm, it is based on the observation that neighbourhood pixels are highly correlated. The main idea is to use a *block* as the base unit of synthesis rather than a single pixel.

This changes our conditional probability calculation to $P(\mathbf{B}|N(\mathbf{B}))$, this is much faster as we now synthesise all pixels in $\mathbf{B}$ at once.

The layout of blocks used greatly effects performance of this algorithm.

- Placing blocks randomly that do not overlap results in sharp, unnatural edges in the synthesised texture
- Making neighbouring blocks overlap produces better but still unnatural results.
- The minimal error boundary cut approach produces the best result

  This approach is done by overlapping 2 blocks. The edge between them is clear so we take the two overlapping sections of each block and calculate the overlap error. This error gives us the *minimal error boundary*, cutting each block along this boundary allows us to accurately stitch these two blocks together.

  This approach can be seen in Figure 1

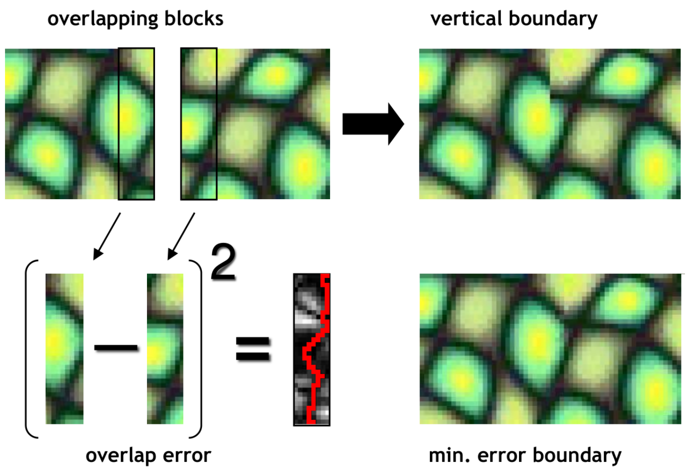


Figure 1: Minimal error boundary calculation

1.3 Texture Transfer

Texture transfer is the process of mapping a texture from one object to another. This requires being able to separate texture from the shape of an object.

This is very difficult to do, so we cheat. We assume we can capture shape through boundaries and rough shading, then we just add similarity to underlying image at the same point as a constraint when sampling.

2 Image Segmentation

2.1 Grouping in vision

The goal of grouping in computer vision is to:

- Collect features that belong together

  This can include determining image regions by colour or texture, or grouping video frames into *shots* to help us extact people or objects from a video.

- Obtain an intermediate representation that compactly describes key image or video components

There are two main approaches to grouping: **Top-down** and **bottom-up** segmentation.

Top down segmentation means that pixels belong together as they are from the same object. This approach requires prior knowledge about objects in the image, we then infer that similar pixels are together.

Bottom up segmentation means that pixels belong together as they *look* similar.

2.2 Bottom up segmentation

Algorithms include k-means clustering and mean-shift clustering. There are also graphical approaches.

The goals of segmentation are to:

- separate images into coherent *objects*.
- group together similar looking pixels.

  This is for efficiency of later processing. These blocks of similar pixels are known as *super-pixels* and are very useful for higher level processing.

2.2.1 k-means clustering

This algorithm will **always converge** to some solution. However, this can be a local minimum. I.e. it does nto always find the global minimum of the objective function $\mathcal{F} = \sum_{i=0}^{k} \sum_{j=0}^{n} \|p_j - c_i\|^2$.

It is also very fast and simple to compute. However, setting k is task specific (or inefficient to determine), the process is sensitive to the initial cluster centres as well as to outliers.

We also assume that the means of the our points can be computed.

```
Randomly initialise the cluster centres $c_1, \dots, c_k$
while cluster assignments change do
    foreach each point $p$ do
        find the closest centre $c_i$, place $p$ into cluster $i$
    end
    foreach cluster, $c_i$ do
        Calculate new centroid (mean point in cluster)
    end
end
```

Algorithm 1: k-means clusering

2.2.2 Segmentation as clustering

We can group pixels in different ways based on what we select to be our feature space.

