TSM_CompVis – Machine Learning in Computer Vision

Pascal Baumann, Selvin Blöchlinger

1. Februar 2024

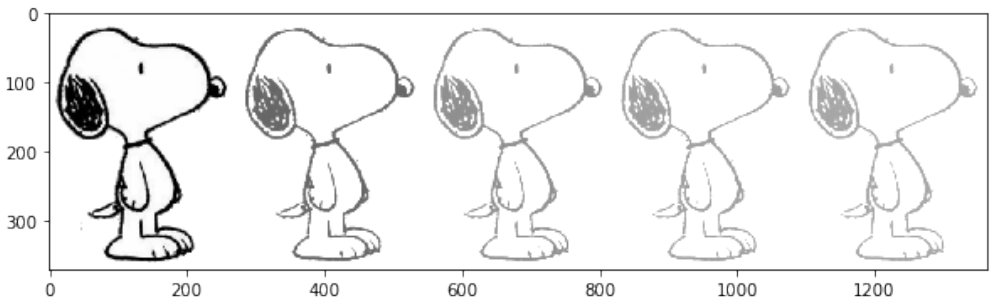
Contents

1 Image Handling in Python

```
plt.imshow(np.hstack((im_float, im_float)), cmap="gray", vmin=0, vmax=1);
```

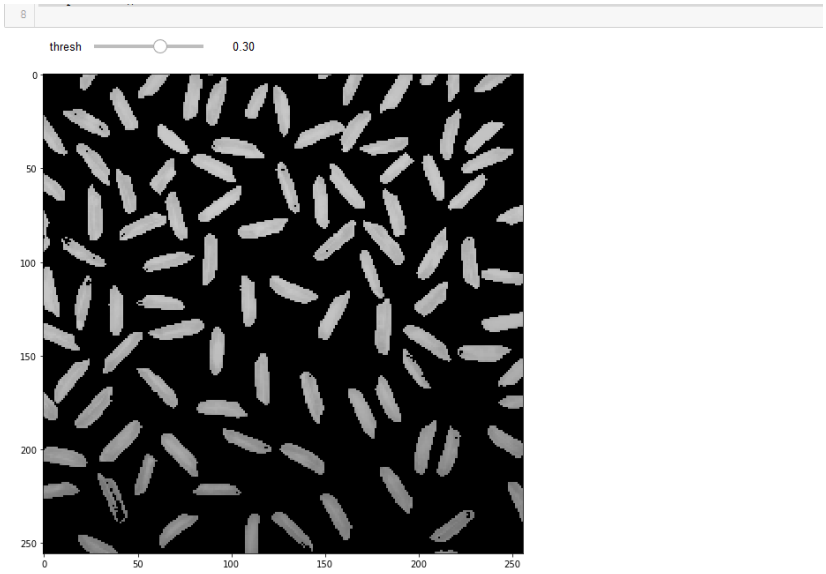
1.1 Fading an Image

```
im = skimage.io.imread("data/snoopy.png")
im = im / 255  # Tip: Use floating point values

ims = []

im_temp = np.fliplr(im)
ims.append(im_temp)
for i in range(1, 5):
    im_temp = im_temp + (0.4) ** i
    ims.append(im_temp)

plt.imshow(np.hstack(ims), vmin=0, vmax=1, cmap="gray")
```



1.1.1 IPyWidgets

```
@interact(thresh=widgets.FloatSlider(min=0.0, max=0.5, step=0.01, value=0.2))
def threshold(thresh):
    im_thresh = []
    for line in im:
        im_thresh.append([pixel if pixel > np.min(line) + thresh else float(0) for pixel in line])
    plt.imshow(im_thresh, cmap="gray", vmin=0, vmax=1)
    plt.show()
```



2 Local Filtering and Edge Detection

2.1 Filtering

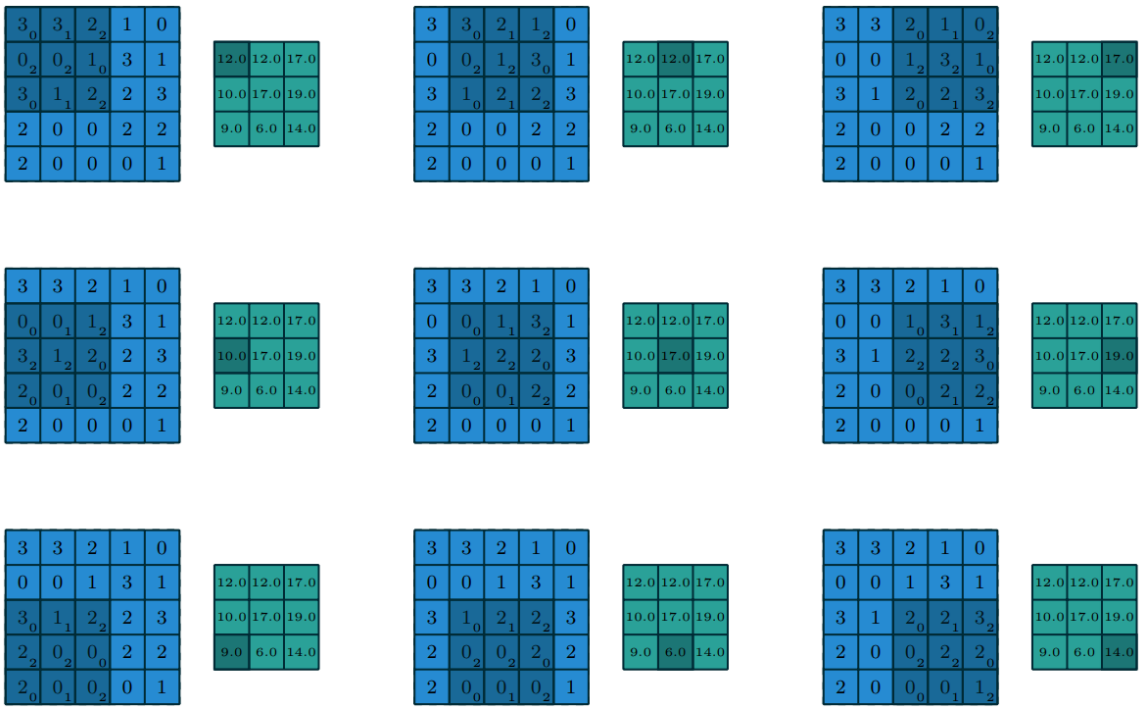
The naive approach to local filtering involves taking a simple moving average of the pixels in the neighborhood.

2.2 Convolution

Convolution entails convolving an input matrix (the input image) with a **kernel**, a matrix defining weights for each element in the neighborhood. This operation can be seen as a moving weighted average of pixels.

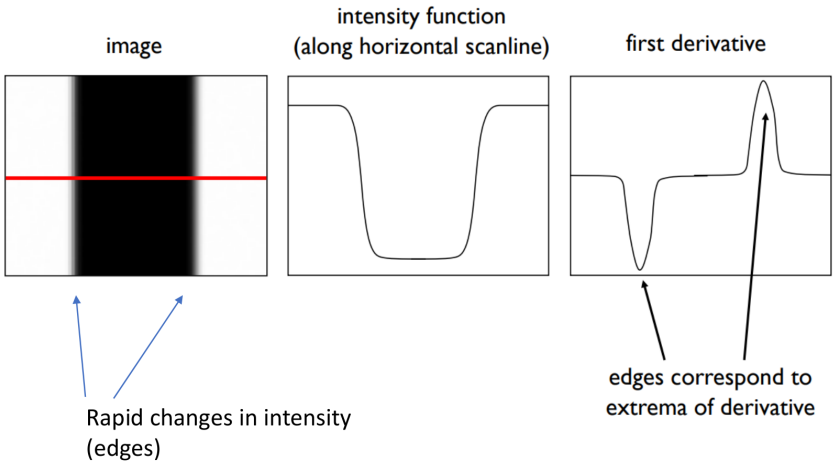
0	1	2
2	2	0
0	1	2

The kernel slides across the input matrix, computing the product between each kernel element and the corresponding input element. The results are summed to obtain the output at the current location. If the input matrix has i rows and the kernel has k rows, the output will have $i - k + 1$ rows (and columns).



2.3 Edge Detection

Edge detection, challenging in image processing, identifies edges based on rapid intensity changes, which can be captured by calculating the derivative.



In two dimensions, the derivative corresponds to the gradient $\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]$, pointing from the edge towards the intensity increase or the lighter side.

$$\text{Gradient} \qquad \nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right] \tag{1}$$

$$\text{Gradient Direction} \qquad \theta = \tan^{-1}\left(\frac{\partial f}{\partial x} / \frac{\partial f}{\partial y}\right) \tag{2}$$

$$\text{Gradient Magnitude (or Modulus)} \qquad \|\nabla f\| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2} \tag{3}$$

However, not all significant edges have strong gradients, nor are all strong gradients indicative of important edges.

2.3.1 Computing the Gradient on an Image

The approximate gradient in the $\frac{\partial f}{\partial x}$ direction is obtained by convolution with the kernel

-1	1

The approximate gradient in the $\frac{\partial f}{\partial y}$ direction is obtained by convolution with the kernel $\begin{array}{|c|}\hline -1 \\ \hline 1 \\ \hline \end{array}$

The drawback of the gradient is its sensitivity to noise, which can be addressed by first smoothing the function and then applying the gradient.

