# Notes CompVis

**Grayscale images:** 2-dimensional arrays. **Note:** our perceived brightness is mediated by our brain

**uint8 arithmetic:** one byte per pixel (values 0 (black) - 255 (white)). It wraps around! For floating point 0 (black) and 1 (white). `print(np.array([0], dtype="uint8")-1)  # [255]`

`a = np.array([100], dtype="uint8")`, `print(a, a*2, a*3)  # [100] [200] [44]`, `np.vstack((im, im*2, im+0.5))` (contrast, brightness), `cmap="gray", vmin=0, vmax=1`)

**Color images:** `im[:,:,0]  # red, im[:,:,1]  # green, im[:,:,2]  # blue` Sometimes a 4 channel for transparency. A pixel with the color red that is shown with full opacity would have a high $\alpha$ value.

```
# Flag of Switzerland
mask = np.zeros((200, 200))
mask[80:120, 40:160] = 1
mask[40:160, 80:120] = 2
flag = np.full((200, 200, 3), [255, 0, 0])
flag[mask==1] = [255, 255, 255]
flag[mask==2] = [255, 255, 255]
plt.imshow(flag)
```

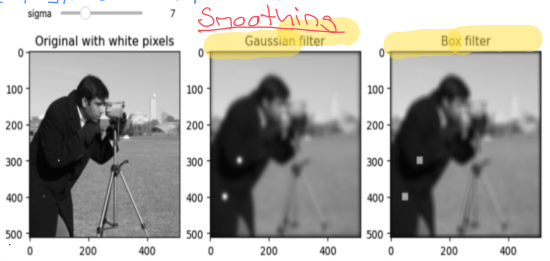
**Convolutions:** Weighted moving average example $a = [1, 4, 6, 4, 1]/16$

**Padding:** 'valid' only compute convs for windows that fully overlap the input image. New image is smaller! `convolve2d(matrix, kernel[::-1,::-1], mode='valid')`

'constant' (= 'same') input is extended by a defined value. Input size = = Output size `convolve(input, kernel[::-1,::-1], mode='constant', cval=0.0)`

**Smoothing:** Box Filter (moving average): just averages all values `kernel = np.full((size, size), 1/(size*size))`

Gaussian Filter (weighted moving average): gives more weight to the central pixels. Variance (sigma) determines the amount of smoothing. Larges sigma → wider smoothed area. Rule of thumb: Set kernel size $k \approx 2\pi\sigma$ (also $3\sigma$ sigma)




**Sharpening:** First we smooth the image, then: original − smoothed = detail; original + detail = sharpened

**Edge detection:** $\frac{\partial f}{\partial x}$ (→): $\boxed{-1\ \ 1}$ $\frac{\partial f}{\partial y}$ (↓): $\boxed{\begin{matrix}-1\\1\end{matrix}}$

Larges values of sigma: larger scale edges detected

Smaller values of sigma: finer details detected

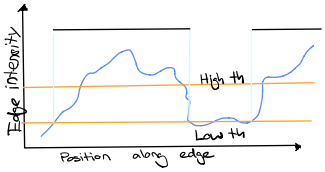
In practice: To compute gradients, convolve with a derivative-of-gaussian filter. This equivalent to smoothing with gaussian and then taking the derivative

**Sobel:** $G_x = \begin{bmatrix}-1 & 0 & +1\\ -2 & 0 & +2\\ -1 & 0 & +1\end{bmatrix}$ and $G_y = \begin{bmatrix}-1 & -2 & -1\\ 0 & 0 & 0\\ +1 & +2 & +1\end{bmatrix}$

`im_sobel = (im_sobelh ** 2 + im_sobelv ** 2) ** 0.5` (Pythagoras) → to obtain the module of the gradient regardless on the direction.

$G = \sqrt{G_x^2 + G_y^2}$ → how big is the edge  $\operatorname{atan}\left(\frac{G_y}{G_x}\right)$ → orientation of the edge

**Canny edge detector:** Convolving with derivative-of-gaussian filter with smoothing parameter sigma. Makes edges 1-pixel-wide (thinning): Non-Maxima Suppression and only keeps strong edges: hysteresis thresholding ✓ ✗




**Binarization:** Thresholding: `mask = im > threshold` (in case of darker background)

**Connected Component Analysis:** `labels = label(mask), regions = regionprops(labels)`

```
@widgets.interact(threshold = (0,1,0.01), minsize = (10,500))
def f(threshold=0.51, minsize=300):
    fig, ax = plt.subplots()
    ax.imshow(im, cmap="gray")
    mask = im > threshold
    labels = skimage.measure.label(mask)
    regions = skimage.measure.regionprops(labels)
    large_regions = [r for r in regions if r.area > minsize]
    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)
```

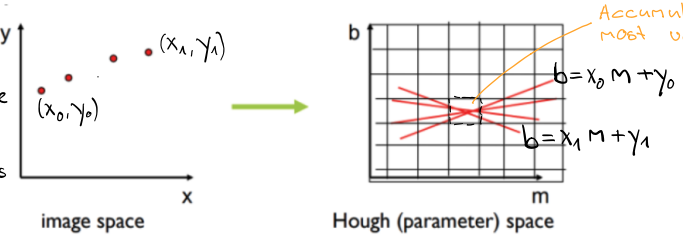
Example: uniform gray background

white objects → white = `im > 0.75`

black objects → black = `im < 0.25`

**Hough transform:**

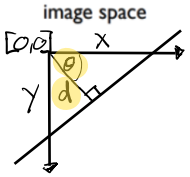
1. Every edge point casts votes for all lines that are compatible with it.
2. We choose lines that accumulated a lot of votes




Accumulate votes in a discrete set of bins; parameters with the most votes indicate a line in image space.

**Problem:** Representing the line with parameters b and m leads to lines with length of infinity!

**Polar coordinates:**




d: perpendicular distance from line to origin

θ: angle perpendicular makes with the x-axis

$x \cos\theta - y \sin\theta = d$

→ The hough space is now $(d, \theta)$ instead of $(m, b)$. But a line in image space is still a point in Hough space as before.

**Hough algorithm**

Initialize H[d,θ]=0

for each edge point (x,y) in the image:
  for θ in range(θmin,θmax):
    d = x cos(θ) - y sin(θ)
    H[d,θ]+=1

Find the value(s) of (d,θ) where H[d,θ] is maximum

The detected line in the image is given by
d = x cos(θ) - y sin(θ)

**Bin size in accumulator:** (Solution for small bin: vote also for neighbours in the accumulator ("smoothing" accumulator image))

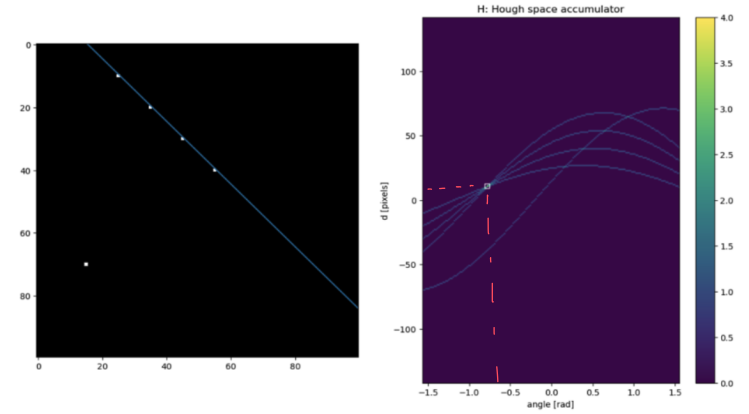
- **Too small:** - many weak peaks due to noise
- **Just right:** - one strong peak per line, despite noise
- **Too large:** - poor accuracy in locating the line
  - many votes from clutter might end up in the same bin

