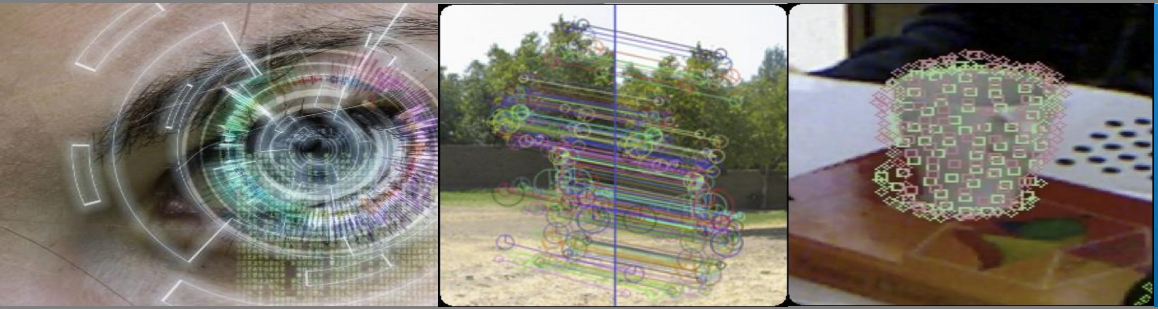
Visual Intelligence Theory

Deep Learning Basics (#22: Pre-trained Classifier-To-Detector)



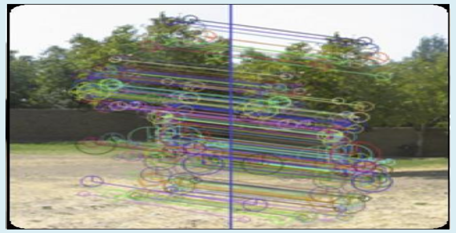
2023 Autumn

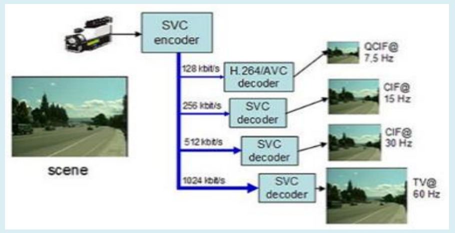
Prof. Byung-Gyu Kim
Intelligent Vision Processing Lab. (IVPL)
http://ivpl.sookmyung.ac.kr
Dept. of IT Engineering, Sookmyung Women's University
E-mail: bg.kim@sookmyung.ac.kr

Goal of this lecture

❖ **Turning any CNN image classifier into an object detector with Keras, TensorFlow, and OpenCV**

- Concept
- Model Structure
- Actual Practices and Applications





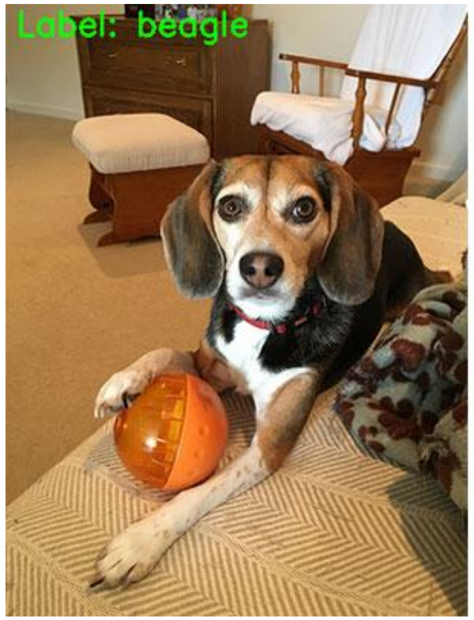
Contents

- **Image classification vs. object detection**
- Deep learning image classifier into an object detector
- Actual Practices and Applications

Image classification vs. object detection (1)

❖ key differences between image classification and object detection tasks:

- **Image Classification**
  - A single class label and a probability associated with the class label prediction (*left*).
  - This class label is meant to characterize the contents of the entire image, or at least the most dominant, visible contents of the image.
  - **We can thus think of image classification as:**
    - One image in
    - One class label out