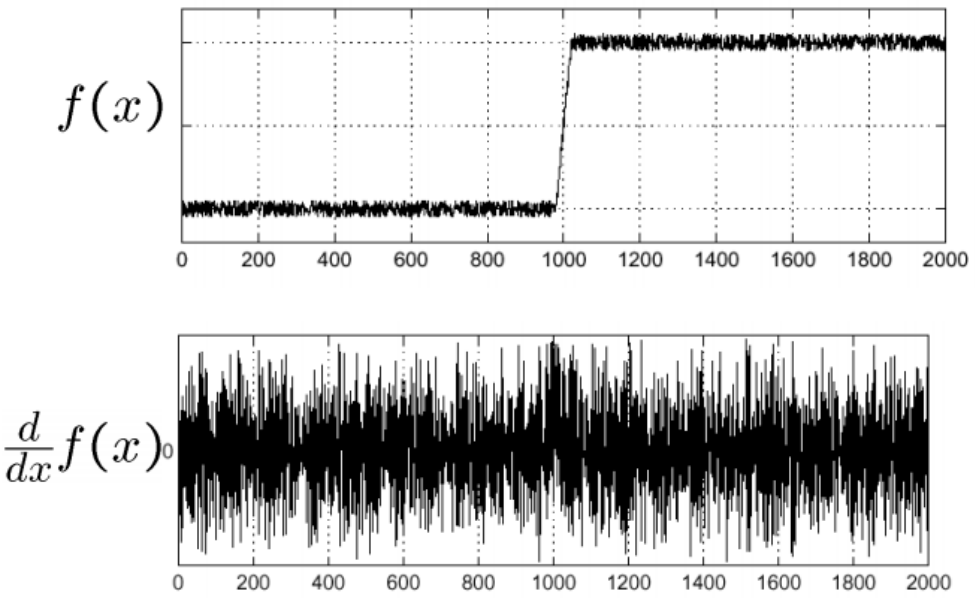


Figure 1: Gradient of a 1D function, illustrating noise amplification and potential signal concealment.

The solution is to smooth the function first, and then apply the gradient. A convolution with a derivative of the Gaussian filter is often used for this purpose, equivalent to smoothing with a Gaussian and then taking the derivative.

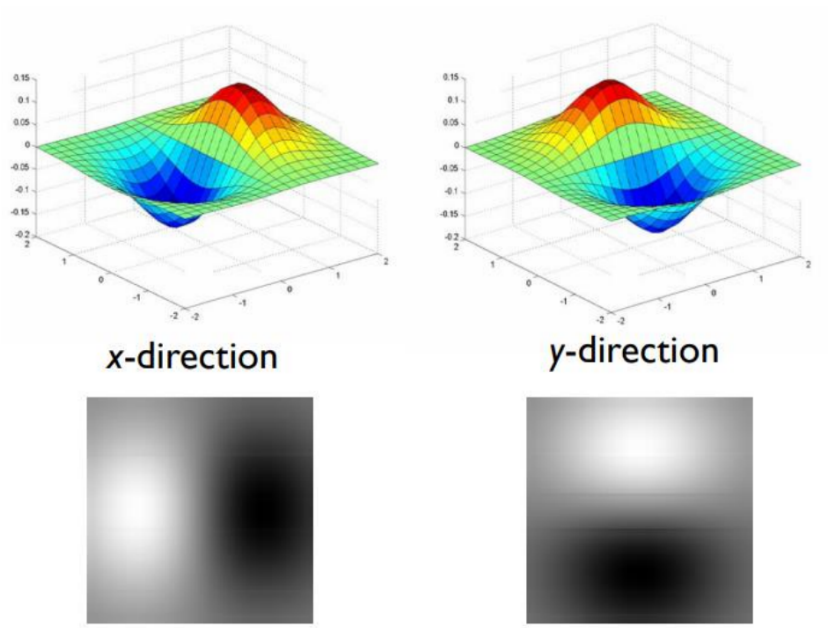


Figure 2: Derivative of Gaussian Filters. A Gaussian smoothing filter removes high-frequency components, and derivative filters yield large responses at points with high contrast.

2.4 Canny Edge Detection

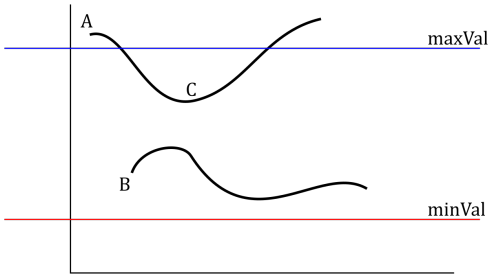
Canny Edge Detection follows a similar approach but includes edge thinning and hysteresis thresholding for enhanced performance. The steps include:

1. Approximating gradients along axes using derivative of Gaussian filters.

2. Computing gradient magnitude.
3. Making edges one pixel wide through non-maxima suppression along the perpendicular direction to the edge.
4. Keeping only strong edges through hysteresis thresholding.

2.5 Hysteresis Thresholding

This stage determines genuine edges by employing two threshold values, `minVal` and `maxVal`. Edges with an intensity gradient surpassing `maxVal` (A) are definitively identified as edges, while those falling below `minVal` are discarded as non-edges. Pixels lying between these thresholds are classified based on connectivity: if connected to "sure-edge" pixels (C), they are deemed part of edges; otherwise (B), they are also discarded.



3 Model Fitting and Line and Circle Detection

Edge detection alone is often insufficient, as images may contain multiple lines and numerous edges unrelated to lines. Noise in line edges can cause misalignment, and lines may not be complete.

3.1 Model Fitting

Model fitting is an algorithm that identifies a high-level model to explain observations effectively. Here, edges serve as observations, and the model consists of one or more lines.

3.2 Voting Algorithms

Voting algorithms are a general technique for decision methods.

1. Every feature casts votes for all compatible models.
2. Models accumulating many votes are chosen.

Clutter and noise may cast many votes but lack consistency. Features belonging to a model concentrate votes for that model.

3.3 Hough Transform

1. Every **edge point** casts votes for **all compatible lines**.
2. **Lines** accumulating many votes are chosen.

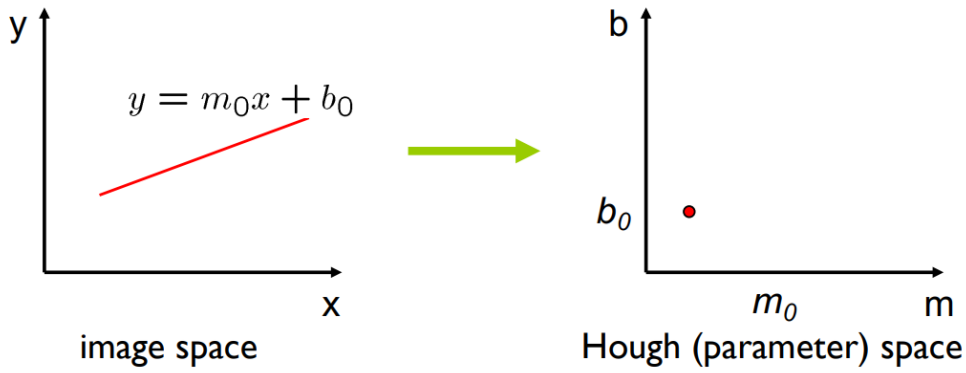


Figure 3: Line representation in Hough space

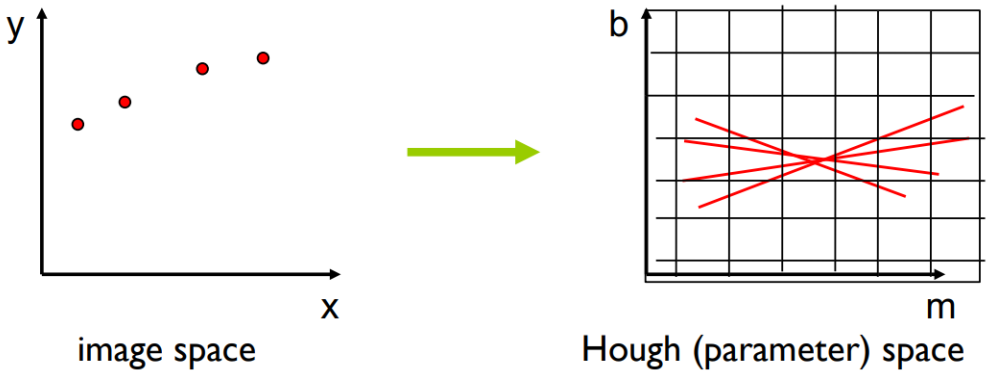


Figure 4: Points on a line in normal space almost intersect in Hough space

The problem with this approach is that

a. the parameter space (m, b) is **not bounded**

b. **vertical lines** cannot be represented

3.4 Representation of Lines in Polar Coordinates

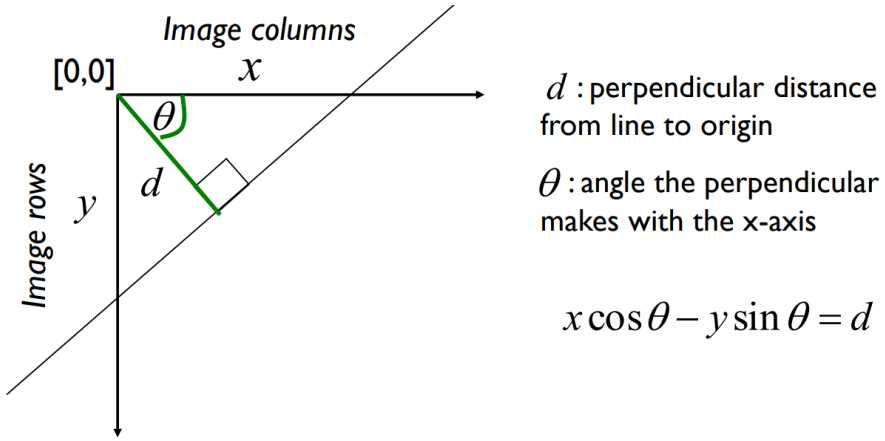


Figure 5: Line representation with polar parameters

This approach solves both the problems of unboundedness and representation. The parameters θ and d have bounded ranges.