**Extension 1:** We know the direction of the gradient for each edge pixel.
- Edge direction is orthogonal to gradient direction → only vote for lines that have (almost) the direction of the edge. - Reduces comp. time and num. of useless votes

**Extension 2:** Keep track of edge points that voted for each line. If a line is chosen, look for groups of edge points that voted for that line. → find start and end point of segment

`probabilistic_hough_line` returns line segments $((x0, y0), (x1, y1))$.

```
im = skimage.data.camera()
imedges = skimage.feature.canny(im)
lines = ht.probabilistic_hough_line(imedges, threshold=100)
lines[0]  # ((406, 509), (287, 285))
```




```
H, angles, distances = ht.hough_line(imedges)
(maxr, maxc) = np.unravel_index(np.argmax(H), H.shape)
d = distances[maxr], theta = angles[maxc], print(d, np.rad2deg(theta)) # 11.0  -45.0

#This is one point on the line in image space
p1 = np.array([d * np.cos(theta), d * np.sin(theta)])
#This is the unit vector pointing in the direction of the line
linedir = np.array([np.cos(theta + np.pi/2), np.sin(theta + np.pi/2)])
# These are two points very far away in two opposite directions along the line
p0 = p1 - linedir * 1000
p2 = p1 - linedir * 1000
```

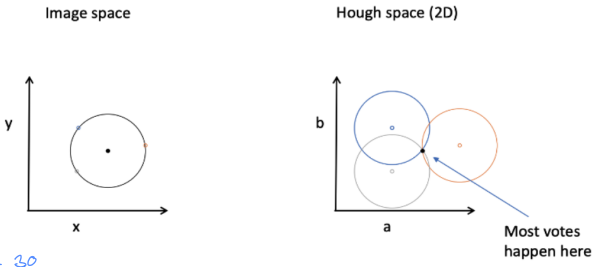
## Hough for circle detection:

**Fix radius:** circle is defined by **3 parameters** (center x, center y and radius)

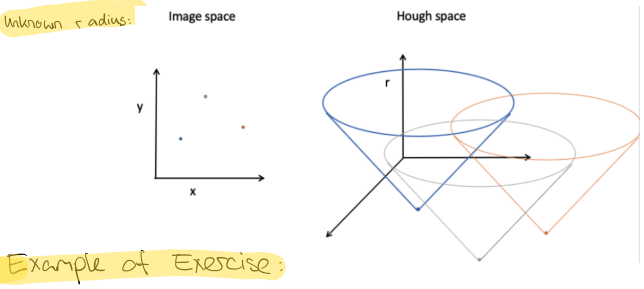
$(x_i - a)^2 + (y_i - b)^2 = r^2$ Center $(x=a, y=b)$ and radius $r$. 3 degrees of freedom.

**Hough space is 2D:** – a: x coord. of circle center. – b: y coord. of circle center

One point in image space maps to a circle in hough space




**Unknown radius:**



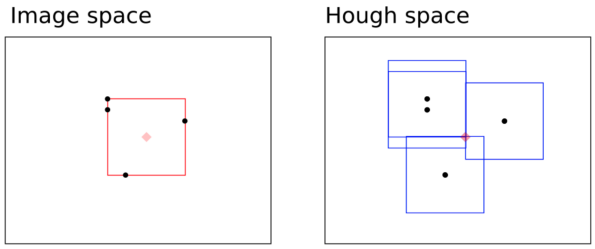

```
radius = 30
H = ht.hough_circle(im, [radius])
ridx, r, c = np.unravel_index(np.argmax(H), H.shape)

accums, cxs, cys, rads = ht.hough_circle_peaks(hspaces=H, radii=radii, threshold=0.4,
                                               min_xdistance=15, min_ydistance=15)
```

## Example of Exercise:

We want to design a detector for SQUARES with a given (known) edge length E based on the hough approach.

Assume that we expect the squares to be straight, i.e. that their edges are aligned to the x and y axes of the image. Also, the edge length E is known a priori and well defined (say: E=100 pixels).




- Describe a possible parametrization for your models. Hint: just like we parametrize a line by d and theta, we can paramerize a square by… x, y position of the center of the square
- How many dimensions does the hough space have? 2
- What does a point in the hough space represent in the image? A point in hough space at coord. (hx, hy) represents a square in the image which is centered on image pixel (hx, hy)
- How is a point in the image represented in the hough space? by a square in the hough space
- Assume we have a single square visible in the image. How does the hough space look like? In general, the hough space will have voks for many squares, but only one bin will accumulate voks by all points
- Assume we have a single square visible in the image, but we can only detect the lateral (vertical) edges. How does the hough space look like? The hough space will miss voks from horizontal edges!

CNN: Calculate params: Conv layer → kernel height × kernel width × num. input maps × output maps + Bias (1 for each output map)

Pool layer → no learnable parameters

FC → ((current layer neurons c * previous layer neurons p) + 1 * c)

Padding: same as on page 1

Stride: For padding $p$, filter size $f \times f$, input img size $n \times n$ and stride $s$ our dimension will be

$$\left[\{(n+2p-f+1)/s\}+1\right] \times \left[\{(n+2p-f+1)/s\}+1\right]$$

1×1: shift one pixel at a time

2×2: skips one patch horizontally and vertically

Sparse connectivity: A dense layer connects all inputs while a CNN only connects a local "patch" of pixels.

Receptive fields: Is defined as the size of the region in the input that produces the feature. Deeper neurons depend on wider patches of the input.

For two sequential conv layers $f_2, f_1$ with kernel size $k$, receptive field $r$: $r_1 = s_2 \times r_2 + (k_2 - s_2)$ initial $r_{last} = 1$

Example: Conv 4×4, Conv 3×3

$r_0 = 3 + (4-1) * 1 = 6 \times 6$

$r_1 = 1 + (3-1) * 1 = 3 \times 3$

Receptive field = 6×6

4 × 4 × 1 × 1 × 1 + 1 = 17

3 × 3 × 1 × 1 × 1 + 1 = 10

Num. of params = 27

Conv 2×2 | Max Pool 2×2 | Conv 2×2

$r_0 = 4 + (2-1) * 1 = 5 \times 5$

$r_1 = 2 + 2 * 1 = 4 \times 4$

$r_2 = 1 + (2-1) * 1 = 2 \times 2$

Receptive field = 5×5

2 × 2 × 1 × 1 × 1 + 1 = 5

0 = 0

2 × 2 × 1 × 1 × 1 + 1 = 5

Num. of params = 10

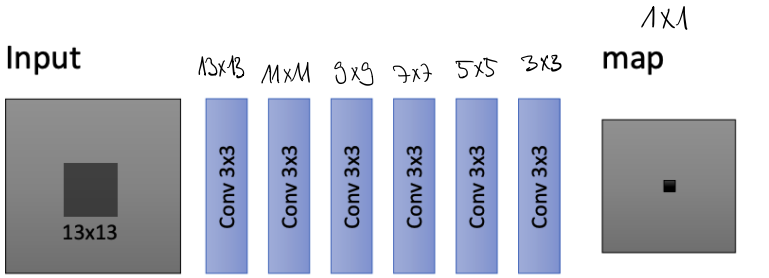
For receptive field max pool 3×3 → ×3!