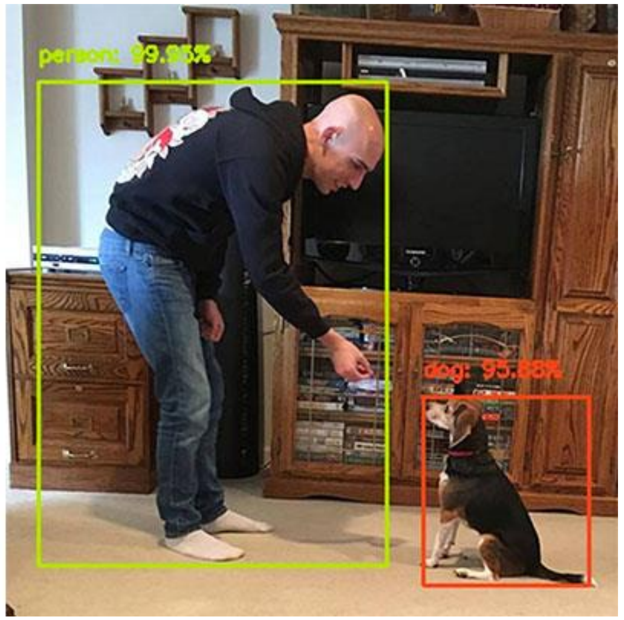


Image classification vs. object detection (2)

- **Object detection**
  - not only tells us *what* is in the image (i.e., class label) but also *where* in the image the object is via bounding box *(x, y)*-coordinates (right).
  - **Therefore, object detection algorithms allow us to:**
    - Input one image
    - Obtain *multiple* bounding boxes and class labels as output

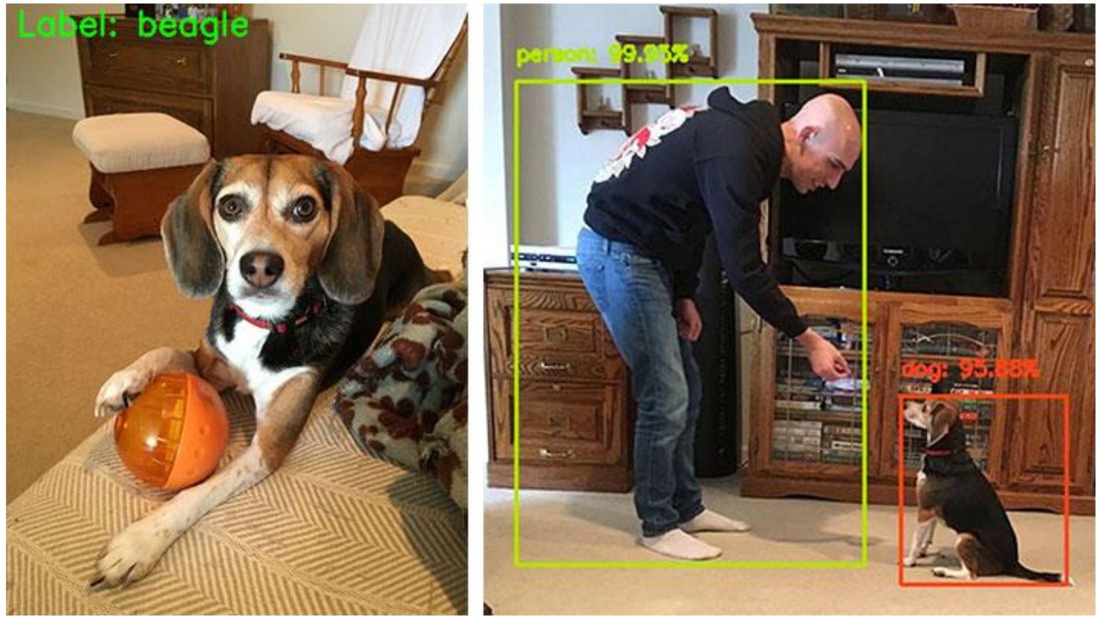
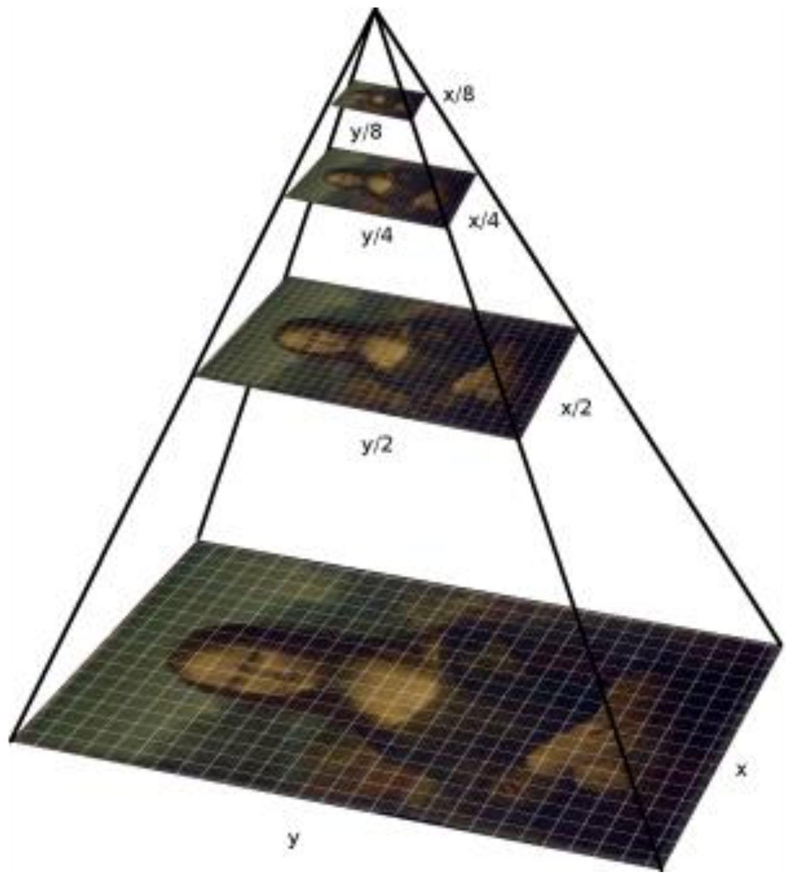


