Templates and Detection

CSE 4310 – Computer Vision
Vassilis Athitsos
Computer Science and Engineering Department
University of Texas at Arlington

The Detection Problem

- One can define the detection problem in various ways.
- In this class, the detection problem is the problem of finding the bounding boxes where some specific type of object appears in an image.
- Example: face detection.
  - Find the bounding boxes of faces.
- Note: we do not know how many times the type of object we are looking for appears in the image.
  - Can be zero, one, multiple times.
  - For example, for face detection, the input image may have zero, one, or multiple faces.

What is a Template?

- A template is an example of how an object looks.
- In template matching, the goal is to find locations in the image that look similar to the template.
- What example would be appropriate if we are looking for a face?
  - A reasonable starting point is another face.

- So, our first approach for first detection is:
  - Pick an example face image as a template.
  - Look for the image locations that are the most similar to the template.
- Have we fully specified the algorithm we will use?
- No. We need to define what we mean by “most similar”.
  - We need to define how to measure similarity between an image location and a template.

Sum of Squared Differences

- A simple way to measure similarity is using sums of squared differences (SSD) between each image subwindow and the template.
- First, let's define the SSD between two vectors.
- Let v, w be D-dimensional vectors.
  - $v = (v_1, v_2, \ldots, v_D), \qquad w = (w_1, w_2, \ldots, w_D)$
- The sum of squared differences $\mathrm{SSD}(v, w)$ is defined as:

$$\mathrm{SSD}(v, w) = \sum_{d=1}^{D} (v_d - w_d)^2$$

Defining Similarity Using SSD

- To find the image subwindow that is the most similar to the template, we need to measure the SSD between each subwindow and the template.
  - We define a function ssd_scores(image, template).
  - This function returns a result of the same size as the image.
  - result(i,j) is the sum of squared differences between the template and the image subwindow centered at (i,j).
  - Specifically, if the template has R rows and C columns (R, C should be odd):

$$\text{result}(i,j) = \sum_{r=1}^{R}\sum_{c=1}^{C}\left(\text{image}\left(i - \left\lfloor\frac{R}{2}\right\rfloor + r, j - \left\lfloor\frac{C}{2}\right\rfloor + c\right) - \text{template}(r,c)\right)^2$$

  - We can ignore boundary pixels (i,j), where no full $\mathrm{R} \times C$ image subwindow centered at (i,j) can be extracted.
- Good matches correspond to **low** SSD scores.

SSD Pseudocode

- Input arguments: grayscale image, template.
- result = 2D array of the same size as the image.
- Initialize all values of result to -1.
- For every location (i,j) in image.
  - window = image subwindow centered at (i,j), of same size as the template.
  - window_vector = window(:) % vector containing the values of window
  - template_vector = template(:) % vector containing the values of template
  - diff_vector = window_vector – template_vector
  - squared_diffs = diff_vector .* diff_vector
  - ssd_score = sum(squared_diffs)
  - result(i,j) = ssd_score
- Return result

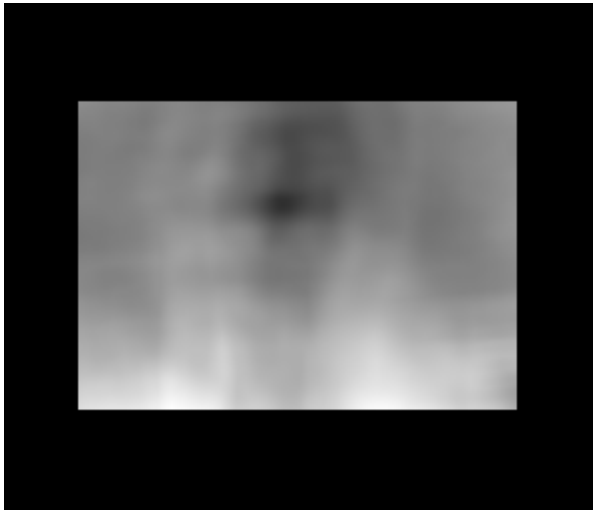
Comments on Pseudocode

- In the pseudocode we just saw, the result array has these values:
  - Boundary pixels (where we could not extract a valid subwindow of the same size as the template) receive values of -1.
  - Interior pixels receive SSD scores.
- The caller function will typically look for the lowest scores in the result, since lowest scores correspond to best matches with the template.
  - The caller function should ignore boundary values when looking for lowest scores. We mark those values with -1, to make them easy to identify.
- Obviously, your code can use different conventions to mark invalid scores in boundary pixels. The pseudocode just provides an example of how to handle this issue.
  - It is important to emphasize, though, that we should always be careful about marking valid and invalid values in our result images.

A Good Result Using SSD Scores

input image

template



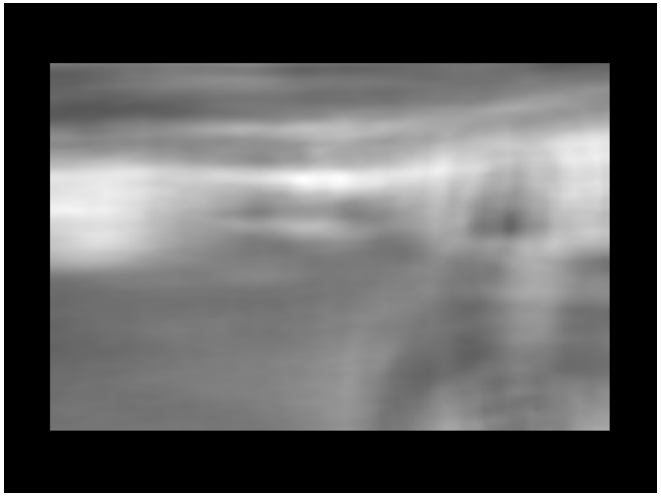
SSD scores

window with lowest score

A Bad Result Using SSD Scores

input image

template



SSD scores

window with lowest score

Some Obvious Shortcomings

- The face detection method we have just outlined is extremely simple.
- There are some obvious shortcomings.
- Can you think of cases where this method would be rather unlikely to succeed?
- Here is a summary of shortcomings.
  - This method works if the face in the image has similar brightness and contrast as the face in the template. We will fix this.
  - This method works if the face in the image has the same size as the face in the template. We will fix this.
  - This method works if the face in the image is rotated in the same way as the face in the template. We will partially fix this.
  - This method works if the face in the image is fully visible. We will NOT fix this (at least not using templates).

Brightness Variations

input image (original version)

window with lowest score

input image (brighter version)

window with lowest score

Brightness Variations and SSD

- Changing the brightness changes the SSD scores.

```
diff_vector = subwindow_ij(:) – template(:)
result(i,j) = sum(diff_vector .* diff_vector)
```

- If the face subwindow is substantially brighter or darker than the template, the SSD score will be high.
- In some cases, we are OK with that.
  - If we are confident that the template and its match in the image should be similarly bright, then we DO want to penalize windows that are much brighter or darker than the template.
- In other cases (like generic face detection) we want to tolerate changes in brightness.

Solution: Normalize For Brightness

- We can subtract from every window its average intensity value.
  - Then, the average intensity value of all windows will be 0.
- For this to work correctly:
  - The template is normalized, by subtracting from it its average intensity.

```
template = template – mean(template(:))
```

  - Each image subwindow is normalized, by subtracting from it its average intensity, right before measuring its SSD with the (normalized) template.

```
subwindow_ij = subwindow_ij – mean(subwindow_ij(:))
diff_vector = subwindow_ij(:) – template(:)
result(i,j) = sum(diff_vector .* diff_vector)
```

- DO NOT simply normalize the entire input image in a single step.
  - We want each individual window to have a mean brightness of 0.

Result

input image (brighter version)

window with lowest SSD score

window with lowest SSD score using brightness normalization

Contrast

- The brightness of a region corresponds to the average of the intensity values in that region.
- The contrast of a region corresponds to the standard deviation of the intensity values in that region.
- High contrast regions are regions whose intensity values have high standard deviation.
  - Many pixels significantly brighter than the average within that region.
  - Many pixels significantly darker than the average within that region.

Examples of High and Low Contrast

- The left image has higher contrast than the right image.
- Both of them have the same average intensity.
- Obviously, lower contrast makes some details harder to see.

Results with High and Low Contrast

- These images show the best matches with the face template, using SSD scores with brightness normalization.
- Contrast makes a difference in the result.

Solution: Normalize for Contrast

- Brightness normalization: we subtract from every window its average intensity value.
  - Then, the average intensity value of all windows will be 0.
- Contrast normalization (done AFTER brightness normalization).
  - We divide each window by the standard deviation of values in that window.
- As was the case for brightness normalization:
  - The template is normalized separately, based on its own mean and standard deviation.
  - Each image subwindow is normalized **separately**, based on its own mean and standard deviation.