For every point in normal space, there will be a sinusoid in the polar Hough space.

3.5 Hough Transformation Algorithm

The algorithm transforms each edge point in the image to Hough space, where an Accumulator array gets filled with votes. The bin size of the accumulator is an important hyperparameter. If too small, there will be many weak peaks due to noise; if too large, accuracy of locating a line drops, and clutter votes might end up in the same bin. A solution is to keep the bin size small but also count votes from neighbors.

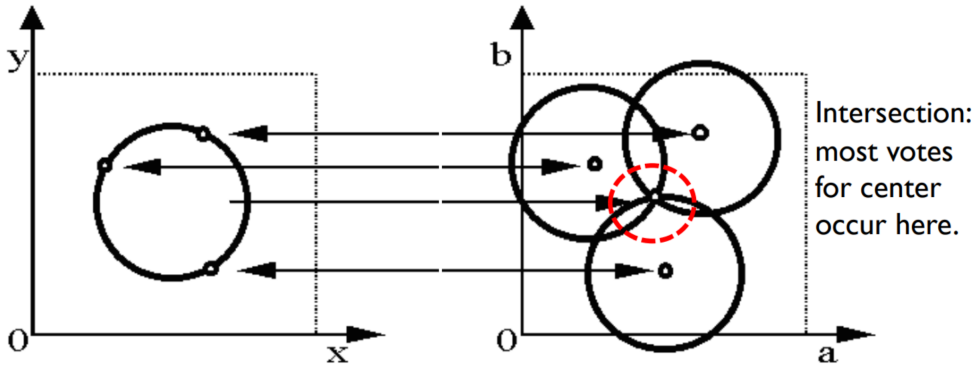
3.5.1 Voting Algorithm for Finding Lines

1. Every edge point casts votes for all circles that are compatible with it.
2. Circles accumulating many votes are chosen.

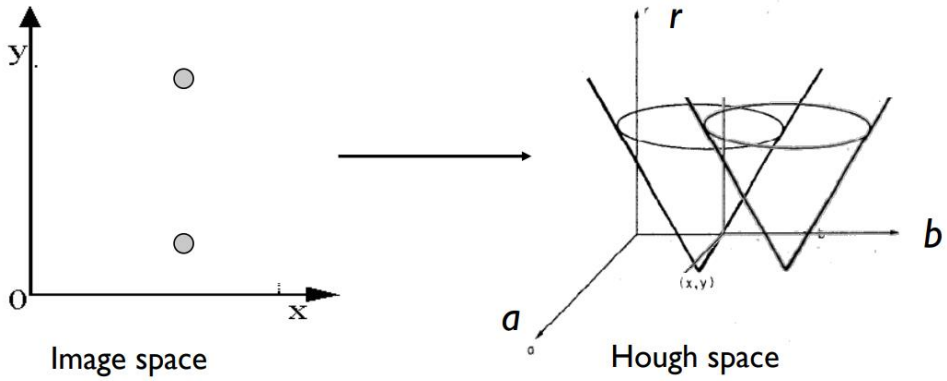
Theorem 1. *Parametrization of a circle:*

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

With center (a, b) and radius r.



3.6 Hough Space for Circles With Unknown Radius



$$y = a \cdot x + b$$

4 Semantic Segmentation

Semantic segmentation involves identifying pixels in images that belong to a specific class, sometimes identifying pixels to belong to a specific instance of a class. It can be seen as supervised learning at the pixel level.

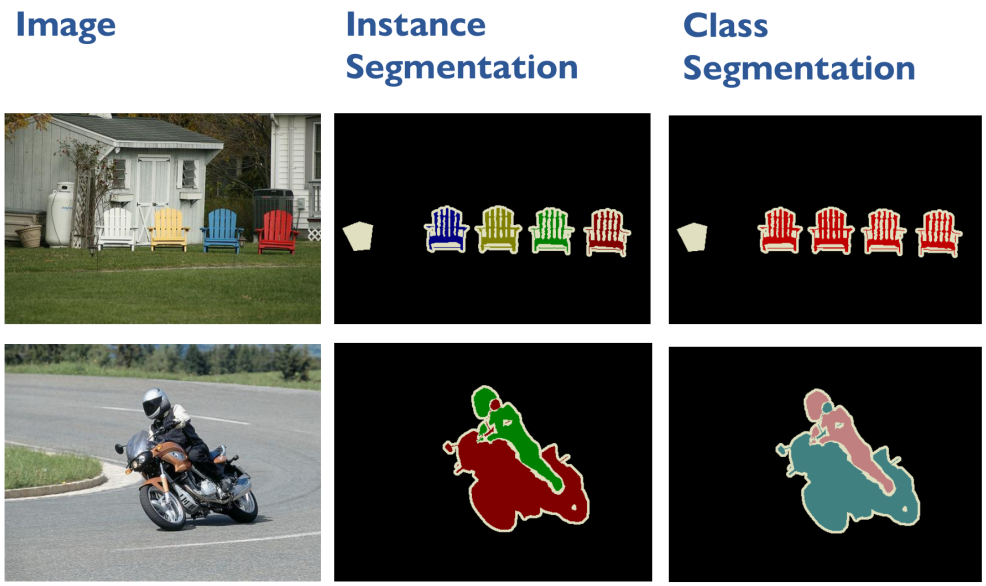


Figure 6: Semantic Segmentation Example

Non-semantic segmentation aims to find regions in images without additional information like classes. It is similar to unsupervised learning and often follows a bottom-up approach. Segmentation, in general, seeks a compact representation of a scene in terms of regions that share common properties.

4.1 k-Means Clustering

K-Means clustering is not an optimal solution for image clustering. Lloyd's algorithm is one possibility for clustering with the following steps:

- Initialize cluster centers (e.g., randomly).
- Assign samples to clusters.
- Calculate mean values of clusters as new centers.
- Iterate until clusters don't change.

Issues with K-Means include the requirement to specify the number of clusters and the assumption of spherical cluster shapes.

4.2 Mean Shift Clustering

Mean Shift clustering allows variable cluster shapes, avoiding the need for predefined shapes. The algorithm involves:

1. Search for the maximum of the feature density from a given position in the image.
2. Define a cluster as the set of positions that converge to the same maximum.

The algorithm deals with the challenge of searching for local maxima by using Kernel Density Estimation with a Gaussian kernel or the Derivative of the Gaussian for direct approximation.

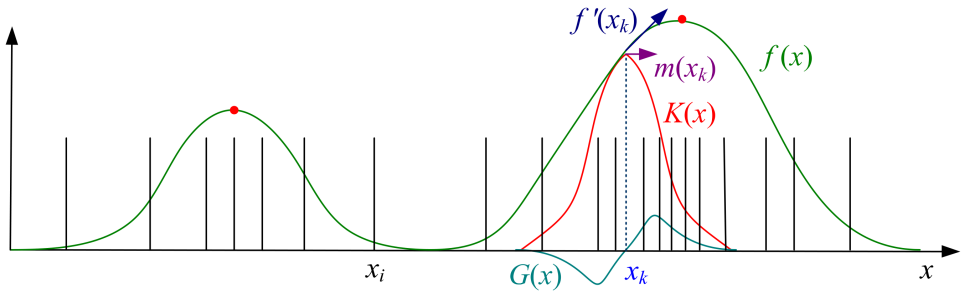


Figure 7: Kernel Density Estimation

4.2.1 Mean Shift Algorithm

For all positions x in the image, calculate the mean $\overline{x}$ in the environment $N(x)$:

$$\overline{x} = \frac{\sum_{x_i \in N(x)} K(x_i - x) x_i}{\sum_{x_i \in N(x)} K(x_i - x)}$$

Set x to the mean $\overline{x}$ and iterate until x does not change. Attraction basins are regions from which all trajectories lead to the same node.

4.3 Segmentation by Graph Cuts

Images are represented as graphs with nodes for pixels and edges between nodes with affinity weights. These weights measure similarity, such as inversely proportional to the difference in color and position.