Our feature space can be many different things including (but not limited to):

- Intensity
- Colour similarity (3D feature space)
- Intensity and position
- Texture similarity
- n-dimensional feature space

2.2.3 Mean shift algorithm

The mean shift algorithm looks for *modes* or local maxima of density in the feature space.

```
Initialise random seed and window $W$
while not converged do
    calculate the centre of gravity (mean) of $W$
    shift $W$ to the mean
end
```

Algorithm 2: Mean shift algorithm

This can be applied to clustering/ segmentation by:

1. Finding features based on some feature space (colour, gradients, texture, etc.)
2. Initialising windows at each individual feature point
3. Performing mean-shift for each window to convergence

4. Merge windows that end up near the same *peak* or mode

Advantages of mean-shift

- Does not assume shape of clusters (k-means assumes spherical clusters)
- Single parameter to tune (window size)
- Generic technique
- Can find multiple modal values

Disadvantages of mean-shift:

- Manual selection of window size
- Does not scale well with dimensionality of feature space (texture etc.)

2.2.4 Graphical approaches

We can also view images as fully-connected graphs. Each node is a pixel and there is a edge between every pair of pixels.

We can assign an *affinity weight* to each edge $w_{p,q}$ measures similarity.

One definition for affinity is:

$$\mathit{aff}(x, y) = \exp\left\{-\left(\frac{1}{2\sigma_d^2}\right)(\|x - y\|^2)\right\}$$

Where small values of σ will group only nearby points and large values of σ will group distant points.

With a image graph and this notion of affinity, we can break graph into segments. We want to remove edges that cross between segments. The easiest method breaks the links that have low similarity. This means that similar pixels should be in the same segments.

MinCut We can construct a notion of **MinCut**, the set of links in a graph G with two subgraphs representing segments A, B whose removal makes G disconnected. The cost of a cut is given as:

$$\mathit{cut}(A, B) = \sum_{p \in A, q \in B} w_{p,q}$$

Finding **MinCut** gives us a segmentation of the image represented by G, there exist efficient algorithms to find this.

The issue with this is that the weight of a cut is proportional to the number of edges in the cut. This means it tends to produce small, isolated components.

Normalised cut Normalised cut is a MinCut alternative. Here we fix the bias of **MinCut** by normalising the size of the segments.

$$Ncut(A,B) = \frac{cut(A,B)}{assoc(A,V)} + \frac{cut(A,B)}{assoc(B,V)}$$

Where $assoc(A,V)$ is the sum of weights of all edges that touch A.

The value of *Ncut* is small when we get two clusters with many edges with high weights and few edges of low weight between them. The a solution to $\min(Ncut)$ is approximately the same as the generalised eigenvalue problem.

Advantages of normalised cuts

- Generic framework, flexible choice of definition for *affinity*, which assigns weight to edges.
- Does not require model of data distribution

Disadvantages:

- Can have high time complexity, dense graphs require many affinity calculations. Also solving the eigenvalue problem is non-trivial
- It has a preference for balanced partitions.

There is an efficient graph-based segmentation approach, it runs in time linear in the number of edges, it is easy to control coarseness of segmentation however, the results can be unstable.

3 Fitting

In fitting we aim to select a parametric model that best represents a set of features.

We know that membership of a model cannot be determined on a point-local basis, we must consider surrounding pixel values also.

We can reduce fitting to three main questions:

1. What model represents this set of features best?
2. Which of several model instances get which feature?
3. How many model instances are there?

When considering fitting we must be mindful of time complexity, it is not computationally feasible to examine every possible set of parameters and every possible combination of features.

3.1 Line fitting

The simplest type of feature we often wish to fit are straight lines. Many objects are characterised by the presence of straight lines.

But why can this not be achieved simply through edge detection?