Normalization Code

- The template is normalized at the beginning.

```
% normalize template for brightness
template = template – mean(template(:))
% normalize template for contrast
template = template / std(template(:))
```

- Each image subwindow is normalized, right before measuring its SSD with the (normalized) template.

```
% normalize subwindow for brightness
subwindow_ij = subwindow_ij – mean(subwindow_ij(:))
% normalize subwindow for contrast
subwindow_ij = subwindow_ij / std(subwindow_ij(:))
diff_vector = subwindow_ij(:) – template(:)
result(i,j) = sum(diff_vector .* diff_vector)
```

Results After Brightness and Contrast Normalization

Normalized Cross-correlation

- Let v, w be D-dimensional vectors.
  - $v = (v_1, v_2, \ldots, v_D), \qquad w = (w_1, w_2, \ldots, w_D)$
- Let μ_v and μ_w be the means of v and w:
  - $\mu_v = \frac{\sum_{d=1}^{D} v_d}{D}, \qquad \mu_w = \frac{\sum_{d=1}^{D} w_d}{D}$
- Let σ_v and σ_w be the standard deviations of v and w:
  - $\sigma_v = \sqrt{\frac{\sum_{d=1}^{D} (v_d - \mu_v)^2}{D-1}}, \qquad \sigma_w = \sqrt{\frac{\sum_{d=1}^{D} (w_d - \mu_w)^2}{D-1}}$
- The normalized cross-correlation $\mathrm{C}(v, w)$ is defined as:

$$\mathrm{C}(v, w) = \frac{\sum_{d=1}^{D} \big((v_d - \mu_v)(w_d - \mu_w) \big)}{\sigma_v \sigma_w}$$

Normalized Cross-correlation

- The normalized cross-correlation $\mathrm{C}(v, w)$ is defined as:

$$\mathrm{C}(v, w) = \frac{\sum_{d=1}^{D}\big((v_d - \mu_v)(w_d - \mu_w)\big)}{\sigma_v \sigma_w}$$

- The normalized cross correlation can be interpreted as the dot product of two unit vectors:
  - First unit vector: $\frac{v-\mu_v}{\sigma_v}$
  - Second unit vector: $\frac{w-\mu_w}{\sigma_w}$

- Note: σ_v is the norm of $v - \mu_v$, σ_w is the norm of $w - \mu_w$.

Normalized Cross-correlation

- The normalized cross-correlation $\mathrm{C}(v, w)$ is defined as:

$$\mathrm{C}(v, w) = \frac{\sum_{d=1}^{D}\big((v_d - \mu_v)(w_d - \mu_w)\big)}{\sigma_v \sigma_w}$$

- Properties of normalized cross-correlation $\mathrm{C}(v, w)$:
  - Highest possible value: 1
    - $\mathrm{C}(v, v) = 1$
  - Lowest possible value: -1
    - $\mathrm{C}(v, -v) = 1$
  - Higher values indicate higher similarity between v and w.
- If v, w are unit vectors, then $\mathrm{C}(v, w) = 1 - \|v - w\|^2$.
  - The lower the Euclidean distance, the higher the correlation.

Normalized Cross-correlation

- Normalized cross-correlation provides an alternative way to find matches of a template with an image.
  - Instead of looking for lowest SSD score, we look for highest normalized cross-correlation score.
- The detection results we get with normalized cross-correlation are **the same** as the results we get with SSD, if we use brightness and contrast normalization when measuring SSD.
  - When we normalize an image window for brightness and contrast, we convert the window to a unit vector.
  - Highest cross-correlation scores correspond to lowest SSD scores.
- Mathematically, both approaches are equivalent.

Normalized Cross-correlation in Matlab

- In Matlab, there is a built-in function called **normxcorr2**.
- **normxcorr2(template, image)** returns an array of normalized cross-correlation scores between the template and each template-sized subwindow of the image.
- However, the result of **normxcorr2** does not have the same size as the image.
  - It includes extra scores in the boundary.

Normalized Cross-correlation in Matlab

- In the code posted on the course website, we include a **normalized_correlation** function, which serves as a convenient wrapper function for **normxcorr2**.
- **normalized_correlation(image, template)** returns a result of the same size as the image.
  - **result(i,j)** is the normalized cross-correlation score between the template and the template-sized image subwindow centered at pixel **(i,j)**.
  - Boundary pixels (where we could not extract a valid subwindow of the same size as the template) receive values of 0.

Detection at Different Scales

- This is an example we saw before, where the face is detected successfully.

template

Detection at Different Scales

- Here is the result on a 23% smaller version of the image.
- The face is now somewhat smaller than the template.
- Normalized cross-correlation (same as SSD with brightness/contrast normalization) cannot handle that.

template

Detection at Different Scales

- Here is the result on a 23% larger version of the image.
- The face is now somewhat bigger than the template.

template

Detection at Different Scales

- SSD and normalized cross-correlation assume that the object that we want to detect is about as large as the template.
- This is too much of a restriction.
  - Typically we do not know in advance how large or small objects of interest are in an image.
  - A detector should be able to detect objects regardless of how big they appear (unless they appear so small that they cannot be seen clearly).

Scaling the Image

- What can we do to detect the face if we know that the face is about three times larger than the template?

Original image, face about three times larger than in the template.

template

Scaling the Image

- What can we do to detect the face if we know that the face is about three times larger than the template?
- First approach: scale down the image.
  - Scale the image down by a factor of three.
  - Get normalized cross-correlation scores between the scaled down image and the template.
    - In the scaled-down image, the face is about the same size as the template.
  - Scale up the cross-correlation scores by a factor of three.
    - So that scores(i,j) corresponds to the subwindow centered at (i,j) in the original image.
  - Find maximum score location in the scaled-up scores.

Example

Original image, face about three times larger than in the template.

template

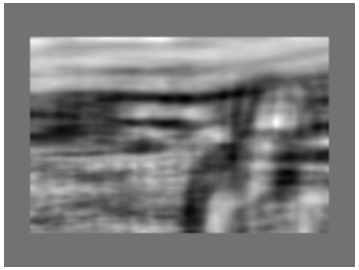
Example

template

Scaled down image,
three time smaller

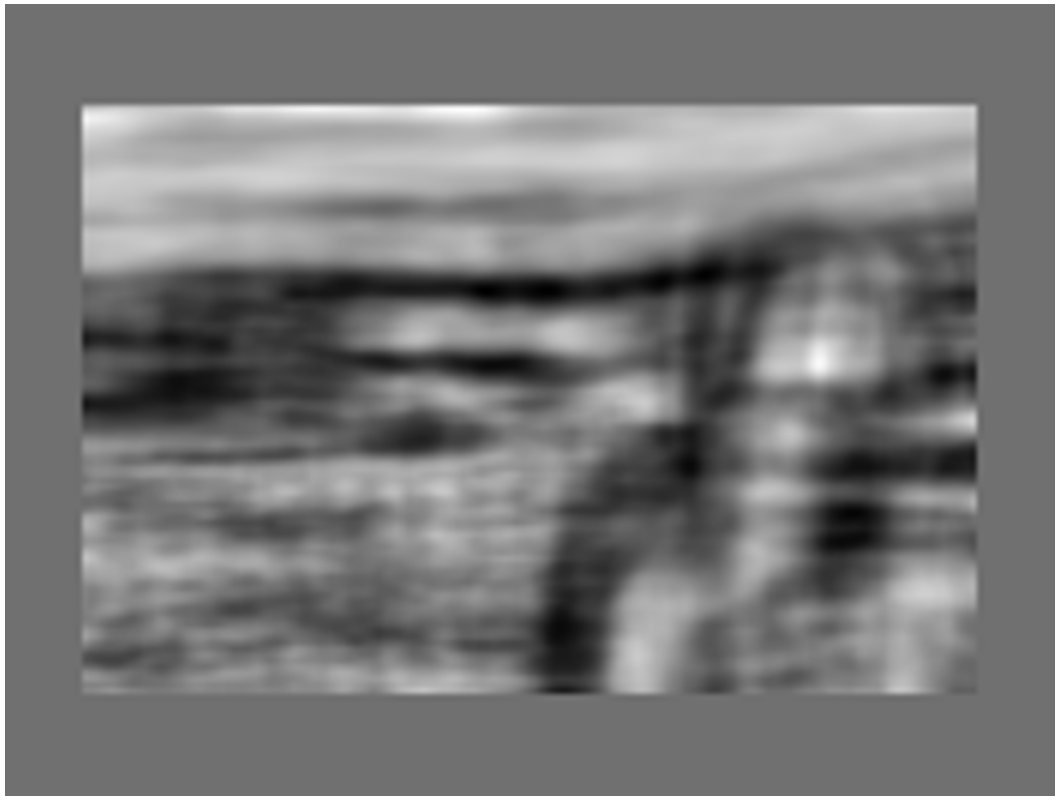
Example

template



Normalized correlation scores between small image and template

Example



template

Normalized cross-correlation scores scaled up three times, so that they match the original image.