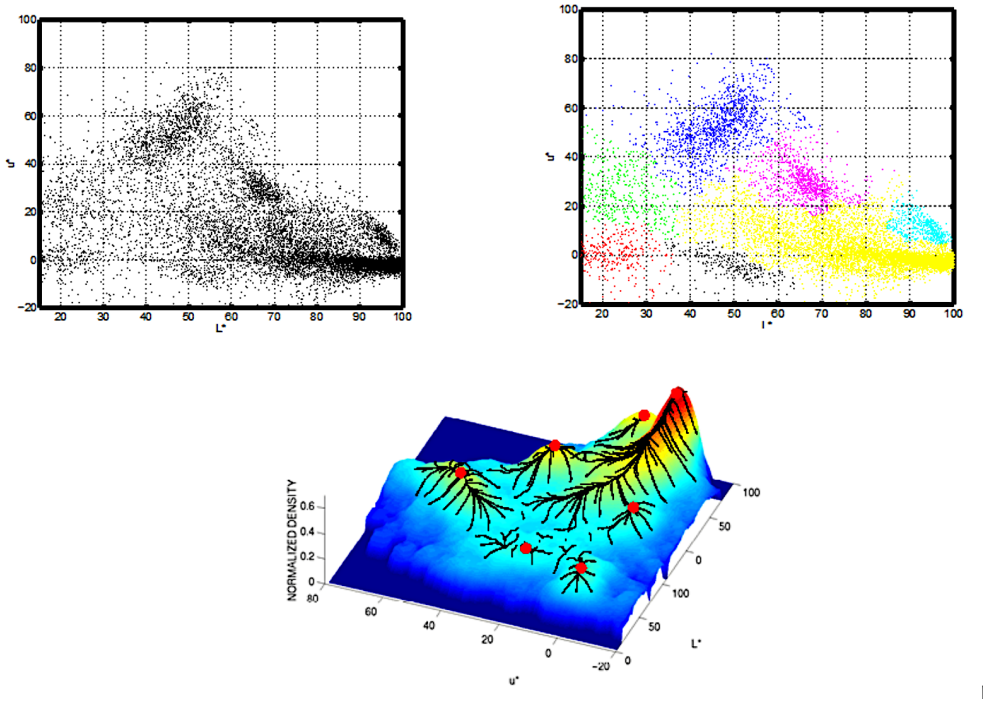


Figure 8: Attraction Basins in Mean Shift Clustering

Definition 4.1. Let $G = (V, E)$ be a graph with vertices or nodes V and connecting edges E. Each edge (u, v) has a weight $w(u, v)$ representing the similarity between u and v. Cut G into two disjoint graphs with node sets A and B by removing all edges between those sets. Assign a value to the cut as follows:

$$\text{cut}(A, B) = \sum_{u \in A, v \in B} w(u, v)$$

Minimal value cuts often lead to small groups. Normalized cuts offer a better approach by considering associations and dissimilarities:

$$\text{Ncut}(A, B) = \frac{\text{cut}(A, B)}{\text{assoc}(A, V)} + \frac{\text{cut}(A, B)}{\text{assoc}(B, V)}, \text{ where } \quad \text{assoc}(A, V) = \sum_{u \in A, t \in V} w(u, t)$$

is the sum of all edges within group A as well as those connecting it to the nodes outside.

4.4 Superpixels

The inefficiency of graph calculations with many pixels in images is addressed by calculating *superpixels*—grouping similar pixels. This cost-effective method may lead to over-segmentation, but applying normalized graph cuts to superpixels can mitigate this issue.

4.5 Features

Raw input pixel values are not directly used as features to avoid a large input space. Instead, feature vectors are calculated for each pixel or set of pixels to capture specific properties of an image or image region. These feature vectors are crucial for various tasks, including classification, subregion detection, and image stitching.

4.5.1 Feature Properties

Features should be invariant to translation, rotation, scaling, illumination changes, color variations, and transformations (skewing).

4.5.2 Filter Banks

Filter banks, as illustrated in Figure 9, consist of RFS Filters (edge and line filters at six orientations and three scales, Gaussian, and Laplacian of Gaussian) and MR8 Filters (Maxima of RFS Filters at each scale).

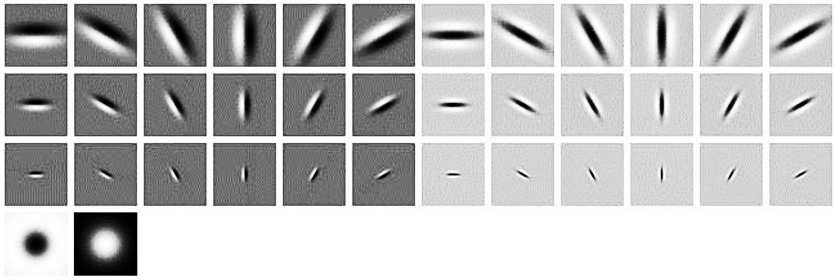


Figure 9: Filter Banks

4.5.3 Texture Characteristics

Textons utilize texture descriptors as vectors of filter bank outputs. Textons obtained through clustering form a dictionary, and the similarity between regions in the texton histograms is used for comparison.

4.5.4 Grey Level Co-occurrence Matrices (GLCMs)

GLCM measures how often a specific combination of color values between a pixel and its neighbors occurs. This results in a 256×256 matrix for the 256 grey values. Additional features like entropy, energy, homogeneity, contrast, or dissimilarity can be calculated from the matrix.

4.5.5 Scale Invariant Feature Transform (SIFT)

SIFT is a popular feature descriptor for tasks such as object matching and image stitching. It combines detection and description, using the histogram of the gradient and resulting in a 128-dimensional vector.

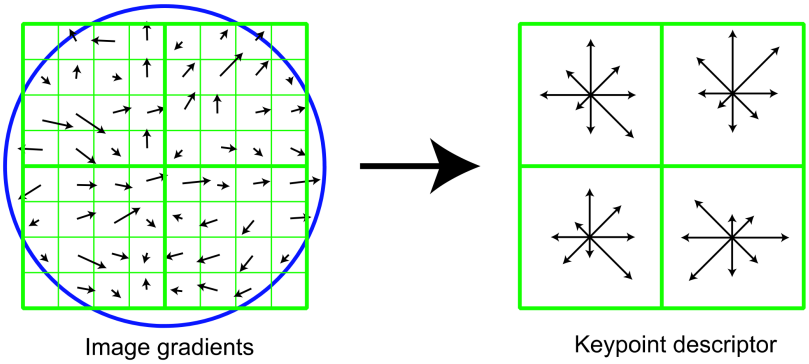


Figure 10: Scale Invariant Feature Transform (SIFT)

4.5.6 Histogram of Oriented Gradients (HOG)

HOG computes gradient images in x and y and divides the image into cells. It then calculates the histogram of gradient orientation in each cell, normalizes, and flattens it into a feature vector.

4.5.7 HOGles

HOGles "inverts" HOG features to simulate the image that generates them. This tool is useful for understanding bad detections.

4.6 Performance Metrics

Metrics for binary classification:

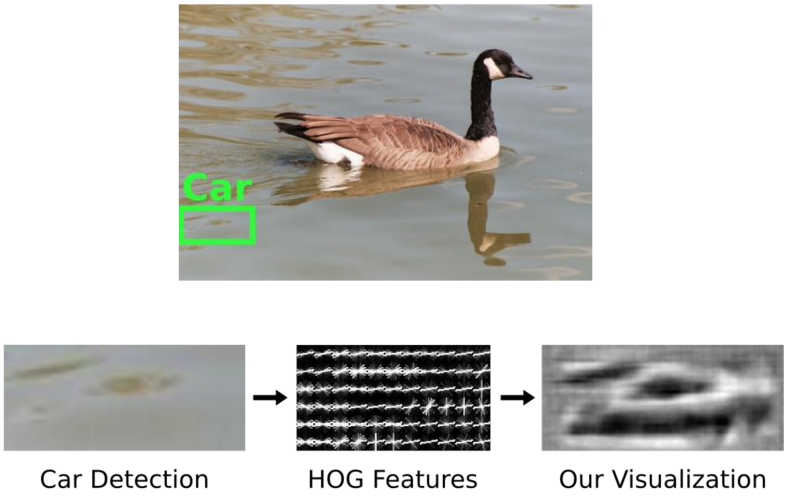


Figure 11: HOGles: Inverted HOG Features

$$\text{Precision} \quad \frac{\text{TP}}{\text{TP}+\text{FP}} \qquad\qquad \text{Accuracy} \quad \frac{\text{TP}+\text{TN}}{\text{TP}+\text{TN}+\text{FP}+\text{FN}}$$

$$\text{Recall} \quad \frac{\text{TP}}{\text{TP}+\text{FN}} \qquad\qquad \text{F1 score} \quad \frac{2\cdot\text{TP}}{2\cdot\text{TP}+\text{FP}+\text{FN}}$$

$$\text{Specificity} \quad \frac{\text{TN}}{\text{TN}+\text{FP}} \qquad\qquad \text{IoU (Intersection over Union)} \quad \frac{\text{TP}}{\text{TP}+\text{FP}+\text{FN}}$$

5 Convolutional Neural Networks (CNNs)

5.1 Network Layer Properties

Given an input shape of `(n, n, c)` and ignoring the batch dimension (`keras` default shape is `(batch_size, height, width, channels)`). Convolution layer arguments are `filters=f` and `kernel_size=k`.