Parameter sharing: sharing of weights by all neurons in a particular feature map. This leads to significantly less parameters compared to a dense layer.

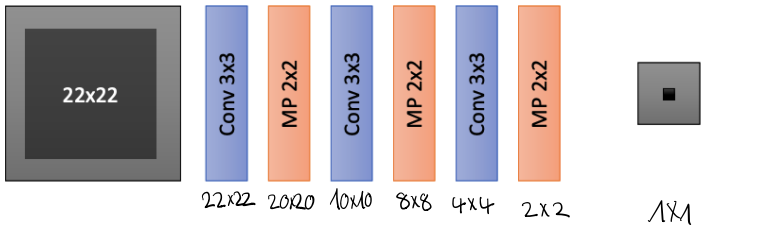
Translational invariance: Ability to ignore positional shifts, or translations, of the target in the image.

Different depth - different features:

- Shallow layers respond to fine, low-level patterns
- Deep layers respond to complex, high-level patterns




How large is the receptive field of the black neuron?



**Semantic Segmentation:** Understanding images at pixel level and assign class label to each pixel.

**Region based segmentation:** - **Split and Merge:** Split inhomogeneous regions in the image recursively (Split). Then merge neighboring regions with matching properties (Merge)
- **Watershed:** Interpret image as height map. We then fill image with water and calculate the lines of watersheds.

**K-Means clustering:** K is given. → Sum of all distances within points of a cluster should be minimal.
The values to be clustered can be anything for example: RGB, HSV, HS, Positions, Color and Position

**Lloyds Algorithm (for k-means clustering)**

- Initialize k Cluster centers (usually randomly)
  1. Calculate to which cluster each sample belongs
  2. Calculate the center point of each cluster
- Iterate until the clusters no longer change

**Mean Shift clustering:** Shape (and number?) of clusters should be variable and not be determined by the method (as in K-Means). The algorithm searches for the max. of the feature density from a given position in the image. It defines cluster as the set of positions that converge to the same maximum. Mean shift operates in the $L^*u^*v^*$ space.

**Problem:** Search for local maxima of the features, but the feature distribution is not available directly.
**Solution:** Kernel density Estimation to describe the distribution, for example using a Gaussian kernel

**Attraction basin:** Region, from which all trajectories lead to the same mode

**Mean shift algorithm**

For all positions x in the image:

- Calculate Mean m(x) and Environment N(x)

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

- Set x to the mean m(x)
- Iterate until x does not change

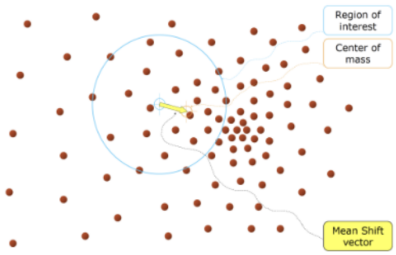



Figure 5.10: Start position of mean shift

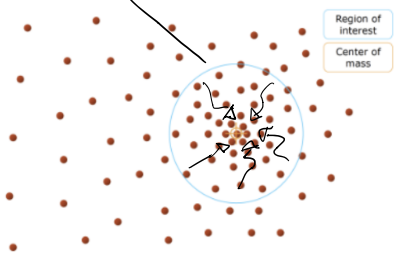



Figure 5.11: Final position of mean shift

**Mean shift algorithm (high level)**

- Calculate features
- Often: include spatial properties (pixel location) in features
- Initialize mean shift at each pixel
- Perform mean shift until convergence
- Merge starting positions that end up near the same peak into a cluster

**Graph cuts:** Image as Graph: - Node for every pixel - Edges between Nodes/Pixels - Affinity weight for each edge
- Measures similarity, for example inversely proportional to difference in color and position

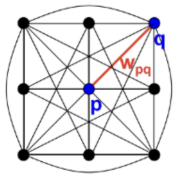


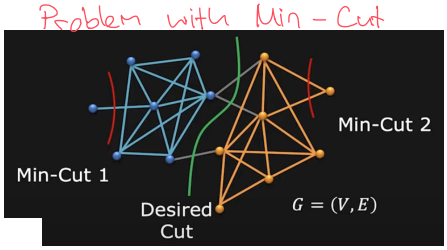

Figure 5.13: Images as graph

**Example**

- Let G = (V,E) be a graph.
- Each edge (u,v) has a weight w(u,v) that represents the similarity between u and v.
- Cut G into 2 disjoint graphs, with node sets A and B, by removing all edges between sets A and B
- Assign a value to the cut as

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

**Problem with Min-Cut**

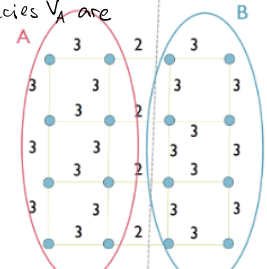



**Min-Cut:** Of all possible cuts cut(A,B) select the one with the minimal value
**Problem:** Minimal cut often cuts of small groups

**Normalized Cuts:** computes how strongly vertecies $V_A$ are associated with vertecies V

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

$$\text{assoc}(A,B) = \sum_{u \in A, t \in V} w(u,t)$$




assoc(A,V) = 38
assoc(B,V) = 38
cut(A,B) = 8
Ncut(A,B) = 16 / 38

**Superpixels:** Images contain many pixels and this is inefficient for graph calculations. → Calculate Superpixels first by grouping similar pixels. This is cheap but gives us oversegmentation. Then we apply normalized cuts

**SLIC (simple linear iterative clustering):** fast method to calculate Superpixels

**Idea:**

- Superpixels are regions with no internal edges
- The number of approximate superpixels is given
- Cluster centers are initialized regularly
- Local clusters are found in the space (L a b x y), i.e. color and pos

**SLIC Algorithm**

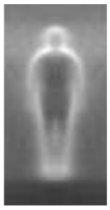
- Given k, set S = sqrt ( N / k)
- Initialize cluster centers in regular grid S pixels apart
- Move cluster centers in 3x3 neighborhood where gradient is lowest (to avoid having a cluster center on an edge)
- Repeat:
  - For each cluster center
    - For each pixel in 2Sx2S neighborhood (avoid large search regions as in k-Means)
    - Assign pixel to nearest cluster center
  - Update new cluster centers
  - Compute error E as difference between old and new cluster centers
- until E < threshold
- Postprocessing of *orphaned* pixels (use connected component analysis to ensure that clusters are connected)

**Features:** We calculate a feature vector for each pixel (or a sample of pixels). Features should be invariant to: Translation, Rotation, Scaling, Illumination (color) changes

**Features for images:** **Histogram of Oriented Gradients (HOG):**
- Compute gradient images in x and y
- Divide image into cells
- Compute HOG (multiplied by gradient value) in cell
- Normalize
- Flatten into feature vector

Are not invariant to scaling and translation

**HOGgles:** Inverts HOG features to simulate (reconstruct) the image that generates it. Helps to understand bad detections



Average Gradient image from a dataset for person detection

Test Image

HOG Descriptors

**Scale Invariant Feature Transform (SIFT):** Combination of detector + descriptor. The descriptor uses histogram of gradient. It results finally in a 128-dimensional vector.