Example

Window of the original image corresponding to best score (according to the scaled up scores).

template

Matlab Code

```
scale = 1/3;
scaled_image = imresize(my_image, scale);
scaled_scores = normalized_correlation(scaled_image, template);
scores = imresize(scaled_scores, size(my_image), 'nearest');

% the rest of the code is just useful for showing the result.
max_value = max(scores(:));
[rows, cols] = find(scores == max_value);

y = rows(1);
x = cols(1);

[template_rows, template_cols] = size(template);
face_rows = round(template_rows/scale);
face_cols = round(template_cols/scale);
result = draw_rectangle2(my_image, y, x, face_rows, face_cols);
imshow(result, [0 255]);
```

Matlab Code

```
scale = 1/3;
scaled_image = imresize(my_image, scale);
scaled_scores = normalized_correlation(scaled_image, template);
scores = imresize(scaled_scores, size(my_image), 'nearest');
```

- These are the important lines that show how to do template-based face detection if the face is three times larger in the image than on the template.
- Note the three main steps:
  - Scale down image.
  - Do normalized correlation of scaled-down image with template.
  - Scale up scores, to match the size of the original image.

Matlab Code

```
scale = 1/3;
scaled_image = imresize(my_image, scale);
scaled_scores = normalized_correlation(scaled_image, template);
scores = imresize(scaled_scores, size(my_image), 'nearest');
```

- Notes on scaling up the scores using **imresize**:
  - We use the 'nearest' option, which does not use interpolation, but simply uses the value of the nearest corresponding pixel from the original image.
  - By passing **size(my_image)** as second argument to **imresize**, we ensure that the result will have the same number of rows and columns as the input image.

Alternative: Scaling the Template

- What can we do to detect the face if we know that the face is about three times larger than the template?
- We saw before that we can scale the image down by a factor of three.
- Alternatively???

Alternative: Scaling the Template

- What can we do to detect the face if we know that the face is about three times larger than the template?
- We saw before that we can scale the image down by a factor of three.
- Alternatively, we can:
  - scale up the template by a factor of three.
  - Get normalized cross-correlation scores between the image and the scaled-up template.
    - The face is about the same size as the scaled-up template.
  - Find maximum score location in the scores.
    - **No scaling of the scores is needed, since normalized correlation scores were obtained using the original image.**

Example

Original image, face about three times larger than in the template.

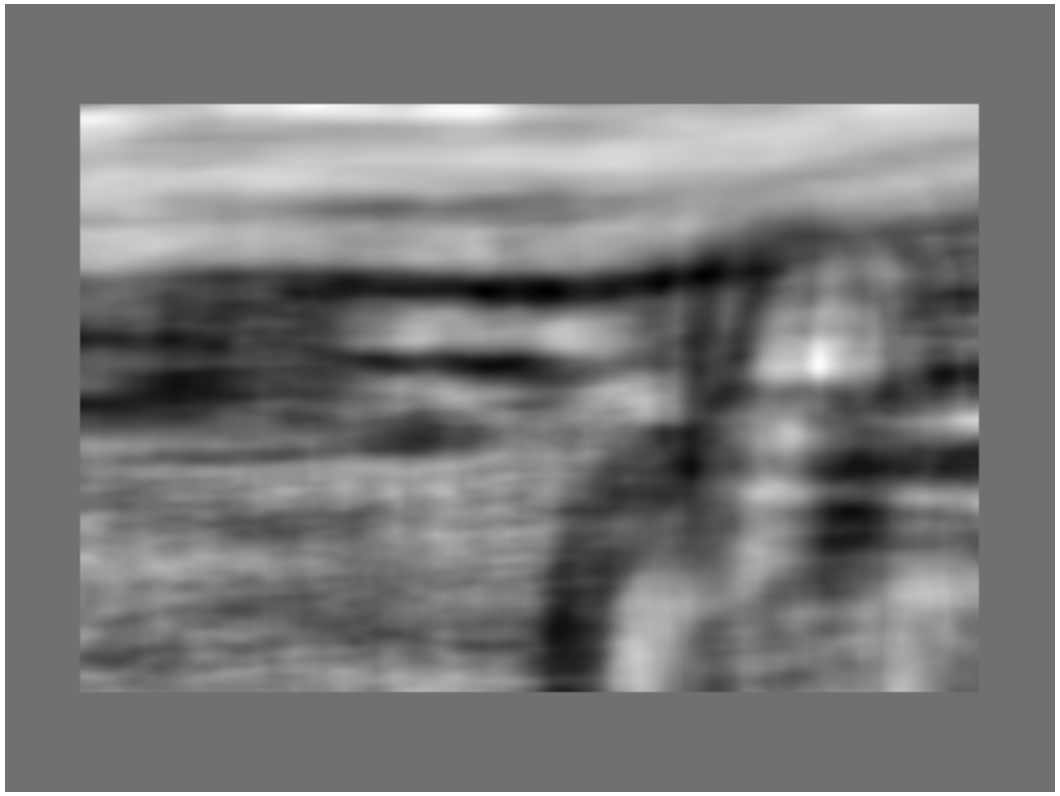
template

Example

Original image, face about three times larger than in the template.

scaled-up template

Example



Normalized cross-correlation scores between original image and scaled-up template.

scaled-up template

Example

Window of the original image corresponding to best score (according to the scaled up scores).

scaled-up template

Matlab Code

```
scale = 1/3;
scaled_template = imresize(template, 1/scale);
scores = normalized_correlation(my_image, scaled_template);

% the rest of the code is just useful for showing the result.
max_value = max(scores(:));
[rows, cols] = find(scores == max_value);

y = rows(1);
x = cols(1);

[template_rows, template_cols] = size(template);
face_rows = round(template_rows / scale);
face_cols = round(template_cols / scale);
result = draw_rectangle2(my_image, y, x, face_rows, face_cols);
imshow(result, [0 255]);
```

Matlab Code

```
scale = 1/3;
scaled_template = imresize(template, 1/scale);
scores = normalized_correlation(my_image, scaled_template);
```

- These are the important lines that show how to do template-based face detection if the face is three times larger in the image than on the template.
- Note the two main steps:
  - Scale up template.
  - Do normalized correlation of image with scaled-up template.

Multiscale Search

- So, when we know the size of the face (or, more generally, the size of the object to be detected), we can scale the image or the template so that the object in the image matches the size of the template.
- However, typically we do NOT know the size of the object(s) we want to detect.
- What do we do then?
- We search at multiple scales.
- For each pixel (i,j), we remember:
  - The max normalized correlation score it got over all scales.
  - The scale that produced that max score.

Multiscale Search: Matlab Code

```
function ...
[result, max_scales] = multiscale_correlation(image, template, scales)

result = ones(size(image)) * -10;
max_scales = zeros(size(image));

for scale = scales;
    % for efficiency, we either downsize the image, or the template,
    % depending on the current scale
    if scale >= 1
        scaled_image = imresize(image, 1/scale, 'bilinear');
        temp_result = normalized_correlation(scaled_image, template);
        temp_result = imresize(temp_result, size(image), 'nearest');
    else
        scaled_image = image;
        scaled_template = imresize(template, scale, 'bilinear');
        temp_result = normalized_correlation(image, scaled_template);
    end

    higher_maxes = (temp_result > result);
    max_scales(higher_maxes) = scale;
    result(higher_maxes) = temp_result(higher_maxes);
end
```

Multiscale Search: Matlab Code

```
function ...
[result, max_scales] = multiscale_correlation(image, template, scales)

result = ones(size(image)) * -10;
max_scales = zeros(size(image));
```

- We will walk through the implementation of multiscale_correlation, which implements multiscale template search using normalized correlation.
- Arguments:
  - Image where we want to detect objects.
  - The template that we want to use.
  - **scales** is an array of scales.
    - E.g., **scales = [0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5]**.

Multiscale Search: Matlab Code

```
function ...
[result, max_scales] = multiscale_correlation(image, template, scales)

result = ones(size(image)) * -10;
max_scales = zeros(size(image));
```

- **scales** is an array of scales.
  - E.g., **scales = [0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, 1.3, 1.4, 1.5]**.
  - Or, equivalently, **scales = 0.5:0.1:1.5**.
- In the above example, the **scales** argument only covers scales between 0.5 and 1.5.
  - Assumes that faces will be between 50% and 150% of the size of the face template.
- Usually this range is too restrictive, we want a larger range.

Multiscale Search: Matlab Code

```
function ...
[result, max_scales] = multiscale_correlation(image, template, scales)

result = ones(size(image)) * -10;
max_scales = zeros(size(image));
```

The recommended practice for **scales** is to:

- Cover a wide range of scales.
  - "Wide range" is not specific enough, but covering scales from 0.1 to 10 is usually a better choice than covering scales from 0.5 to 1.5.
- Use a sequence of scales that increases geometrically, not arithmetically.
    - It makes sense to go from scale 0.1 to scale 0.11, because the increase from one scale to the next is about 10%. If anything, this type of gap may be too large in some cases.
    - However, going from scale 10.00 to scale 10.01 is usually too conservative, as the increase is only 0.1%, which is needlessly small.