There are often multiple extra edge points in an image which can lead to clutter in our models as we ask the question *which points go with which lines, if any?*

Sometimes only sections of a line are detected, how do we find lines that span sections of an image with imperfect information?

There is often noise in measured edge points and orientations, how do we remove noise to detect the true underlying parameters?

3.2 Voting

We have said that it is not feasible to check all combinations of features by fitting a model to each possible subset. **Voting** is a general technique whereby we let each feature *vote* for all models that are compatible with it.

It works by cycling through features and casting votes for model parameters, we then look for the model parameters that receive the most votes.

Features created due to noise and clutter will still vote in this system but generally their votes should be outweighed by the larger set of *"good"* features.

3.3 Hough Transforms

Hough Transforms are a voting technique used to determine the following information:

- Given some point that belong to *a* line, what is the line?
- How many lines are there in a given image?
- Which points belong to which line?

The main idea behind this technique is:

- each straight line in an image can be described by a (unique) equation
- Each point on this line, when considered in isolation, could lie on an infinite number of other straight lines
- In the Hough Transform, each point votes for every line it could be on; the lines with the most votes *win*

How do we represent lines? Any line can be represented using 2 numbers, w and ϕ. We say a line is a line from an agreed origin of length w at an angle ϕ to the horizontal.

Since we can use (w, ϕ) to represent any line in the image space, we can represent any line in the image space as a point in the plane defined by (w, ϕ), we call this plane the **Hough Space**.

How does a point in the image space vote? We can transform a point in the image space to a sinusoidal curve in the Hough Space through the equation:

$$w = x\cos(\phi) + y\sin(\phi)$$

Therefore, two points in the image space correspond to two curves in the Hough space. The intersection of those two curves has two *votes* . This intersection represents the straight line in the image space that passes through both points.

The basic Hough Transform algorithm is:

```
Create an array A indexed by $\phi$ and $w$
**foreach** *point* $(x, y)$ *in the image space* **do**
    **foreach** *angle* $\phi \in [\phi_{min}, \phi_{max}]$ **do**
        $w = x\cos(\phi) + y\sin(\phi)$
        $\mathtt{A}[\phi, w] = \mathtt{A}[\phi, w] + 1$
    **end**
**end**
Find the value(s) of $w, \phi$ where $\mathtt{A}[w, \phi]$ is maximum
The detected line in the image is given by $w = x\cos(\phi) + y\sin(\phi)$
```

Algorithm 3: Hough Transform

When we have a noisy image however, we still see votes being cast in the Hough space. This can lead to false positives.

We can extend Hough transform to include the gradient direction, this is a piece of information that we calculate during the edge detection phase.

We can alter our algorithm to form:

```
Create an array A indexed by $\phi$ and $w$
**foreach** *point* $(x, y)$ *in the image space* **do**
    $\phi$ = gradient at $(x, y)$
    $w = x\cos(\phi) + y\sin(\phi)$
    $\mathtt{A}[\phi, w] = \mathtt{A}[\phi, w] + 1$
**end**
Find the value(s) of $w, \phi$ where $\mathtt{A}[w, \phi]$ is maximum
The detected line in the image is given by $w = x\cos(\phi) + y\sin(\phi)$
```

Algorithm 4: Hough Transform

This reduces the degrees of freedom.

Other extensions that we can make are:

- Give more votes to stronger edges by incorporating the magnitude of the gradient
- Change the sampling of w, ϕ to increase or reduce the resolution
- We can extend this procedure for use on circle, square or any other regular shape.