Layer	Output shape	# of params
`Conv2D(f, k, padding='same')`	`(n, n, f)`	$k \cdot k \cdot c \cdot f + f$
`Conv2D(f, k, padding='valid')`	`(n-k+1, n-k+1, f)`	$k \cdot k \cdot c \cdot f + f$
`Conv2D(f, k, strides=(sr, sc), padding='same')`	`(n/sr, n/sc, f)`	$k \cdot k \cdot c \cdot f + f$
`Conv2DTranspose(f, k, strides=(sr, sc), padding='same')`	`(sr*n, sc*n, f)`	$k \cdot k \cdot c \cdot f + f$
`MaxPool2D(pool_size=(sr, sc))`	`(n/sr, n/sc, c)`	0
`UpSampling2D(size=(sr, sc))`	`(sr*n, sc*n, c)`	0

The required **memory** of a layer is equal to the product of the output shape of a layer (times whatever the data type requires).

The **receptive field** of a given neuron can be calculated as follows:

1. Start with the dimension of the last convolutional kernel (e.g. 3×3).
2. For every layer further upp, extend the area as follows:
   - For every pooling layer of size $p \times p$, increase the area dimensions by $2 \cdot (p-1)$.
   - For every convolutional layer of kernel size $k \times k$, increase the area dimensions by $k-1$.

6 Finding Multiple Objects

6.1 Viola Jones Face Detection

The core problem with face detection (or object detection in general) is that there are many positions that have to be scanned for a possible face. Viola and Jones introduced a solution to this based on three main principles:

- **Boosting**: (Linearly) combine multiple weak classifiers into a strong one.
- **Haar Features**: Use simple filters that only contain $+1$ or -1 in their masks.
- **Cascaded classifiers**: Cascade multiple classifiers and reject false results early in the cascade.

6.1.1 Boosting using AdaBoost (Adaptive Boosting)

Given input training data with weights, at each step $t \in 1..T$, train a weak classifier. Let it classify the training data and increase the weight on incorrectly classified samples (therefor adaptive). The final strong classifier $F(x)$ is a weighted, linear combination of the weak classifiers.

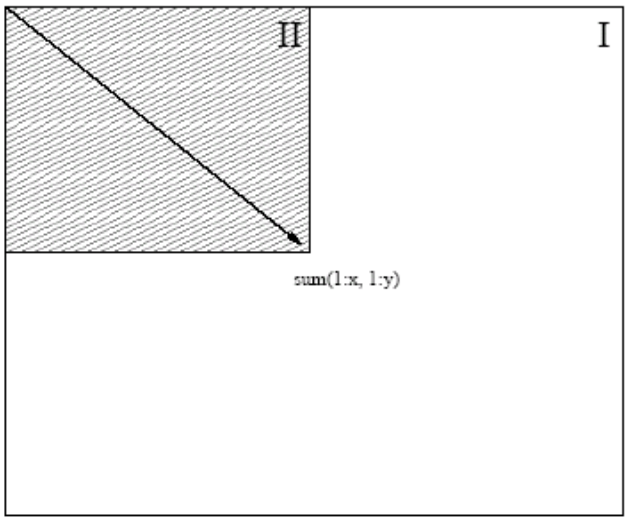
$$F(x) = \alpha_1 f_1(x) + \alpha_2 f_2(x) + \alpha_2 f_2(x) + \dots$$

The weak classifier $h_j(\theta)$ is defined with features f, threshold θ, and parity p_j:

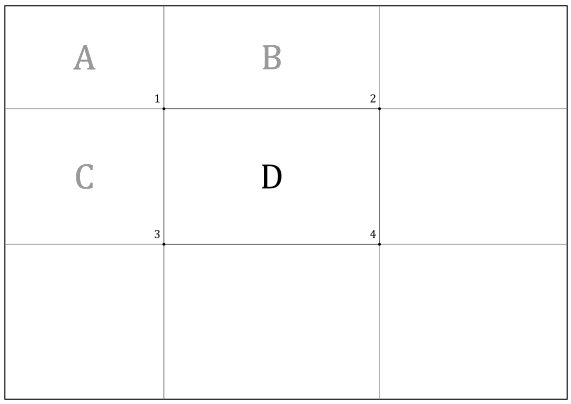
$$h_j(\theta) = \begin{cases} 1 & p_j f_j < p_j \theta_j \\ 0 & \text{otherwise} \end{cases}$$

6.1.2 Haar Features

To avoid computationally intensive filter responses, Haar features use simple filters with -1 or 1 in masks. Features are calculated on sliding sub-windows (e.g., 24×24 pixels) using integral images for fast computation. The humongous size of features can be evaluated efficiently using integral images.



(a) Integral Image Visualization



(b) Calculating Haar Features with Integral Image

6.1.3 Cascaded Classifiers

The challenge is evaluating thousands of possible position or scale combinations efficiently, with true faces being rare. A classifier with a high *true* positive rate ($85\% \dots 95\%$) and a very low *false* positive rate ($10^{-5} \dots 10^{-6}$) can be achieved by cascading classifiers with a very high TPR ($\sim$99%) and a moderately low FPR ($\sim$30%) (TPR and FPR rates can be multiplied). False results are rejected early in the processing.

6.2 HOG for Human Detection

Detection pipeline:

- Calculate HOG Descriptors
- Collect HOGs in Detection Window
- Use Linear SVM for Person or Non-Person Classification

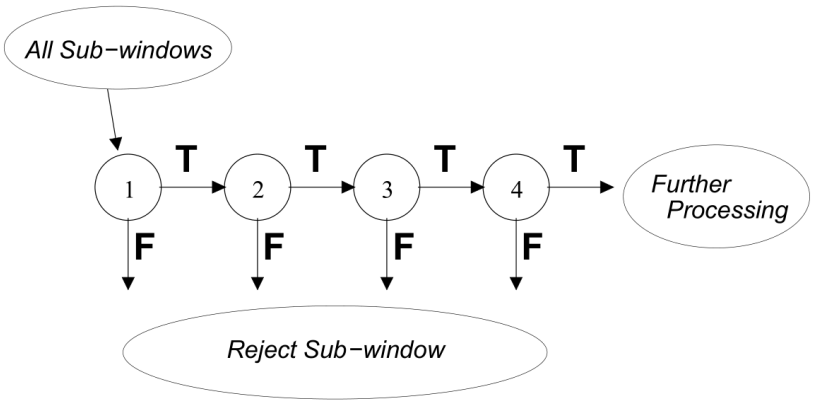


Figure 13: Cascade Classifier Approach

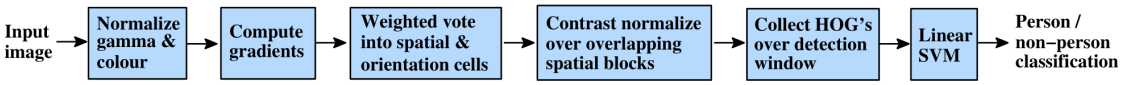


Figure 14: HOG for Human Detection

6.3 OverFeat

Combining Classification, Localization, and Detection involves:

- Classification
  - CNN
  - Sliding Windows applied at every possible pixel and at multiple scales
  - Subsampling factor of 12
- Localization
  - Generate Bounding Box Prediction as a Regression Problem
  - Uses the first five layers in the CNN
- Detection
  - Combine Classification and Localization
  - Merge and accumulate bounding boxes

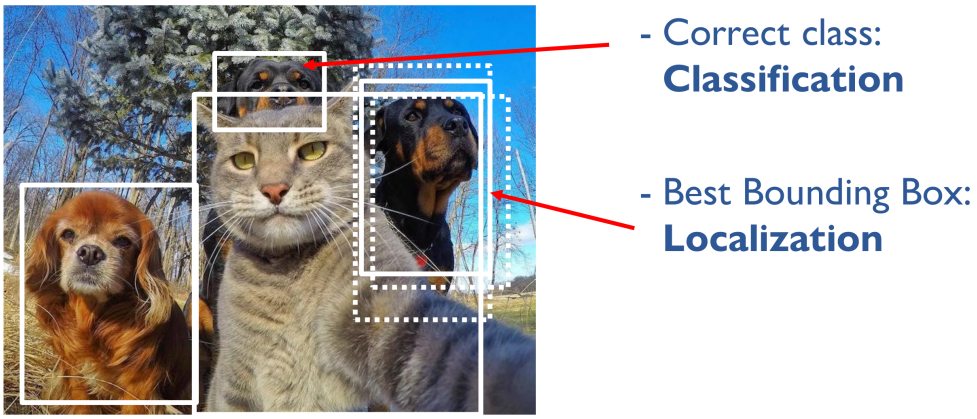


Figure 15: Classification and Localisation Visualization

Sliding windows have the drawback of a huge number of different positions and sizes that need evaluation. A better idea is to find image regions likely to contain an object, called region proposals.

6.4 Region Proposals

To avoid the huge number of possible positions and sizes that need to be evaluated with the sliding window approach, image regions that are likely to contain an object are identified. This is relatively fast for e.g. 1000 proposals (much less than sliding window approach). The regions can, for example, be obtained by Region Segmentation (graph based) or Hierarchical Grouping.

6.5 R-CNN (Region-based CNN)

Extract region proposals, compute CNN features, and classify regions using SVM.