Multiscale Search: Matlab Code

- This sample code generates a "reasonable" value for the **scales** argument, given a smallest scale, largest scale, and ratio of 1.1 between two consecutive values.
  - Remember, though, what is "reasonable" depends on the application and the data.

```
low_scale = 0.1;
high_scale = 10;
step_ratio = 1.1;
ratio = high_scale / low_scale;
number_of_steps = ceil(log(ratio)/log(step_ratio));
scales = low_scale * (step_ratio .^ (0:number_of_steps));
```

- The above code generates an array **scales** of size 50.

```
scales = [0.10, 0.11, 0.121, 0.133, 0.146, …,
          7.29, 8.02, 8.82,  9.70, 10.67]
```

Multiscale Search: Matlab Code

```
function ...
[result, max_scales] = multiscale_correlation(image, template, scales)

result = ones(size(image)) * -10;
max_scales = zeros(size(image));
```

- Back to the **`multiscale_correlation`** function.
- We just talked about the arguments.
- Next: we discuss the return values.

Multiscale Search: Matlab Code

```
function ...
[result, max_scales] = multiscale_correlation(image, template, scales)

result = ones(size(image)) * -10;
max_scales = zeros(size(image));
```

- Return values.
  - **result(i,j)** will store the maximum normalized correlation score found for location **(i,j)** over all scales that we try.
  - **max_scales(i,j)** will store the scale that produced that maximum normalized correlation score for location **(i,j)**.
- What value is each **result(i,j)** initialized to, and why?

Multiscale Search: Matlab Code

```
function ...
[result, max_scales] = multiscale_correlation(image, template, scales)

result = ones(size(image)) * -10;
max_scales = zeros(size(image));
```

- Return values.
  - **result(i,j)** will store the maximum normalized correlation score found for location **(i,j)** over all scales that we try.
  - **max_scales(i,j)** will store the scale that produced that maximum normalized correlation score for location **(i,j)**.
- What value is each **result(i,j)** initialized to, and why?
  - **result(i,j)** is initialized to -10.
  - Since **result(i,j)** will store a maximum normalized correlation value, whose range is between -1 and 1, it can be initialized to any value less than -1.

Multiscale Search: Matlab Code

```
...
for scale = scales;
    if scale >= 1
        scaled_image = imresize(image, 1/scale, 'bilinear');
        temp_result = normalized_correlation(scaled_image, template);
        temp_result = imresize(temp_result, size(image), 'nearest');
    else
        scaled_image = image;
        scaled_template = imresize(template, scale, 'bilinear');
        temp_result = normalized_correlation(image, scaled_template);
    end
...
```

- Note how we do the scaling.
  - For scales greater than 1 (meaning that the object is larger than the template), we scale down the image, instead of scaling up the template.
  - For scales smaller than 1 (meaning that the object is smaller than the template), we scale down the template, instead of scaling up the image.
- We prefer scaling down over scaling up, for efficiency.

Multiscale Search: Matlab Code

```
...
    higher_maxes = (temp_result > result);
    max_scales(higher_maxes) = scale;
    result(higher_maxes) = temp_result(higher_maxes);
...
```

- **temp_result:** normalized correlation scores for current **scale**.
- **result(i,j):** maximum score found so far for **(i,j)**.
- **max_scales(i,j):** the scale that gave the maximum score for **(i,j)**.
- **higher_maxes(i,j)** is a boolean, indicating if **temp_result(i,j)** (the score at the current scale) is larger than **result(i,j)** (the best score found before the current scale).
  - If **higher_maxes(i,j)** is true, we must update **result(i,j)** and **max_scales(i,j)**.
- The next two lines update the values in the locations of **max_scales** and **result** that need to be updated.

Using Multiscale Search

- This code calls **multiscale_correlation** and draws the bounding box of the best matching location (at the right scale).

```
scales = make_scales_array(0.25, 4, 1.1);

[scores, scales] = multiscale_correlation(photo, template,
scales);
max_score = max(scores(:));
[rows, cols] = find(scores == max_score);
y = rows(1);
x = cols(1);
scale = scales(y, x);
[template_rows, template_cols] = size(template);
face_rows = round(template_rows * scale);
face_cols = round(template_cols * scale);

result = draw_rectangle2(photo, y, x, face_rows, face_cols);
imshow(result, [0 255]);
```

Some Results of Multiscale Search

scales = make_scales_array(0.2, 5, 1.1);
scale of highest score: 0.354

Some Results of Multiscale Search

scales = make_scales_array(0.2, 5, 1.1);
scale of highest score: 2.17

Some Results of Multiscale Search

scales = make_scales_array(0.2, 5, 1.1);
scale of highest score: 0.22

Obviously, this is the wrong result.
Searching over multiple scales, there are more potential false matches.

Rotations

- In our previous examples, faces had a very specific orientation:
  - The front of the face was fully visible.
  - The orientation of the face was upright.
- The orientation of the face can vary in two different ways:
  - In-plane rotations.
  - Out-of-plane rotations.
- In-plane rotations do not change what is visible, they simply change the orientation.

In-Plane Rotations

- In-plane rotations do not change what is visible, they simply change the orientation of the visible part.
- In-plane rotations of faces facing the camera result to faces still facing the camera.
  - The front of the face is still fully visible.
  - However, the face is not oriented upright anymore.

Out-of-Plane Rotations

- In out-of-plane rotations, the object rotates in a way that changes what is visible.
- If we start with a face whose front is visible, an out-of-plane rotation results to part of the front (or the entire front) not being visible.
  - Other parts become visible, like side/top/back of head.



Handling Rotations

- Handling in-plane rotations can be done similar to the way we handled multiple scales:
  - Search at multiple rotations, keep track at each location of the rotation that gives the highest score.
  - We can rotate the image, or we can rotate the template.
  - In Matlab, **imrotate** can be used to do the rotations.
- However, we must be careful of some practical details:
  - If we rotate the image, and get normalized cross-correlation scores, we should make sure we know what original image locations those scores correspond to.
  - If we rotate the template, we may lose parts of it, or introduce new parts, that may change the correlation scores.

Handling Rotations

- Out-of-plane rotations cannot be handled, unless we have templates matching those rotations.
  - Note that we did not need new templates to handle different scales or different in-plane rotations.
- For example, to detect faces seen at profile orientation, we should use templates corresponding to profile orientation as well.
- In this course, we will not worry about handling out-of-plane rotations.

Number of Detection Results

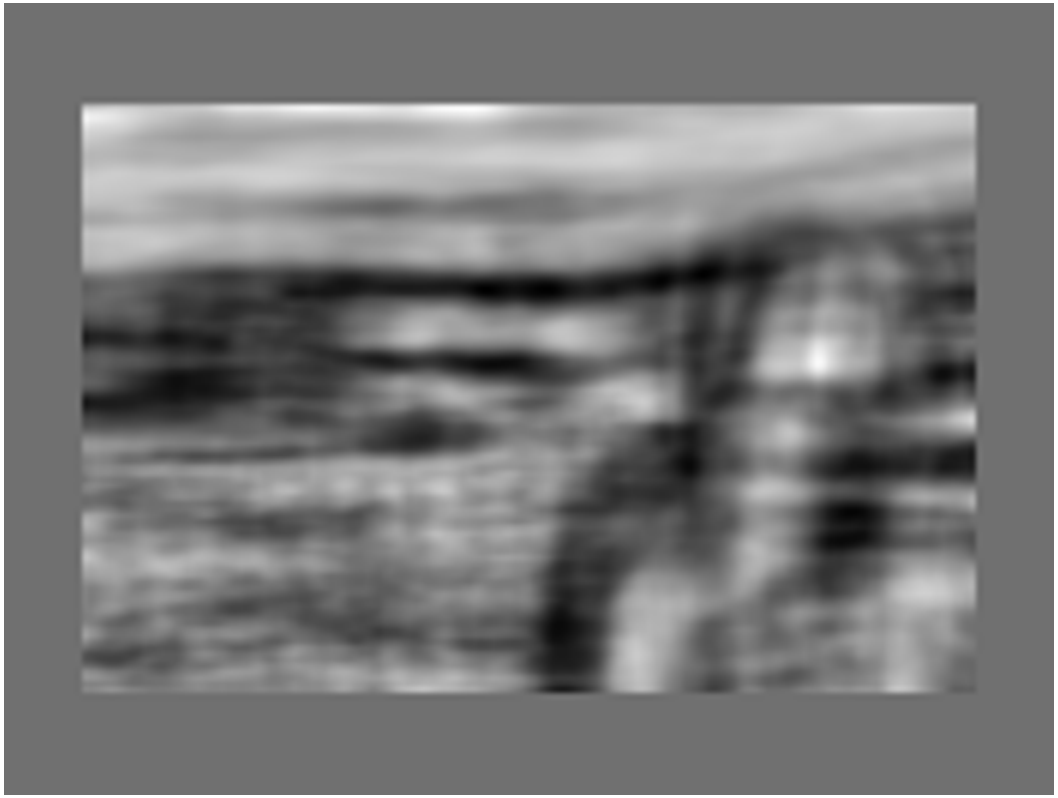
- In our examples so far, we have computed scores at all image locations (and possibly at multiple scales/orientations), and then shown the best score.
- However, we typically do not know how many objects we should detect.
- We need an automatic way to decide the number of detections.
- Standard approach:
  - Set a threshold on the score.
  - Any score above the threshold qualifies as a detection.
    - With one exception…

Non-Maxima Suppression

- Here is an example result that we have seen before.