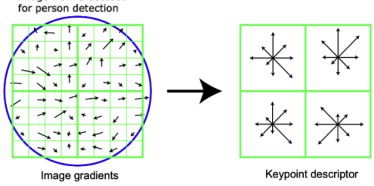


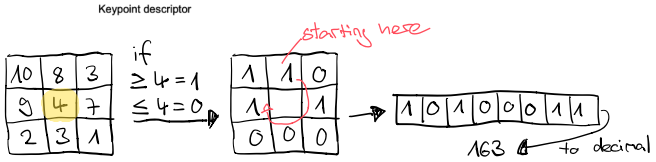
Image gradients

Keypoint descriptor

**Features for semantic segmentation or Texture recognition**

**Local Binary Patterns (LBP):** Analyses the 3x3 neighborhood of a pixel and assigns a binary code by comparing the center pixel with the neighbors. Code converted to integer is used as descriptor. Illumination invariant?

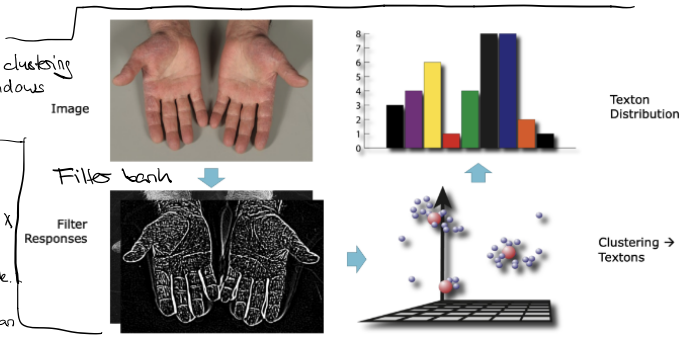
**Extensions:** Rotation Invariance, Greyscale invariance




**Filter Banks:** **RFS Filters:** Edge and line filters at 6 orientations and 3 scales, Gaussian and Laplacian of gaussian

**MR8 Filters:** Maxima of RFS Filters at each scale

**Textons:** Texture descriptor as vector of filter bank outputs. Textons are found by clustering the filter results, called dictionary generation. Similarity between regions/windows by comparison of Texton histograms.

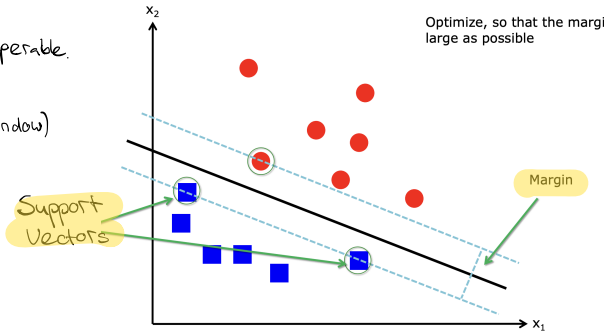



**Gray level co-occurrence matrices (GLCMs):** How often a specific combination of a color value between a pixel and it's neighbors occur. It results in a 256 x 256 matrix (for 256 gray values). Additional features can be calculated from the matrix (entropy, energy, homogeneity, contrast, dissimilarity)

**Classification:** **Nearest Neighbor:** Classify sample according the label of the nearest training example. Only a distance function needed.

**k-Nearest Neighbor:** use the labels of the k nearest neighbors instead. Better set k to an odd number like 3 to avoid a tie in a decision.

**Support Vector Machine (SVM):** Out of all possible linear functions $f(x) = wx + b$ choose the one that results in the largest gap between the categories. Minimize penalty function for wrong samples.




**Nonlinear SVM:** Transform into higher dimensional space, where the data is linearly seperable.

$\phi: x \rightarrow \varphi(x)$

**Sliding Window:** Calculate features for each pixel from the local neighborhood (window). Classify central pixel

**Recipe for semantic segmentation using supervised learning:**

1. Calculate local features from an image patch, for example:
   - Filter responses, Textons, Local Binary Patterns, Histogram of gradients
2. Train classifier on features, for example:
   - k-Nearest neighbors, Decision Trees, SVM

**Metrics:**

$$\text{Precision} = \frac{TP}{TP+FP} =$$

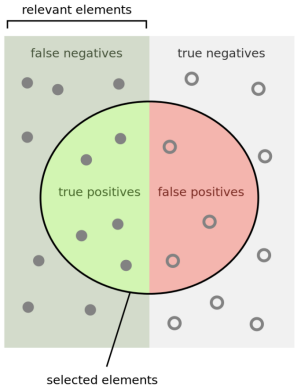
$$\text{Recall} = \frac{TP}{TP+FN} =$$

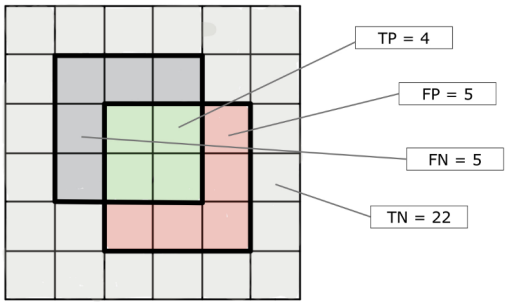
$$\text{Specifity} = \frac{TN}{TN+FP}$$

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN}$$

$$F1 = \frac{2TP}{2TP+FP+FN}$$

$$IoU = \frac{TP}{TP+FP+FN}$$




| | | |
|---|---|---|
| Accuracy | = 0.72 | $= \frac{TP+TN}{TP+TN+FP+FN} = \frac{4+22}{4+5+5+22}$ |
| Precision | = 0.44 | $= \frac{TP}{TP+FP} = \frac{4}{4+5}$ |
| Recall | = 0.44 | $= \frac{TP}{TP+FN} = \frac{4}{4+5}$ |
| F1 Score | = 0.44 | $= \frac{2TP}{2TP+FP+FN} = \frac{2 \cdot 4}{2 \cdot 4+5+5}$ |
| IoU | = 0.28 | $= \frac{TP}{TP+FP+FN} = \frac{4}{4+5+5}$ |

## FCNs with downsampling and upsampling:

**Normal FCN Problem:**
A normal FCN without down- and upsampling would need to be very deep or with large convolutional filters to have sufficiently large receptive field.

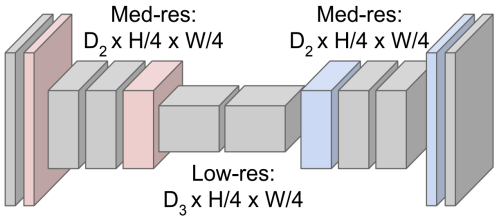
**Sliding Window Problem:** Very inefficient! Not reusing shared features between overlapping patches.

**Downsampling:**
Pooling, strided convolution

**Upsampling:**
Unpooling or strided transpose convolution

Design network as a bunch of convolutional layers, with **downsampling** and **upsampling** inside the network!