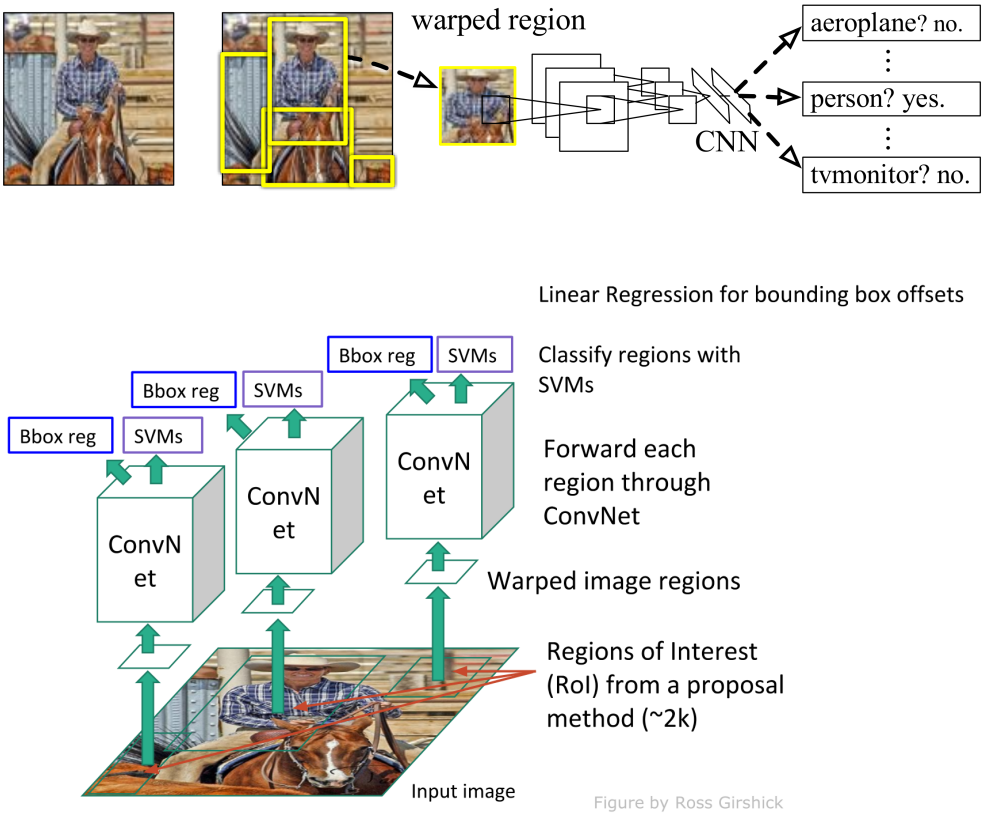


Figure 16: R-CNN Structure

6.5.1 Feature Extraction

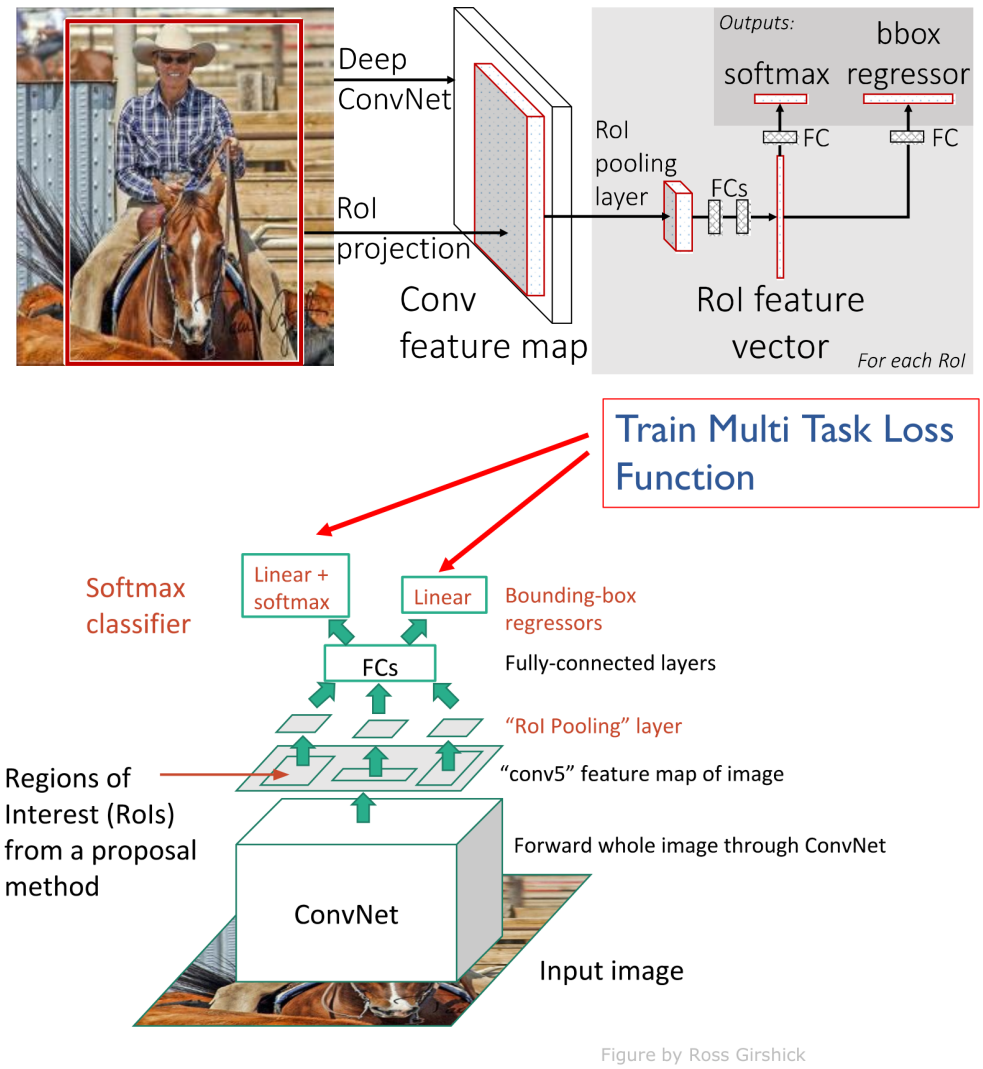
- Proposal regions are warped to 227×227 pixel images
- 4096 Features from 227×227 RGB Image using five Convolutional and two fully connected layers from AlexNet Architecture
- Supervised pre-training on the ILSVRC2012 classification data set
- Domain-specific fine-tuning continues training on warped region proposals

6.5.2 Classifier

Use one linear SVM per class, considering regions with overlap of 30% as a positive example. Treat bounding box position as a regression problem.

6.5.3 FAST R-CNN

Replace SVM of R-CNN with a fully connected neural network for classification of objects and refined bounding boxes.



6.6 You Only Look Once (YOLO)

YOLO is a single neural network predicting bounding boxes and classification probabilities. It is fast with good classification but less accurate localization.

6.6.1 Improvements

- Add Batch Normalization
- Use higher resolution (448×448)
- Use Anchor Bounding Boxes from k-Means Analysis on the training data
- Use a new base net (darknet19 and darknet 58)

7 Generating Images

7.1 Conditional GANs for Image-to-Image Transfers

A GAN learns the mapping from random noise z to output image y:

$$G : z \to y$$

In conditional GANs, the goal is to learn the mapping from input image x and noise z to output image y:

$$G : \{x, z\} \to y$$

Adding noise as input is ineffective, and one solution is to use dropout as noise, both during training and testing.

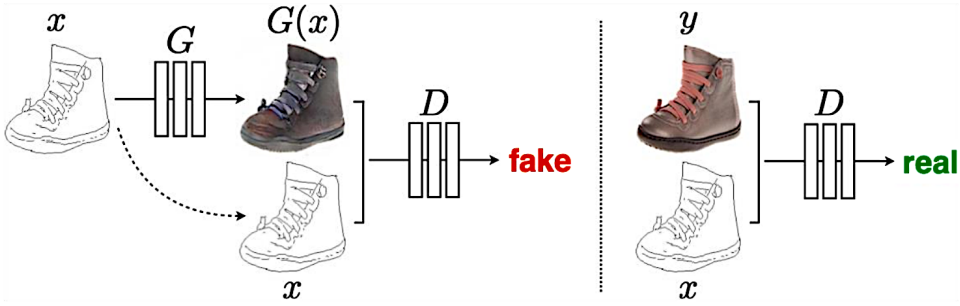


Figure 17: In conditional GANs, both the Generator G and the Discriminator D input the edge map.

7.1.1 Conditional GANs

Generator architectures with U-Net and connections between layers seem to work better.

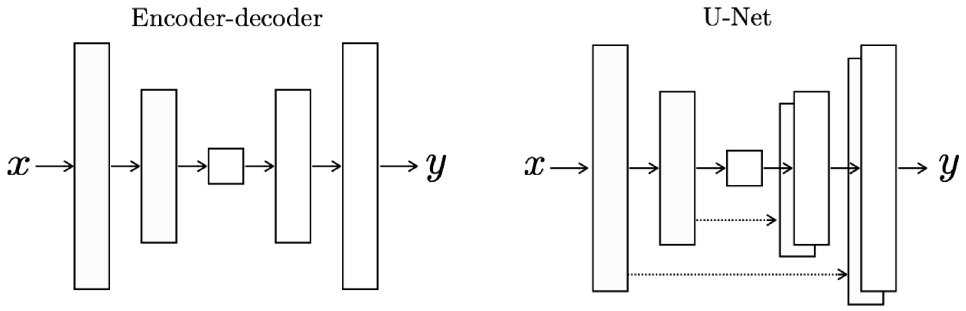


Figure 18: U-Net architectures with connections between the layers.

7.2 Neural Texture Synthesis

Given a low-resolution texture image, two main approaches for generating a larger image are:

- Replicate pixels and patches using a suitable algorithm.
- Find a model of the texture and generate new textures from the model.