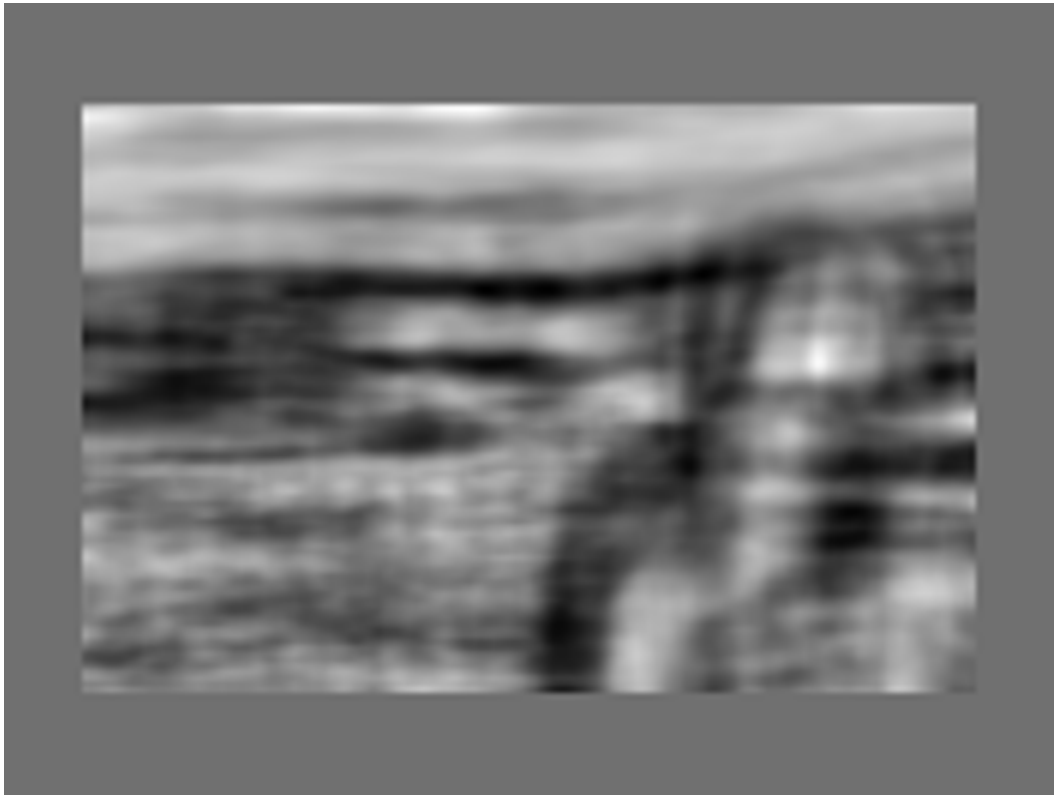
Non-Maxima Suppression

- And here is the normalized cross-correlation result on that image.
  - Where do you think the second highest score is?



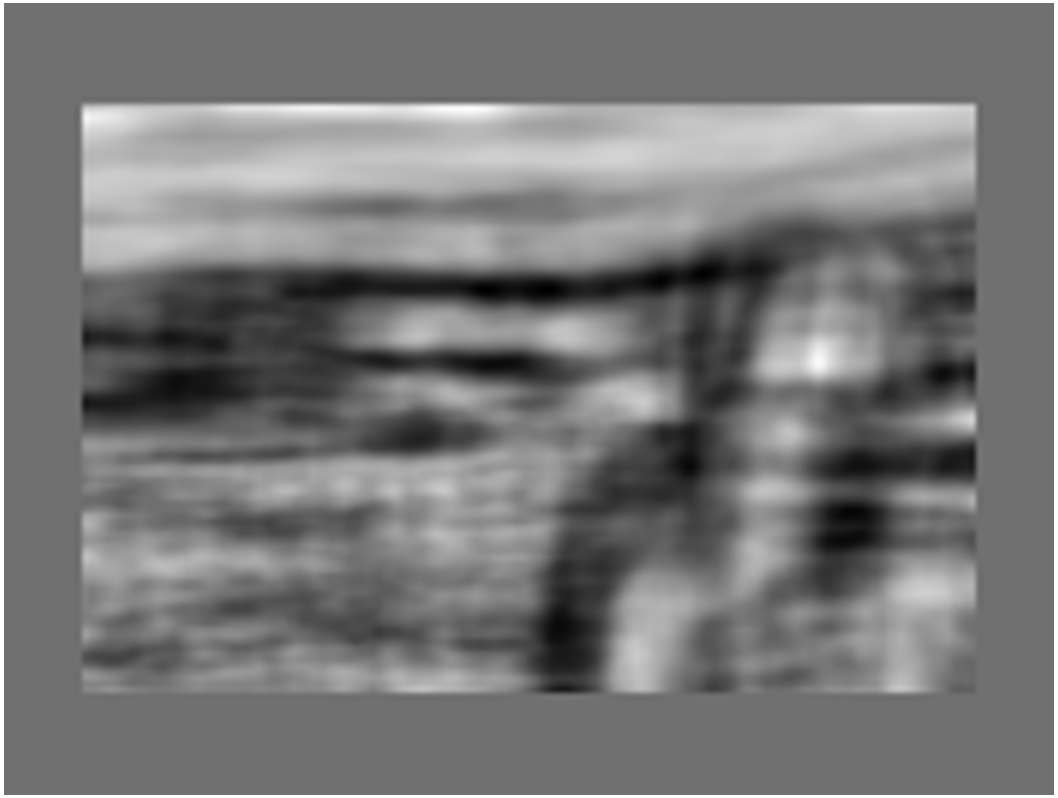
Non-Maxima Suppression

- And here is the normalized cross-correlation result on that image.
  - Where do you think the second highest score is?
  - Right next to the highest score.



Non-Maxima Suppression

- If we want to produce multiple (and meaningful) detection results, we need to suppress results that are almost identical to other results.
  - We call this operation **non-maxima suppression.**



Non-Maxima Suppression

- Returning multiple results while performing non-maxima suppression of detection results is pretty simple.
- Let T be a detection threshold.
- Pseudocode:
  - Let S = normalized cross-correlation scores (optionally at multiple scales, rotations, etc).
  - While (true)
    - Let s be max value in S, and let (i,j) be location of s in S.
    - If s < T, break.
    - Add s to detection locations.
    - Zero out an area of S centered at (i,j). How big this area should be is an implementation choice. For example, it can be the size of the template.

Using Average Images as Templates

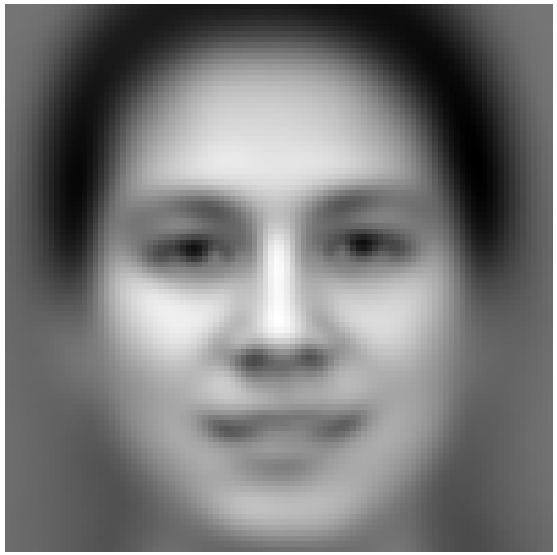
- In our examples, we used a face image as a template, to detect other faces.
- However, using a specific face may be problematic.
  - Some faces may work better than others.
- A common approach is to use an **average image** as a template.
- An average image is (as the name indicates) the average of multiple images.

Computing an Average Face

- We need *aligned* face images:
- In this case, *aligned* means:
  - same size
  - significant features (eyes, nose, mouth), to the degree possible, are in similar pixel locations.

an example set of aligned face images

Result: Average Face



Cropping

- It may be beneficial to crop a template, so that we keep the parts of it that are most likely to be present in the image:
  - Exclude background.
  - Exclude forehead (highly variable appearance, due to hair).
  - Exclude lower chin.

average face

face template

Measuring Accuracy

- Measuring accuracy is an important part of the daily routine in computer vision work.
- We implement various methods, and variations of those methods, and we want to see what works best.
- However, measuring accuracy is not a trivial process.
- Different computer vision tasks (detection, recognition, tracking, segmentation) have their own conventions, that we will learn when we study each task.
- What should be emphasized from the beginning:
  - Numbers are meaningless, without a clear explanation of what is being measured, how it is being measured, and on what data it is being measured.
  - For example, a statement that system X is “99% accurate” does not provide any useful information, in the absence of such specifications.