Input:
3 x H x W




High-res:
D1 x H/2 x W/2

High-res:
D1 x H/2 x W/2

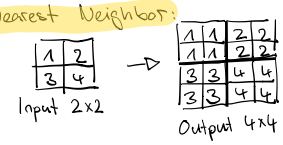
Predictions:
H x W

**Similar to encoder / decoder architecture**

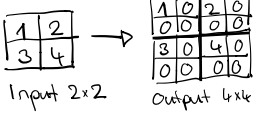
**Upsampling:**

**Unpooling:**

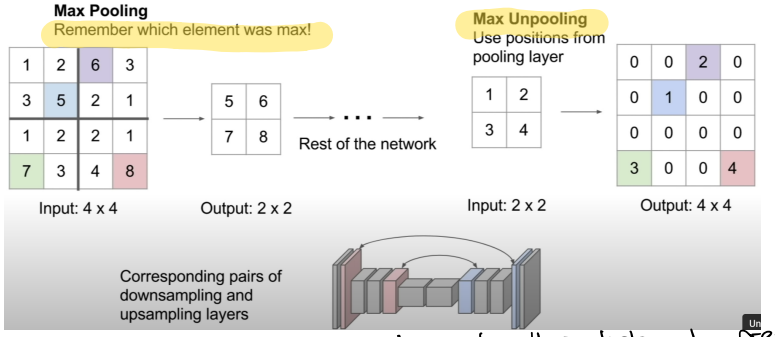
**Nearest Neighbor:**




**"Bed of Nails"**

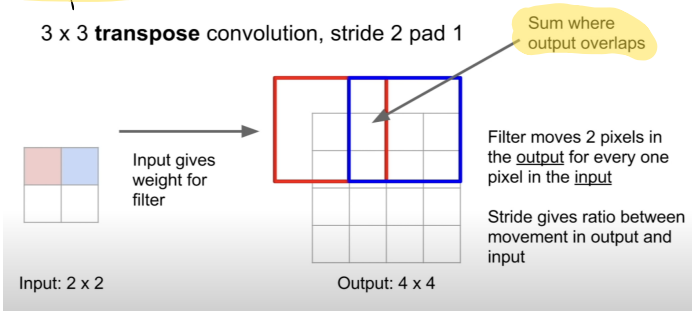


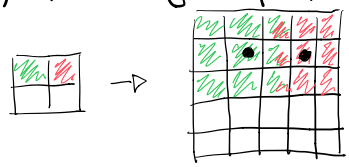

**Max Unpooling:**




Adding this overlapping values together helps to preserve the integrity of the original input pixel values during the upsampling process

**Transposed Convolution:**



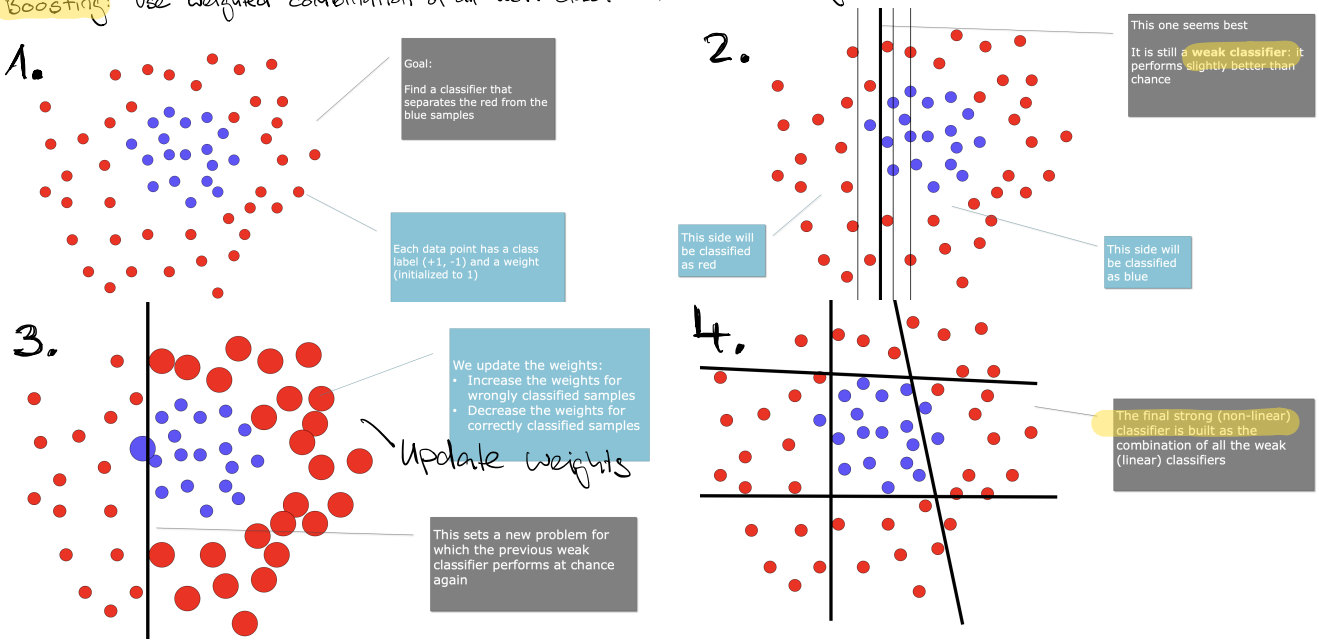
**U-Net Architecture:** Down- and Upsampling with max pooling. Additional skip connections between early and late layers in the network.

**Problems with SegNet FCN:** Loss of resolution with max-pooling and strided convolutions. Large receptive fields of 200 x 200 pixels or more
Idea → Learn on stack (batch) image patches, ...

**Finding and Classifying Objects:** Different challenges like different backgrounds, illumination, deformation, etc.

**Viola Jones Face-Detector:** Combines Boosting (AdaBoost), simple Features and Cascade classifiers

**Boosting:** Use weighted combination of all weak classifiers for the final (strong) classifier. The weak classifiers are just linear functions.




**Haar Features:** Uses simple filters of different shapes and sizes with only -1 or +1 in the filter masks. The features are calculated on sub windows (24 x 24 pixels) (sliding window). All possible combinations: 160'000

**Integral Image:** To achieve a fast calculation of the filter results.

| | | | | |
|---|---|---|---|---|
| 1 | 7 | 4 | 2 | 9 |
| 7 | 2 | 3 | 8 | 2 |
| 1 | 8 | 7 | 9 | 1 |
| 3 | 2 | 3 | 1 | 5 |
| 2 | 9 | 5 | 6 | 6 |

Source Image

| | | | | |
|---|---|---|---|---|
| 1 | 8 | 12 | 14 | 23 |
| 8 | 17 | 24 | 34 | 46 |
| 9 | 26 | 40 | 55 | 71 |
| 12 | 31 | 48 | 68 | 85 |
| 14 | 42 | 64 | 90 | 113 |

Integral Image

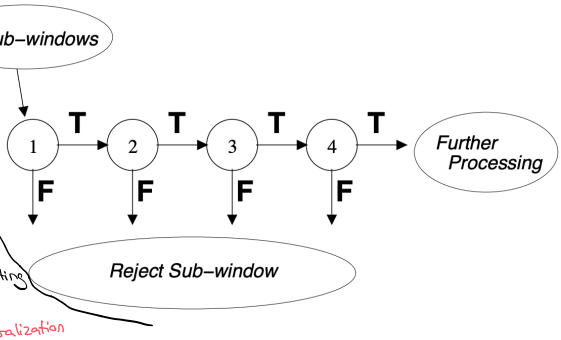
Area wanted

113 - 64 - 71 + 40 = 18

check source image to verify

| | | | | |
|---|---|---|---|---|
| 1 | 8 | 12 | 14 | 23 |
| 8 | 17 | 24 | 34 | 46 |
| 9 | 26 | 40 | 55 | 71 |
| 12 | 31 | 48 | 68 | 85 |
| 14 | 42 | 64 | 90 | 113 |

**Cascaded Classifiers Approach:** In order to be fast the goal is to reject false results early in the process. Each feature acts as a binary classifier in a cascade filter. Each step of the 10-step cascade is trained with AdaBoost using 20 (different) features.