7.2.1 Texture Synthesis Using CNNs

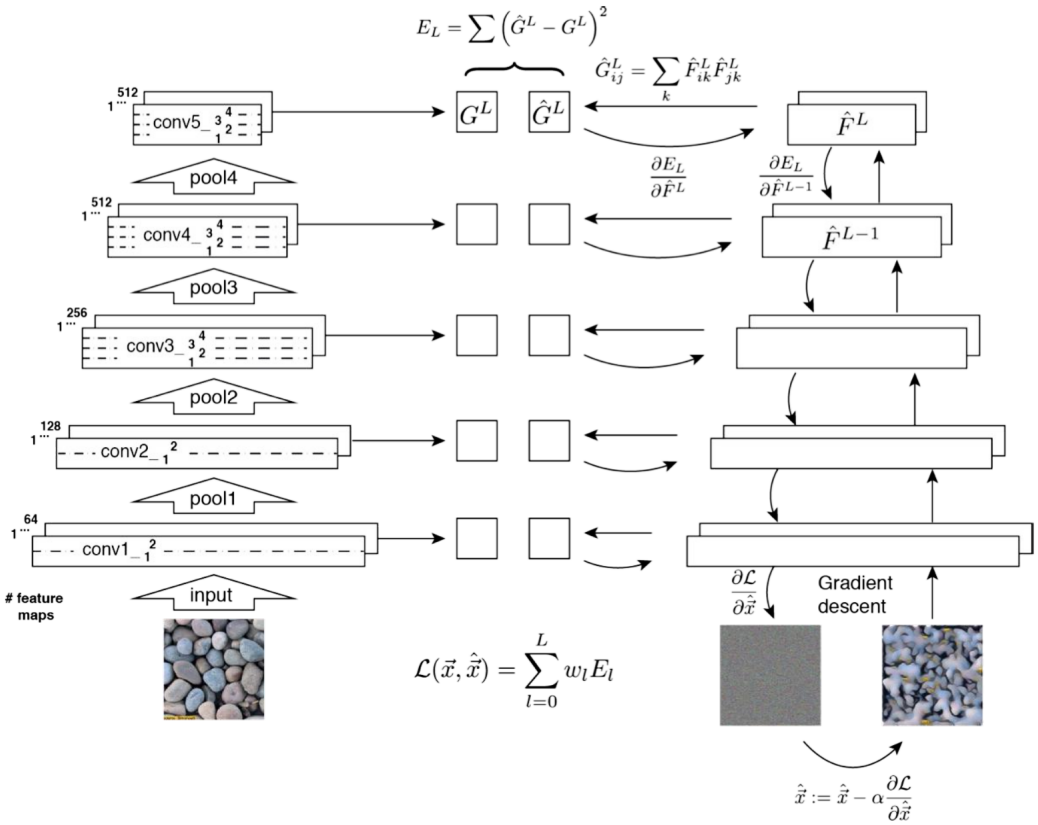
Pass image x through the network; the activations from each layer are the feature maps $F^l \in \mathbb{R}^{N \times M}$ for the texture. Calculate the Gram matrix to compute correlations between feature maps (on the same layers):

$$G^l_{ij} = \sum_k F^l_{ik} F^l_{jk}$$

These Gram matrices describe the texture model.

7.2.2 Generating Textures

1. Start with a noise image.
2. Use gradient descent to find another image that matches the Gram-matrix representation of the original image.
3. Optimize by minimizing the mean-squared distance between the matrices from the original and generated images.



7.3 Neural Style Transfer

7.3.1 Content Representation

Each input image generates filter responses $F^l \in \mathbb{R}^{N \times M}$ at each layer. Run gradient descent on noise images to find an image that generates the same response. If $F^l \in \mathbb{R}^{N \times M}$ is the response from the original image and $P^l \in \mathbb{R}^{N \times M}$ the response from the generated image, the loss is:

$$Ł_{\text{content}}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} (F^l_{ij} - P^l_{ij})^2$$

Later layers in the network define the content.

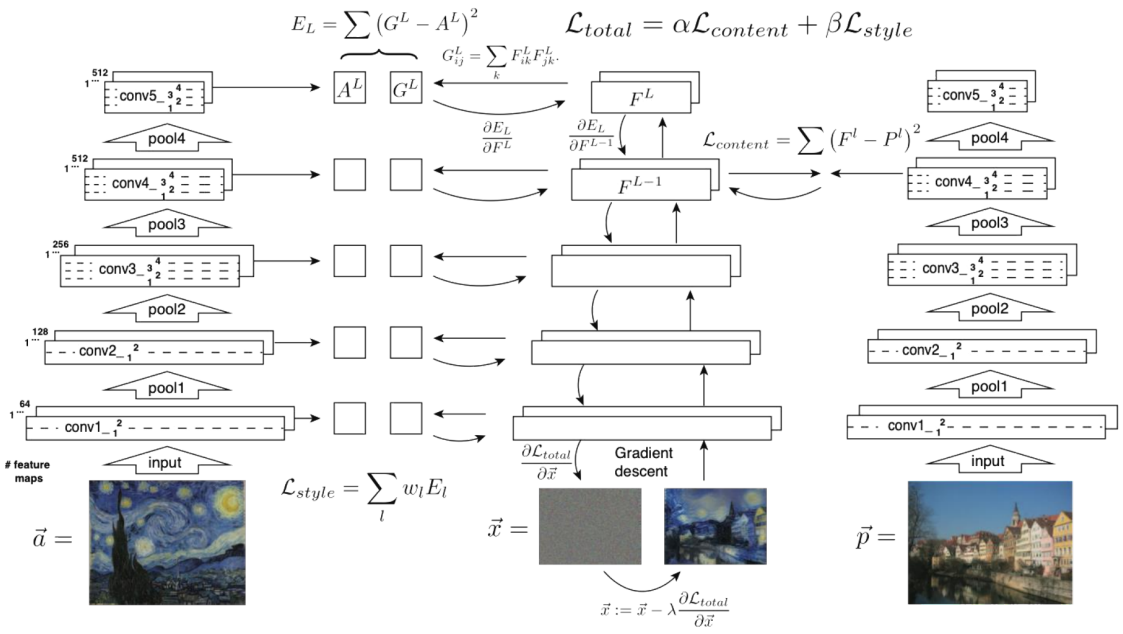
7.3.2 Style Representation

Each input image generates filter responses $F^l \in \mathbb{R}^{N \times M}$ at each layer. Calculate Gram matrices from the features. If A^l is the matrix from the original image and G^l the matrix from the generated image, the loss for one layer is:

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G^l_{ij} - A^l_{ij})^2$$

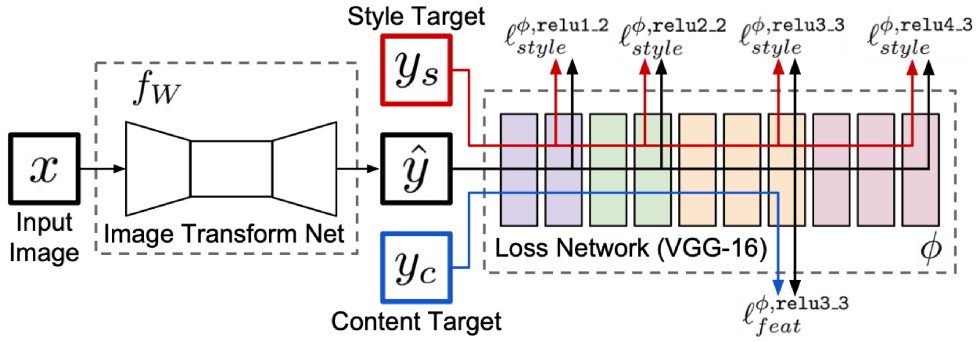
Including several layers:

$$Ł_{\text{style}}(\vec{a}, \vec{x}) = \sum_{l=0}^{L} w_l E_l$$



7.3.3 Fast Style Transfer

Style transfer is slow, requiring many passes through VNN. The solution is to train a neural network to perform style transfer. Train a feedforward network for each style, use a pretrained CNN with losses as before, and stylize an image with a single forward pass.