Some Standard Steps

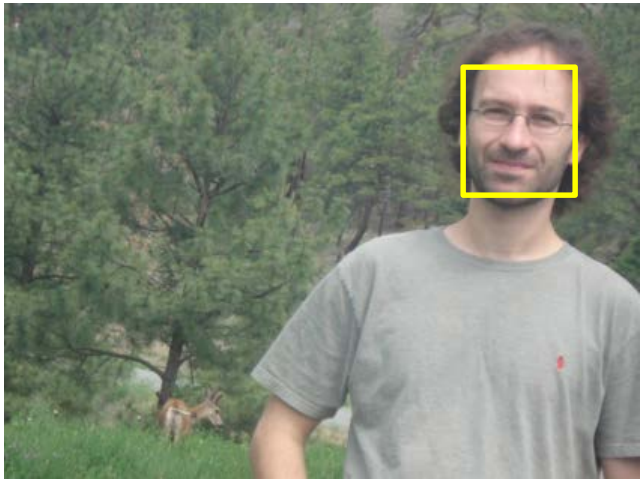
- In measuring accuracy, we need to specify a **test set**.
  - Accuracy will be measured on images from that set.
- For that set, we need to know the "correct" answers.
  - So, in addition to images, the test set needs to contain those answers.
- These "correct" answers are called **ground truth**.
- Then, the best possible result for the system we are evaluating would be to produce answers identical to the ground truth.
- In the typical case where the system's answers are not identical to the ground truth, we need to define **quantitative measures of accuracy**, that measure how close the results are to the ground truth.

Measuring Detection Accuracy

- Detection is a good task for introducing such concepts.
- Measuring detection accuracy is more complicated than measuring other types of accuracy.

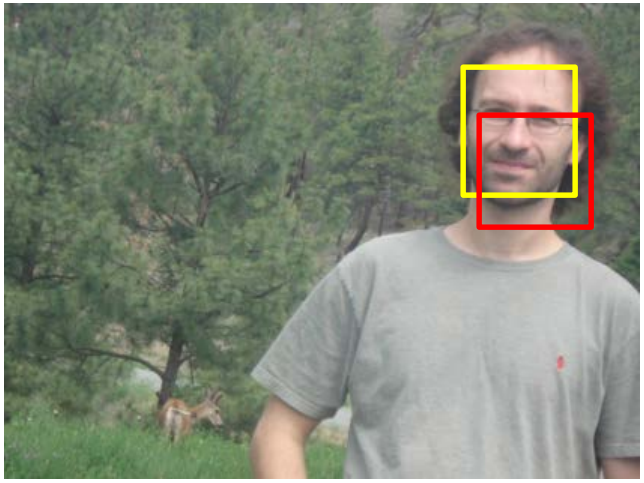
Ground Truth Annotations

- To measure detection accuracy, we need to use some dataset of images.
- For every image in that dataset, we need **annotations** that specify:
  - How many objects of interest are present.
  - The bounding box of every object of interest.
- Bounding box placement is not an exact science.
  - Different people will slightly disagree in where exactly the corners of the bounding box should be.
- However, establishing some clear guidelines is useful, to maintain consinstency as much as possible.
  - Does the box go down to the bottom of the chin? Up to the top of the forehead?



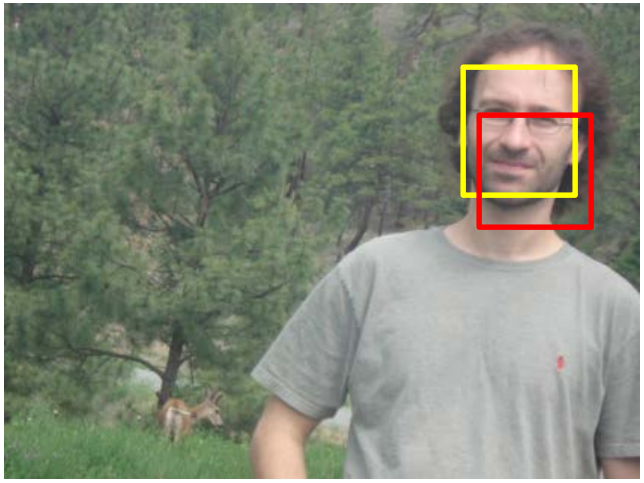
Is a Detection Correct?

- In the image below:
  - Suppose that the yellow box is the ground truth annotation.
  - Suppose that the red box is the detection result.
  - Is that result correct or not?



Is a Detection Correct?

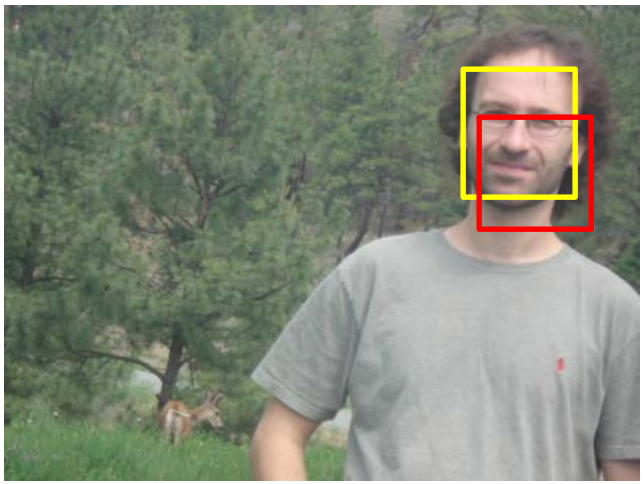
- In the image below:
  - Suppose that the yellow box is the ground truth annotation.
  - Suppose that the red box is the detection result.
  - Is that result correct or not?
- It is important to have specific rules, so that evaluation of correctness can be done automatically.
- Standard approach: measure the **intersection over union** (IoU) score between the detection and the ground truth.



Intersection Over Union

- Intersection over Union (IoU) is a score that measures the overlap between two regions A and B.
- In our case:
  - The first region is the ground truth location of the object of interest.
  - The second region is a detected bounding box.
- IoU(A,B) is defined as this ratio:

$$\frac{\#\text{ of pixels in } A \cap B}{\#\text{ of pixels in } A \cup B}$$



- Remember: $A \cap B$ is intersection, $A \cup B$ is union.

Computing Intersection Over Union

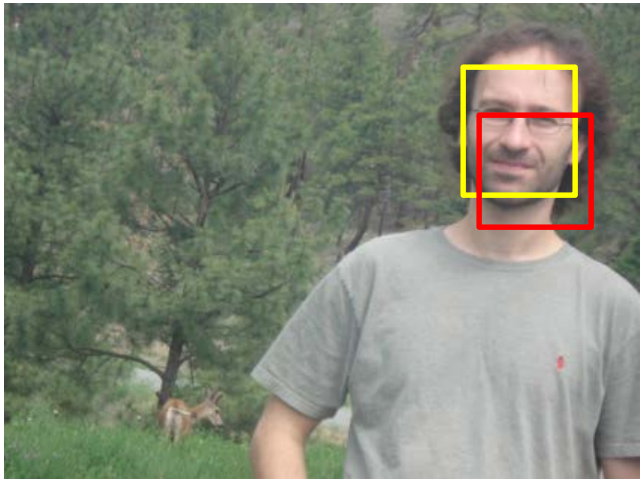
- IoU(A,B) is defined as this ratio:

$$\frac{\#\text{ of pixels in } A \cap B}{\#\text{ of pixels in } A \cup B}$$

- In our example:
  - $A \cap B$ is shown by the green rectangle.
  - $A \cup B$ is the union of the yellow region, the red region, and the green rectangle.

Intersection Over Union

- To determine if a detection result B is correct, we simply check if there is a ground truth location A such that IoU(A,B) > T, where T is some pre-specified threshold.
- Obviously, the choice of T can make a big difference.
  - We typically use several values of T, and we report performance for each T.