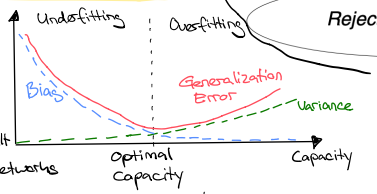



**How to train deep networks:** Deeper Neural Networks: Raw Input → Features → Output

**Training Error:** Error on training set, **Validation Error:** Error on validation set

**Generalization Error:** Gap between the error on the training and validation set

**Regularization:** Limit the capacity of the network by adding a parameter norm penalty to the loss in order to avoid overfitting on training set.




- **$L^2$-Regularization (Ridge):** $\Omega(\Theta) = \frac{1}{2}\|w\|_2^2$
- **Ensemble Methods:** Use several models that are trained separately and then vote on the result
- **Dropout:** Randomly drop a node on a hidden layer or an input value. Training learns sub networks of the original network
- **Early Stopping:** Stop when error on val. set starts to increase again. Training steps becomes a hyperparameter.

**Mini-batch size:** Large batches → better estimates, but non-linearly increased cost. Very small batches underutilize hardware. Memory might limit batch size! in parallel processing

**Stochastic Gradient Descent:**

- Sample minibatch of $m$ examples
- Compute gradient estimate $\hat{g} \leftarrow \frac{1}{m} \nabla_\theta \sum L(f(x^{(i)}; \theta), y^{(i)})$
- Apply update $\theta \leftarrow \theta + \epsilon \hat{g}$

**Adaptive Learning Rate Algorithms:**

- **Adagrad:** Scale lr inversively by squared gradient estimate
- **RMSProp:** Use exponentially weighted moving average for gradient estimation for scaling
- **Adam:** Additionally use **Momentum:** Accumulate average of past gradients

→ Start with SGD until model converges. Tune LR. Use AdaGrad or Adam then to make convergence faster and more robust

**Batch Normalization:** Normalizes input to layers. Usually inserted after Conv and Dense layers before the activation function. Improves Gradient flow. Allows higher LR. Reduces dependence on (correct) initialization

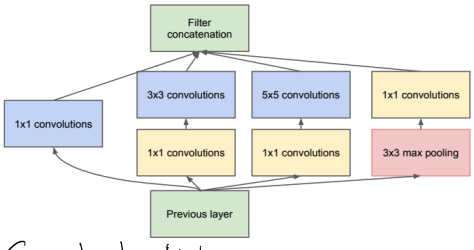
**Activation Functions:** Relu, Leaky Relu (Addresses dying Relu issue for neg. values), Sigmoid (don't use in hidden layers), Softplus

→ **Baby-Sit Learning Process:** Start with simple data and see that model converges, Check loss before and after Regularization, Experiment with LR

**Most important hyperparameters:** Network capacity / Architecture, LR (decay), Regularization weights, Dropout percentage

## AlexNet

[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)




## GoogLeNet

version with dimensionality reduction (bottleneck layers)

**Network Architectures:** for ImageNet challenge

**LeNet (1998):** 2 Conv, 2 Pooling Layers

**AlexNet (2012):** 5 Conv, Max-Pooling Layers, 3 Dense layers at the end, ReLU Activation → 60 Mio. Params

**GoogLeNet - Inception Network:** Bottleneck Layers (1x1 Filters), 22 Layers deep, ca. 100 layer building blocks, see Inception ResNet v2 for 1x1 conv

**VGG Net:** Examine effect of depth of networks, 3x3 Conv Layers only (some with pooling) because stack of 3 3x3 filters has the same receptive field as a 7x7 filter but less parameters and more non-linearity

**ResNet:** Residual connections to learn identity transform. Therefore, able to train very deep models (152 layers)

**Inception ResNet v2:** uses 1x1 convolutions as a dimensionality reduction technique to reduce the number of parameters in the model and improve computational efficiency. (56, 56, 64) → 1x1 conv with 32 filters → (56, 56, 32)

**Region Proposals:** Sliding window approach not suitable for CNN because we would need to apply the CNN to a huge number of different positions and sizes. Therefore, we use region proposals. It finds image regions that are likely to contain an object.

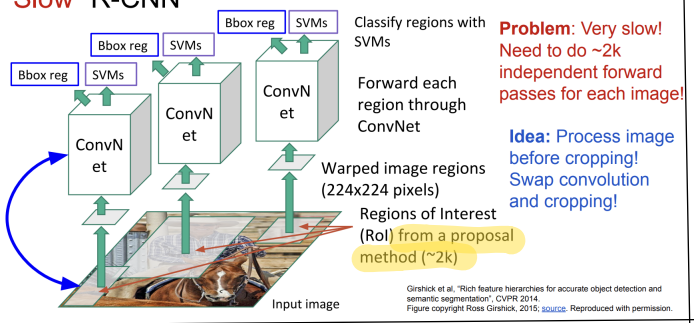
**R-CNN:**

- Extract region proposals
- Compute CNN features
- Classify regions using SVM

Classifier Training: Regions with overlap of 30% is considered a positive example

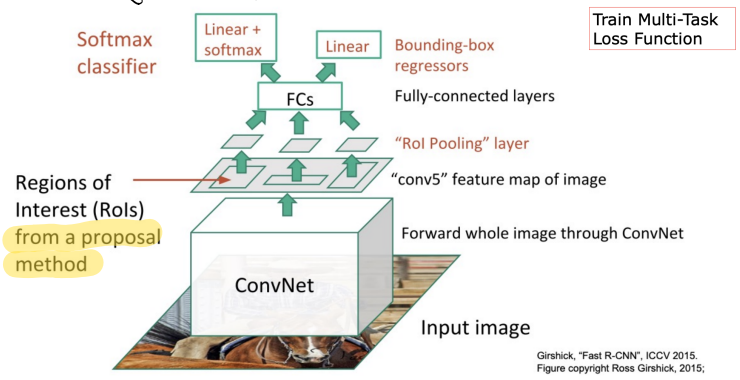
Bounding box position: Treat as regression problem

Predict "corrections" to the RoI: 4 numbers: (dx, dy, dw, dh)




**Fast R-CNN**

Replace SVM of R-CNN with fully connected neural network for classification of object and refind bounding box of object.