8 Python Code

8.1 General Image Handling

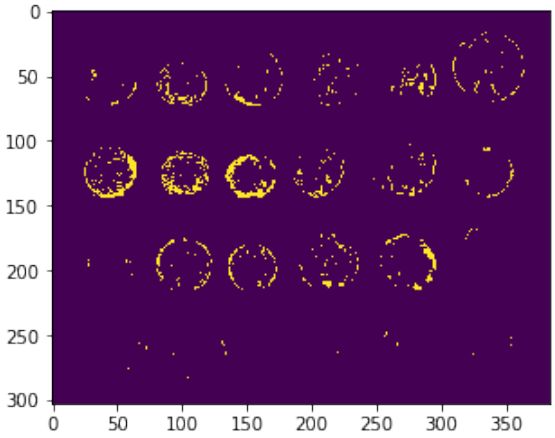
```
# Image dimensions
im = skimage.io.imread("data/snoopy.png")
# axis 0: rows (y-axis), going from top (0) to bottom
# axis 1: columns (x-axis), going from left (0) to right
# axis 2: color channels [and alpha transparency] (r, g, b, [a]), if applicable
# -> general shape of an image is (r, c), (r, c, 3) or (r, c, 4)

# Printing images in grayscale with matplotlib
plt.imshow(im,
           vmin=0,   # value which maps to black
           vmax=255, # value which maps to white
           cmap="gray")
# Floating point images
im_float = im/255
plt.imshow(im_float, cmap="gray", vmin=0, vmax=1)
```

```
# Stacking images
np.vstack((im_float, im_float))  # vertical stacking
np.hstack((im_float, im_float))  # horizontal stacking

# uint8 arithmetic wraps around
print(np.array([255],dtype="uint8")+1)
> [0]
```

8.2 Binarization and Filtering

8.2.1 Binary Mask

```
mask = im > 0.8 # bright pixels will be True in the mask, others will be False
plt.imshow(mask)
```



8.2.2 Connected Component Analysis

```
import skimage.measure
labels = skimage.measure.label(im > 0.5)
print(labels.shape, labels.dtype)
print("Unique values in labels:", np.unique(labels))
> (303, 384) int64
> Unique values in labels: [  0   1   2   3   4   5   6   7   8   9  10
> 11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28
> 29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46
> 47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64
> 65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82
> 83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100
> 101 102 103 104 105 106 107  108 109 110 111 112 113 114 115 116 117 118 119]

regions = skimage.measure.regionprops(labels)

large_regions = [r for r in regions if r.area > 100]

# equivalent to
large_regions = []
for r in regions:
        if r.area > 100:
                large_regions.append(r)

print(f"There are {len(large_regions)} large regions")
> There are 25 large regions
```

The `regionprops` (documentation) function can compute properties for each connected component. Some important properties are the following:

`area : int` Number of pixels of the region.

`bbox : tuple` Bounding box (`min_row, min_col, max_row, max_col`). Pixels belonging to the bounding box are in the half-open interval `[min_row; max_row)` and [min_col; max_col).

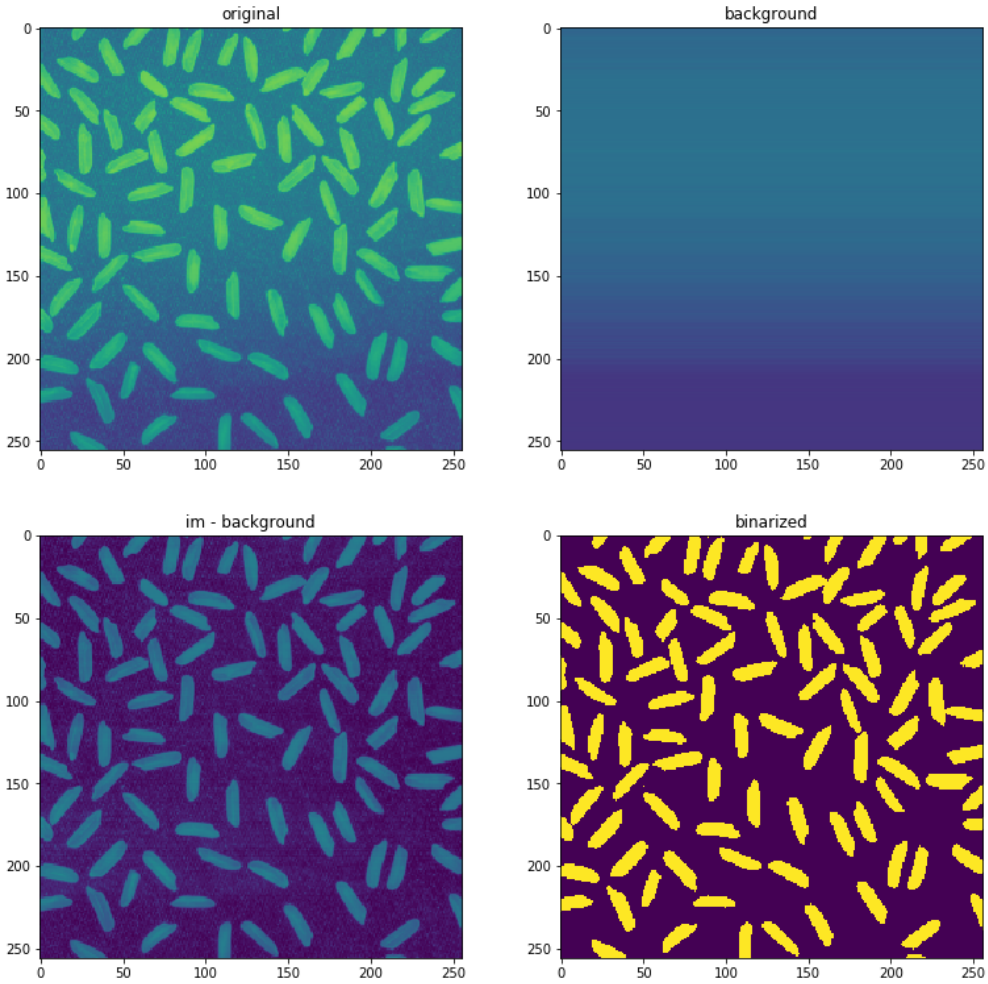
`centroid : array` Centroid coordinate tuple (row, col).

`convex_area : int` Number of pixels of convex hull image, which is the smallest convex polygon that encloses the region.

`label : int` The label in the labeled input image.

```
import matplotlib.patches as patches
fig, ax = plt.subplots()
ax.imshow(im, cmap="gray")
for r in large_regions:
        (min_row, min_col, max_row, max_col) = r.bbox
        width = max_col - min_col
        height = max_row - min_row
        rect = patches.Rectangle((min_col,min_row),width,height,
        linewidth=1,edgecolor='b',facecolor='none')
        ax.add_patch(rect)
```

```
background = (np.min(im,axis=1,keepdims=True) * np.ones((1,im.shape[1])))

fig, axs = plt.subplots(ncols=2, nrows=2, figsize = (13,13))
axs[0,0].imshow(im, vmin=0, vmax=1)
axs[0,0].set(title="original")
axs[0,1].imshow(background, vmin=0, vmax=1)
axs[0,1].set(title="background")
axs[1,0].imshow(im - background, vmin=0, vmax=1)
axs[1,0].set(title="im - background")
mask = (im - background) > 0.25
axs[1,1].imshow(mask, vmin=0, vmax=1)
axs[1,1].set(title="binarized")
```



8.3 Canny Edge Detection

```
import skimage.feature
im_edges = skimage.feature.canny(im, sigma=2.0, low_threshold=0.2, high_threshold=0.8)
fig, ax = plt.subplots(figsize=(10,10))
ax.imshow(im_edges)
```

8.4 Model Fitting Hough Lines

```
import skimage.feature
import skimage.transform.hough_transform as ht
import matplotlib.pyplot as plt
import numpy as np

import skimage.data
import skimage.feature

im = skimage.data.camera()
imedges = skimage.feature.canny(im)

lines = ht.probabilistic_hough_line(imedges, threshold=100)
len(lines)

# Draw detected lines
fig,ax = plt.subplots(figsize=(10,10))
```

```
ax.imshow(im,cmap="gray")
for ((x0,y0),(x1,y1)) in lines:
ax.plot([x0,x1],[y0,y1],'b-')
```

8.5 Superpixel Segmentation

```
# import the necessary packages
from skimage.segmentation import slic
from skimage.segmentation import mark_boundaries
from skimage.util import img_as_float
from skimage import io
import matplotlib.pyplot as plt
import argparse

# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True, help = "Path to the image")
args = vars(ap.parse_args())

# load the image and convert it to a floating point data type
image = img_as_float(io.imread(args["image"]))

# loop over the number of segments
for numSegments in (100, 200, 300):

# apply SLIC and extract (approximately) the supplied number
# of segments
segments = slic(image, n_segments = numSegments, sigma = 5)

# show the output of SLIC
fig = plt.figure("Superpixels -- %d segments" % (numSegments))
ax = fig.add_subplot(1, 1, 1)
ax.imshow(mark_boundaries(image, segments))
plt.axis("off")

# show the plots
plt.show()
```

8.6 CNNs

8.6.1 U-Net-like Architecture for Semantic Segmentation

```
inputs = Input(shape=(input_height, input_width, channels))
x = inputs

x = Conv2D(filters=8, kernel_size=3, padding='same', activation='relu')(x)
t_1 = x

x = Conv2D(filters=8, kernel_size=3, strides=(2, 2), padding='same', activation='relu')(x)
t_2 = x

x = Conv2D(filters=16, kernel_size=3, padding='same', activation='relu')(x)
t_3 = x

x = Conv2D(filters=16, kernel_size=3, strides=(2, 2), padding='same', activation='relu')(x)
```

```
t_4 = x

x = Conv2DTranspose(filters=16, kernel_size=3, strides=(2, 2), padding='same')(x)
x = Concatenate(axis=3)([x, t_3])
x = Conv2D(filters=16, kernel_size=3, padding='same', activation='relu')(x)

x = Conv2DTranspose(filters=32, kernel_size=3, strides=(2, 2), padding='same')(x)
x = Concatenate(axis=3)([x, t_1])
x = Conv2D(filters=32, kernel_size=3, padding='same', activation='relu')(x)

x = Conv2D(filters=num_classes, kernel_size=1, activation='softmax')(x)

model = Model(inputs=inputs, outputs=x, name='unet')
model.compile(loss='categorical_crossentropy', metrics=['accuracy'])
```