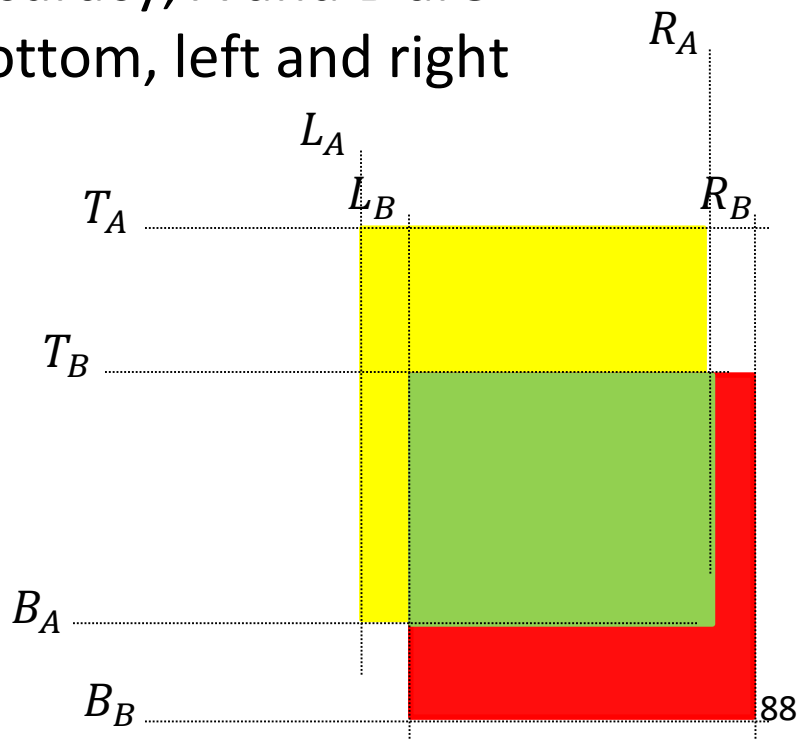


Computing Intersection Over Union

- IoU(A,B) is defined as this ratio:

$$\frac{\#\text{ of pixels in } A \cap B}{\#\text{ of pixels in } A \cup B}$$

- Typically, in evaluating detection accuracy, A and B are rectangles, specified by their top, bottom, left and right coordinates.
  - A is specified as (T_A, B_A, L_A, R_A).
  - B is specified as (T_B, B_B, L_{BA}, R_B).
  - T_A, T_B, B_A, B_B are row numbers.
  - L_A, L_B, R_A, R_B are column numbers.

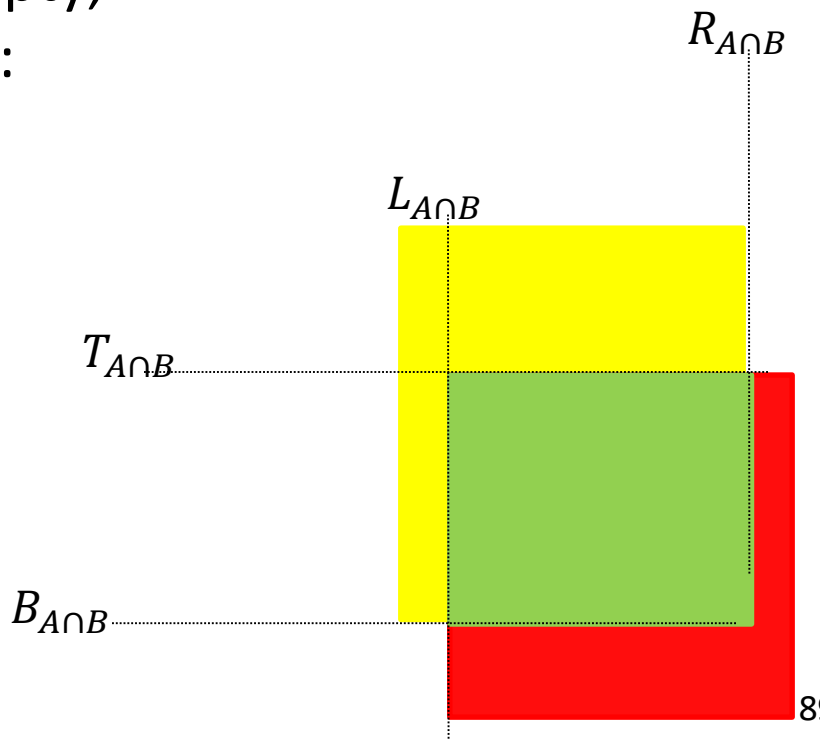


Computing Intersection Over Union

- IoU(A,B) is defined as this ratio:

$$\frac{\#\text{ of pixels in } A \cap B}{\#\text{ of pixels in } A \cup B}$$

- The rectangle $A \cap B$, if it is not empty, is specified by these four numbers:
  - $T_{A\cap B} = \max\{T_A, T_B\}$
  - $B_{A\cap B} = \min\{B_A, B_B\}$
  - $L_{A\cap B} = \max\{L_A, L_B\}$
  - $R_{A\cap B} = \min\{R_A, R_B\}$

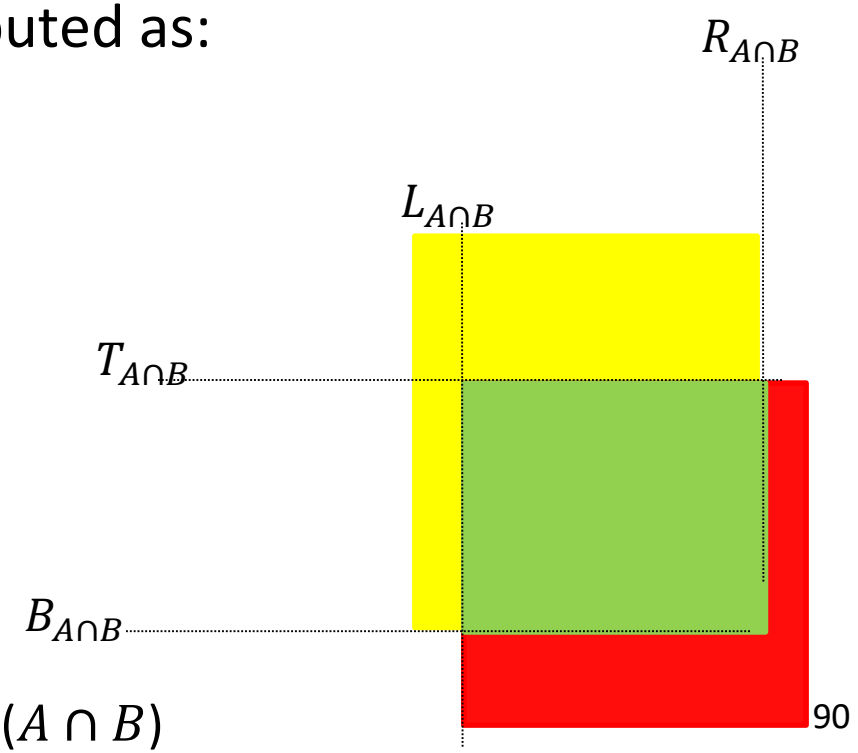


Computing Intersection Over Union

- IoU(A,B) is defined as this ratio:

$$\frac{\#\text{ of pixels in } A \cap B}{\#\text{ of pixels in } A \cup B}$$

- The areas of A, B, $A \cap B$ are computed as:
  - Height(A) = $B_A - T_A + 1$
  - Width(A) = $R_A - L_A + 1$
  - Area(A) = Height(A)* Width(A)
  - Height(B) = $B_B - T_B + 1$
  - Width(B) = $R_B - L_B + 1$
  - Area(B) = Height(B)* Width(B)
  - Height($A \cap B$) = $B_{A\cap B} - T_{A\cap B} + 1$
  - Width($A \cap B$) = $R_{A\cap B} - L_{A\cap B} + 1$
  - Area($A \cap B$) = Height($A \cap B$)* Width($A \cap B$)



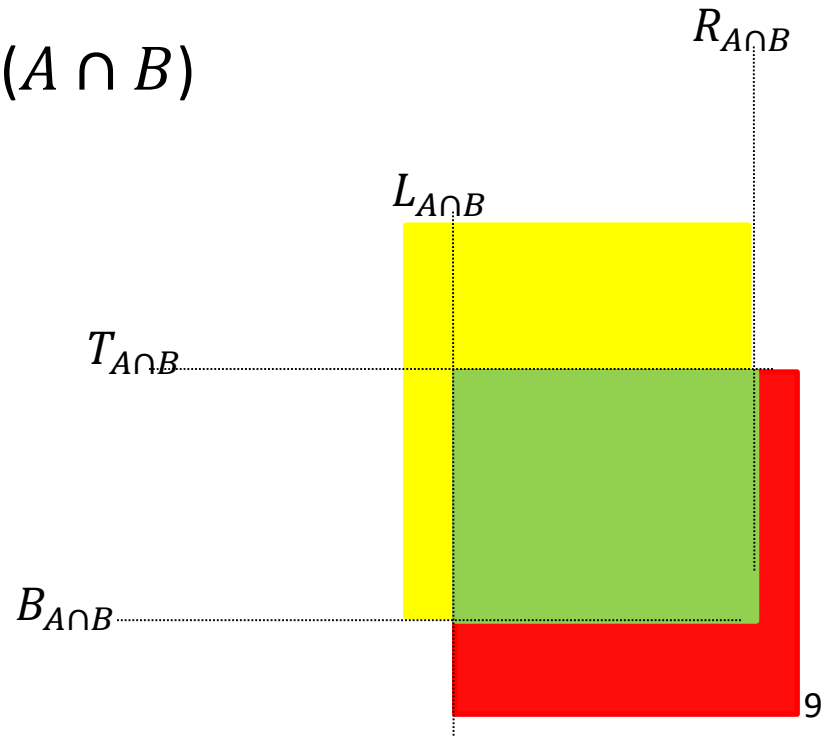
Computing Intersection Over Union

- IoU(A,B) is defined as this ratio:

$$\frac{\#\text{ of pixels in } A \cap B}{\#\text{ of pixels in } A \cup B}$$

- The area of $A \cup B$ is computed as:

Area($A \cup B$) = Area(A)+Area(B)-Area($A \cap B$)

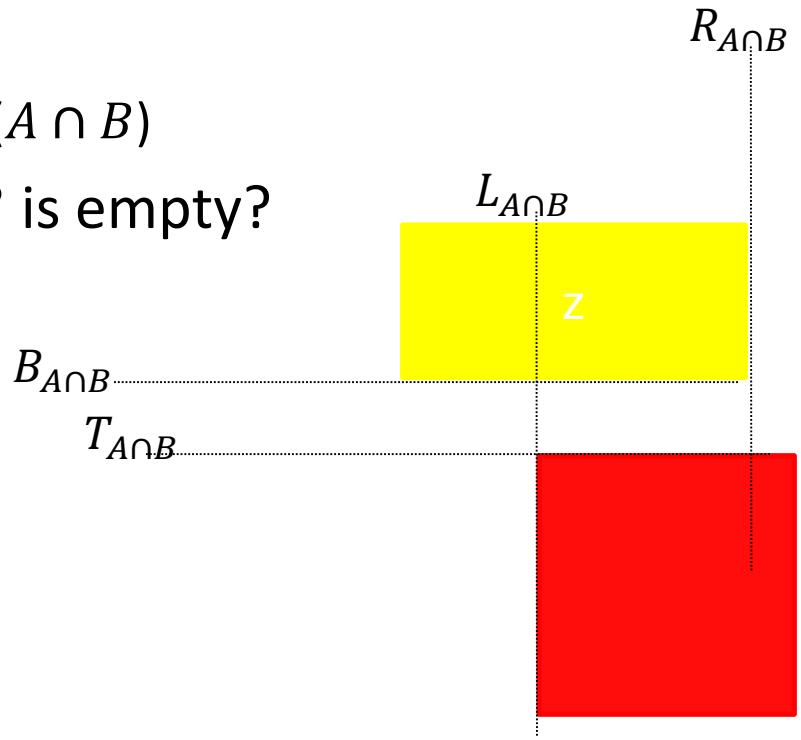


Computing Intersection Over Union

- IoU(A,B) is defined as this ratio:

$$\frac{\#\text{ of pixels in } A \cap B}{\#\text{ of pixels in } A \cup B}$$

  - Height($A \cap B$) = $B_{A\cap B} - T_{A\cap B} + 1$
  - Width($A \cap B$) = $R_{A\cap B} - L_{A\cap B} + 1$
  - Area($A \cap B$) = Height($A \cap B$)* Width($A \cap B$)
- How can our code tell when $A \cap B$ is empty?
  - In that case, at least one of Height($A \cap B$) and Width($A \cap B$) is zero or negative.
  - Your code needs to identify such cases, and report that IoU(A,B)=0 in such cases.



Measuring Accuracy over a Dataset

- Suppose we have some dataset for face detection.
  - Let's say we have 10,000 images, and the ground truth marks 20,000 face locations on those images.
- Suppose we choose an IoU threshold of 0.8.
- Suppose that 100% of the detected boxes pass the IoU threshold.
  - This means that, for each image in the dataset, each of the detected boxes has an IoU score of at least 0.8 with at least one ground truth location in that image.
- Is this a good result?

Measuring Accuracy over a Dataset

- Suppose we have some dataset for face detection.
  - Let's say we have 10,000 images, and the ground truth marks 20,000 face locations on those images.
- Suppose we choose an IoU threshold of 0.8.
- Suppose that 100% of the detected boxes pass the IoU threshold.
- Is this a good result?
- This is a classic example of an uninformative and incomplete result.
- Vital missing information: how many ground truth boxes were NOT matched by any detection result?

Measuring Accuracy over a Dataset

- Suppose we have some dataset for face detection.
  - Let's say we have 10,000 images, and the ground truth marks 20,000 face locations on those images.
- Suppose we choose an IoU threshold of 0.8.
- Suppose that 100% of the detected boxes pass the IoU threshold.
- One extreme example:
  - Every ground truth box was matched by one and only one detected box.
  - Then, our detector had perfect accuracy.

Measuring Accuracy over a Dataset

- Suppose we have some dataset for face detection.
  - Let's say we have 10,000 images, and the ground truth marks 20,000 face locations on those images.
- Suppose we choose an IoU threshold of 0.8.
- Suppose that 100% of the detected boxes pass the IoU threshold.
- Another extreme example:
  - 10% of the ground truth boxes were matched by one and only one detected box.
  - 90% of ground truth boxes were not matched.
  - Then, even though 100% of the detected boxes pass the IoU threshold, the detection accuracy is pretty low.

True/False Positives/Negatives

- **True positive**: a detection box whose IoU score with a ground truth box is over the threshold.
  - Intuitively: a true positive is a case where:
    - an object is present in an image, and
    - the detector correctly detects the location of that object in the image.

True/False Positives/Negatives

- **True positive**: a detection box whose IoU score with a ground truth box is over the threshold.
- **False positive**: a detection box whose **maximum** IoU score with all ground truth boxes is under the threshold.
  - Intuitively, a false positive is a false detection:
    - an object was detected, but
    - there is no object at that location.
  - In borderline cases that count as false positives:
    - there is an object in the detected location, but
    - the detected location was not accurate enough, and the IoU score between the detected location and the ground truth location was too small.

True/False Positives/Negatives

- **True positive**: a detection box whose IoU score with a ground truth box is over the threshold.
- **False positive**: a detection box whose **maximum** IoU score with all ground truth boxes is under the threshold.
- **False negative**: a ground truth box whose maximum IoU score with a ground truth box is over the threshold.
  - Intuitively, a false negative is an object that was not detected.
  - In borderline cases that count as false negatives:
    - there is a detection that overlaps with the ground truth box.
    - the detected location was not accurate enough, and the IoU score between the detected location and the ground truth location was too small.

Measuring Accuracy over a Dataset

- **To measure detection accuracy over a dataset we must always report two numbers.**
- One number should be about true positives, or false negatives.
  - For example: report the ratio of number of true positives over number of object locations marked by the ground truth.
    - A 93.5% ratio indicates that 93.5% of the objects of interest that were present in the dataset were correctly detected.
  - Alternatively, report the ratio of number of false negatives over number of object locations marked by the ground truth.
    - A 2.3% ratio indicates that 2.3% of the objects of interest that were present in the dataset were not detected.

Measuring Accuracy over a Dataset

- **To measure detection accuracy over a dataset we must always report two numbers.**
- One number should be about true positives, or false negatives.
- One number should be about false positives.
  - For example: report the total number of false positives in the dataset.
    - A number of 635 indicates that there were 635 detections in the dataset that did not correspond to actual locations of objects of interest.
  - Alternatively: report the number of false positives per image.
    - A number of 0.4 indicates that, on average, there were 0.4 cases per image where a detection did not correspond to an actual location of an object of interest.

Thresholds

- In measuring detection accuracy, we need to use two thresholds:
- One threshold is the IoU threshold.
  - 0.5 is a common choice, but it is better to try different values and report results for each value.
- The second threshold is the **detection threshold**.
  - Usually, for any detector, we need to set a threshold to decide what constitutes a “detection”.
  - For example, for normalized cross-correlation, we need to set a threshold, such that any score above that threshold should be treated as a detection (unless that score is removed by non-maxima suppression).

Detection Threshold

- Suppose that we are using normalized cross-correlation for detection.
- Consider two detection thresholds:
  - $T_1 = 0.7$
  - $T_2 = 0.8$.
- Suppose that we fix the IoU threshold to 0.5.
- How do we expect the results of the two detection thresholds to compare to each other?
- With $T_1 = 0.7$, there should be more detections than with $T_2 = 0.8$.
  - Some of those extra detections with $T_1 = 0.7$ will be correct, so T_1 will produce a higher true positive ratio.
  - Some of those extra detections with $T_1 = 0.7$ will be incorrect, so T_1 will produce a higher false positive rate.

Detection Threshold

- Changing the detection threshold usually leads to one of the two numbers (true positives or false positives) getting better and the other one getting worse.
- Usually, to evaluate a detector, we try several different detection thresholds.
  - Every threshold leads to a true positive ratio and a false positive rate.
- We make a **true positive vs. false positive** plot, where, for example:
  - The x-axis is the true positive ratio.
  - The y-axis is the false positive rate.
  - Every point on that plot corresponds to a (true positive, false positive) result obtained by a specific choice of detection threshold.

Detection Threshold

- To compare two detectors A and B, we look at their two plots.
  - If detector A is clearly better, its curve should be **lower** than B's curve.
  - Lower curve means that for the same x-axis value (same true positive ratio), the false positive rate is lower.