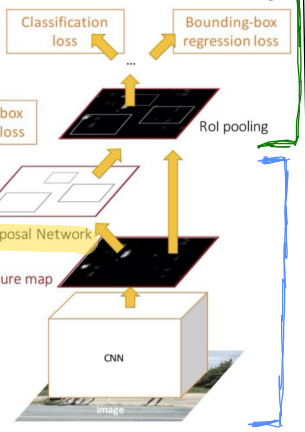



**Faster R-CNN**

Replace proposal region function with fully convolutional network → Region Proposal Network (RPN). Otherwise same as Fast R-CNN

Jointly train 4 losses:

- RPN classification object / no object
- RPN regress box coordinates
- Final classification (object class)
- Final bounding box




Faster R-CNN is a two stage detector

First stage: Run once per image
- Backbone network, - RPN

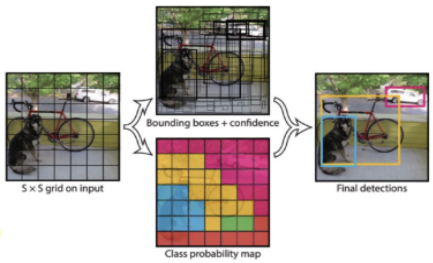
Second stage: Run once per region
- Crop features: RoI pool / align
- Predict object class
- Prediction bbox offset

**Detection without Proposals (Single-Stage Detectors)**

**YOLO** (You only look once)

Approach

1. Divide Image into $S \times S$ grid
2. Each grid cell predicts B bounding boxes with confidence scores (x, z, w, h, score)
3. Each grid cell predicts class probabilities (independent of bounding boxes)
4. Multiply confidence score and probabilities at test time




**SSD** (Single Shot Detector)

Approach

1. Divide Image into $S \times S$ grid
2. Use few standard bounding boxes for each grid cell
3. Predict shape offset and confidence for each object class
4. Faster and more accurate than YOLO (say the authors ;))

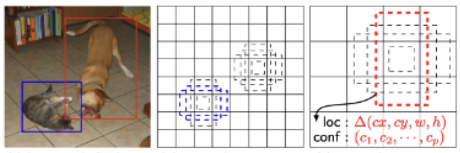
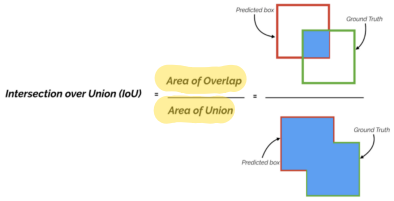



Figure 9.6: SSD

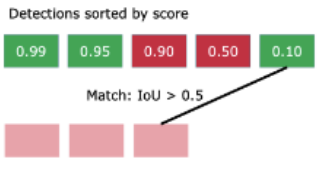
**Assessment of Object Detection:** Object Detection usually measured as mAP @ IoU 0.5

**Intersection over Union (IoU):**

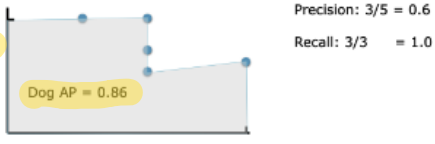



**Mean Average Precision (mAP):**

Mean average precision is the mean of the average precision (AP) of each class. So, calculate AP first for each class. Then take the mean over all classes.




1. Sort Detection by score
2. For each detection (higher scores first)
   (a) If there is a match with a ground truth box with IoU > 0.5, mark it as positive
   (b) Otherwise mark it as negative
   (c) Plot a point on Precision/Recall Curve
3. Average Precision (AP) = Area under PR curve



## Generating Images

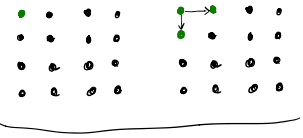
Generative Models
- Explicit density (solve for $p_{model}(x)$)
  - Tractable (Pixel RNN/CNN)
  - Approximate
    - Variational (VAE)
- Implicit density (Learn model that can sample from $p_{model}(x)$)
  - Direct (GAN)

The basic idea is that training samples (image, no labels) are given and then new samples are generated from the same distribution.

**Pixel RNN / CNN**: Model likelihood of image as dependency to previous pixel values: $p(x) = \prod_{i=1}^{n} p(x_i \mid x_1, \dots, x_{i-1})$

Train to maximize the likelihood of the training data.

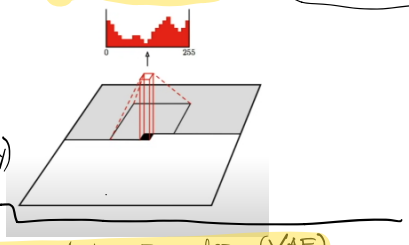
**Generation with RNN**: Generate images starting from a corner. Model dependency on previous pixels using a recurrent neural network (LSTM). Drawback: sequential generation is slow.



etc.

**Generation with CNN**: Still generates image pixels starting from corner.

Dependency on previous pixels now modeled using a CNN over context region. Training is faster than PixelRNN but generation must still proceed sequentially (= still slow).




→ Both are generating good samples with high resolution images (not blurry) but they have a long generation time.

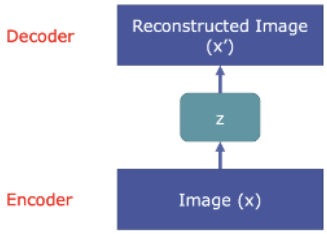
## Auto Encoder

**Decoder:**
*Classic:* Linear with sigmoid
*Now:* Upscaling CNNs

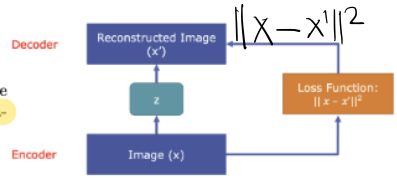
**z smaller than x:** Dimensionality reduction because we want only the important features in z!

**Encoder:**
*Classic:* Linear with non linearity (sigmoid)
*Now:* CNNs with ReLU

1.

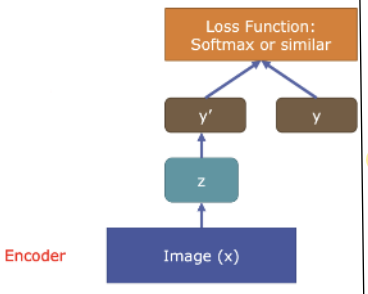



2.




Train so that the data can be reproduced from the variables z. The L2 loss functions does not use labels!

3.

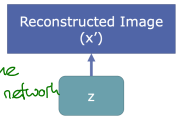



After training we can throw away the decoder. The encoder can now be used to initialize a supervised model. The encoder can be fine-tuned jointly with a classifier.

→ Auto encoders capture the essence of the image properties into variables z.

## Variational Auto Encoders (VAE)

Idea: Sample from the model to generate the data. Assume that training data is generated from some (unobserved) representation z. This representation could be learnd from the steps before.




$p_\theta(x|z^{(i)})$

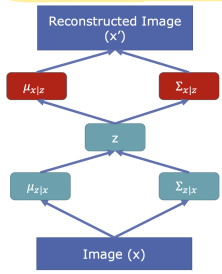
$p_\theta(z)$

Can we describe the distribution of the data likelihood z?

$$p_\theta(x) = \int p_\theta(x)\, p_\theta(x|z)\, dz$$

✓ gaussian prior
✓ described by the decoder neural network
✗ integral is intractable to compute for every z! But we can rewrite it to calculate at least a lower bound.