3.3.1 Hough Transform for circles

Given a circle of centre (a, b) and radius r we define it parametrically as:

$$(x_i - a)^2 + (y_i - b)^2 = r^2$$

Known r For a known radius r the parameter space is reduced to 2D, each point represents the centre of a circle. For each point (x, y) on the original circle, there is a corresponding circle centred at (x, y) with a radius r in the Hough space. The intersection point of all such circles in the Hough space would be corresponding to the centre point of the original circle. See Figure 2

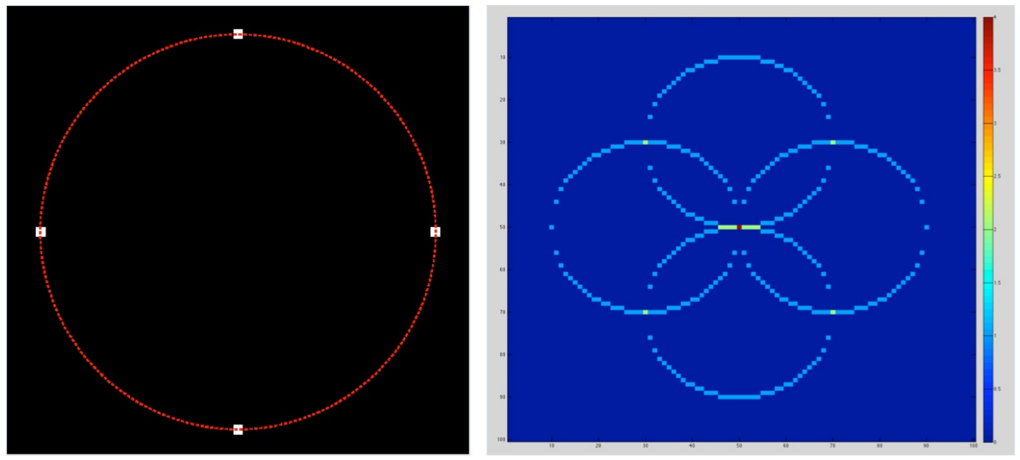


Figure 2: Circle Hough Transform

Unknown r In the case where the radius, as well as the centre, is unknown, the parameter space (Hough space) is 3 dimensional. Each point in the image space corresponds to a cone in the Hough space. The point of the cone is at $r = 0, x, y$. We can still find the set of parameters that gain the most votes in the Hough space by finding intersections between these cones. See Figure 3

Unknown radius, known gradient direction This is a subset of the previous case. By knowing the gradient direction, we make our calculation much simpler. Every point in our image space now becomes a single straight line in the Hough space. Votes are cast by finding intersections between these lines. See Figure 4

In this case, we can re-arrange our circle equation into the form:

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

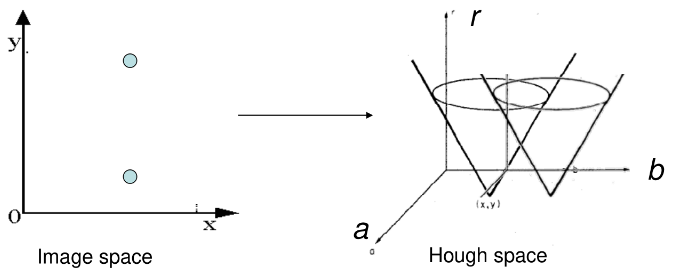


Figure 3: Circle Hough Transform for an unknown r

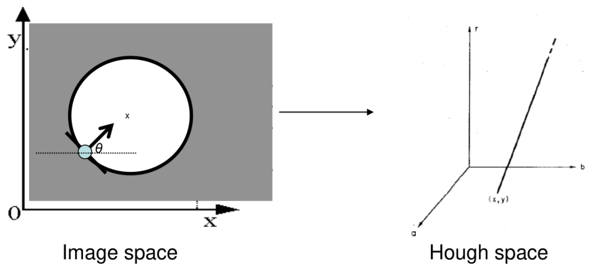


Figure 4: Known gradient direction Hough Transform

Here, every point (x, y) in the image space will be equivalent to a circle in the (a, b) Hough space. By rearranging these equations we get:

$$\begin{aligned} a &= x_i - r\cos\theta \\ b &= y_i - r\sin\theta \end{aligned}$$

If the radius is know then the centre of the circle can be calculated.
UNFINISHED