### VAE: Probability distributions



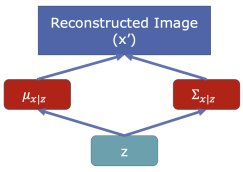

Sample x from $N(\mu_{x|z}, \Sigma_{x|z})$

Decoder CNN

Sample z from $N(\mu_{z|x}, \Sigma_{z|x})$

Encoder CNN

### VAE: Generate Images




Decoder CNN

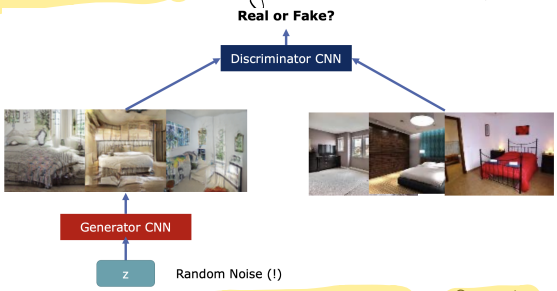
Use Decoder CNN but now sample z from N(0,1)

→ Allows inference of $p(z|x)$ which can be used for other tasks.
Generated samples are blurrier and lower quality than GAN.

---

**Generative Adversial Networks (GAN)**: We can think about it as two person game where one player (generator) tries to make bills indistinguishable from real bills while the other player (discriminator) has the objective to distinguish false bills from real bills.

**Generator CNN**: Produce real looking images that fool the discriminator
**Discriminator CNN**: Distinguish between real and generated images




**GANs: Convolutional Architectures**: Generator is an upsampling network with fractionally-strided convolutions. Discriminator is a convolutional network.

**Architecture guidelines for stable Deep Convolutional GANs**

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers.

### Training GANs

Train both networks jointly in a MiniMax:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

Discriminator for real image — Discriminator for fake image

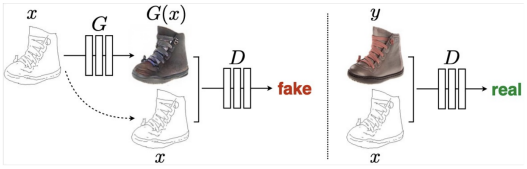
Figure 10.11: GAN training

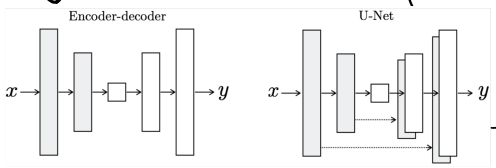
- Discriminator ($D$) wants to maximize objective such that $D(x)$ is close to 1 (real) and $D(G(z))$ is close to 0 (fake)
- Generator ($G$) wants to minimize objective such that $D(G(z))$ is close to 1 (discriminator is fooled into thinking generated $G(z)$ is real)

Alternate between:

1. **Gradient ascent** on discriminator
2. **Gradient descent** on generator
   In practice, optimizing this generator objective does not work well!
   **Instead: Gradient ascent** on generator, but different objective

Conditional GAN: Learn mapping from input image x and noise z to output image y: $G: \{x, z\} \rightarrow y$



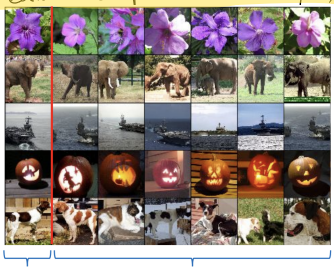

→ U-Net architectures with connections between the layers seem to work better.
→ also use dropout in the intermediate layers

Both the generator G and Discriminator D input the edge map

Visualization approaches:

Visualize filters directly: Works best in first few filters that relate to the input image

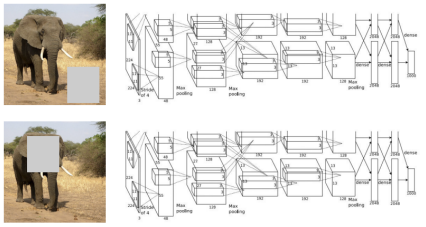
Feature comparison (last layers)



Test Image

Training images closest to 4096 dim feature vector before logits and softmax

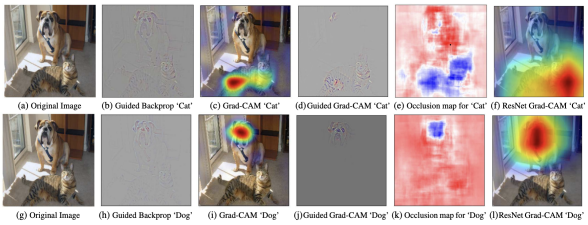
Occlusion-Maps: Mask part of an image. Check how much the predicted probabilities change.



P(elephant) = 0.95

P(elephant) = 0.75

Grad-CAM: Class Activation Maps




Deep Dream: Uses amplification of the activation of some layers in the network as kind of visualization.
Select image and layer in a CNN: 1. Forward computation of activations at chosen layer 2. Set gradient of equal to its activation 3. Backward: Propagate to compute gradient on image 4. Update image

Neural Texture Synthesis: 2 Approaches: • Replicate pixels and patches using a suitable algorithm • Find a model of the texture and generate new texture from this model

Texture Synthesis using CNNs

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

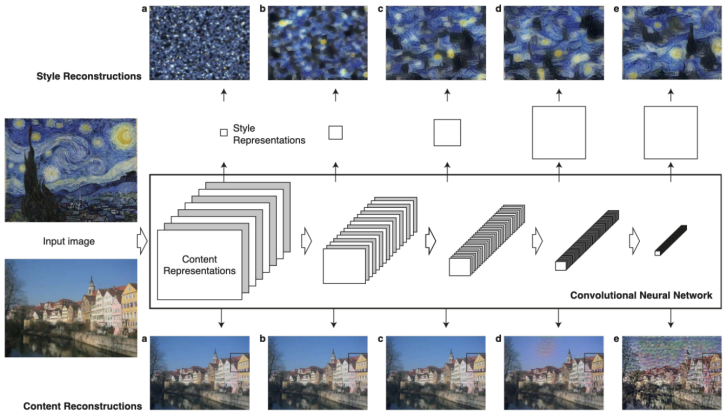
$$G^l_{ij} = \sum_k F^l_{ik} F^l_{jk}$$

**These gram matrices describe the texture model.**

Generating Textures

- Start with noise image
- Use gradient descent to find another image that matches the Gram-matrix representation of some layer(s) of the original image
- Optimize by minimizing the mean-squared distance between the matrices form the original and generated image
- Use standard forward/backward computations of the network!

Neural Style Transfer




Content representation

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

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

- Later layers in the network define the content

Style representation

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

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

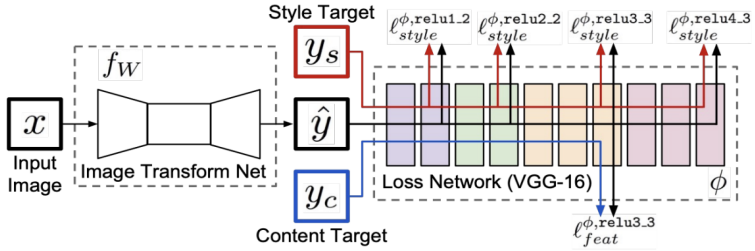
- and from including several layers

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

Fast Style Transfer

Problem: Style transfer is slow as it requires many passes through CNN

Solution: Train neural network to perform style transfer.




→ Are generally faster when only one style is learned and applied. Also the results are better.