Image classification vs. object detection (3)

- Any **object detection algorithm** (regardless of traditional computer vision or state-of-the-art deep learning)
  - **1] Input:** An image that we wish to apply object detection to
  - **2] Output:** Three values, including:
    - 2a) A **list of bounding boxes,** or the (x, y)-coordinates for each object in an image
    - 2b) The **class label** associated with each of the bounding boxes
    - 2c) The **probability/confidence score** associated with each bounding box and class label

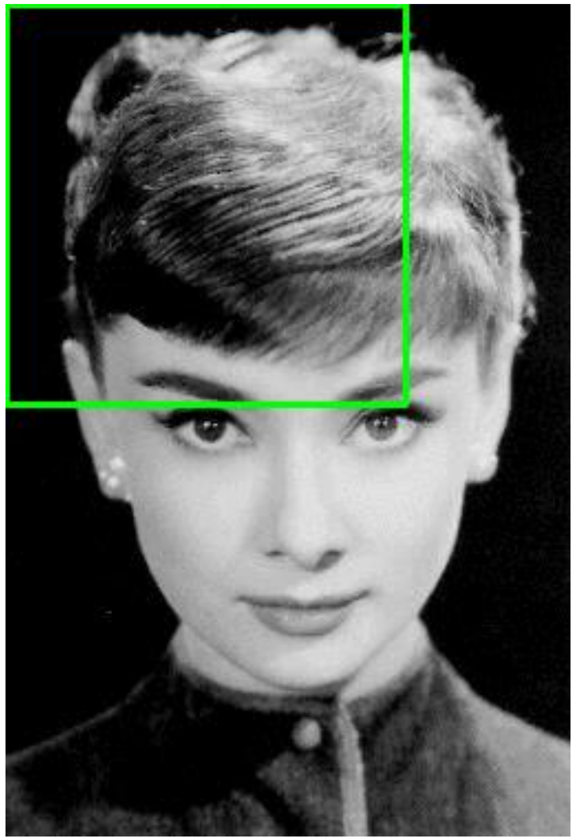
Turn any deep learning image classifier into an object detector (1)

- Key ingredients
  - **1] The first key ingredient is to use *image pyramids***
    - to **find objects in images at different scales (i.e., sizes)** of an image



Turn any deep learning image classifier into an object detector (2)

- 2] **The second key ingredient we need is *sliding windows:***
  - A sliding window is a fixed-size rectangle that slides from *left-to-right* and *top-to-bottom* within an image.

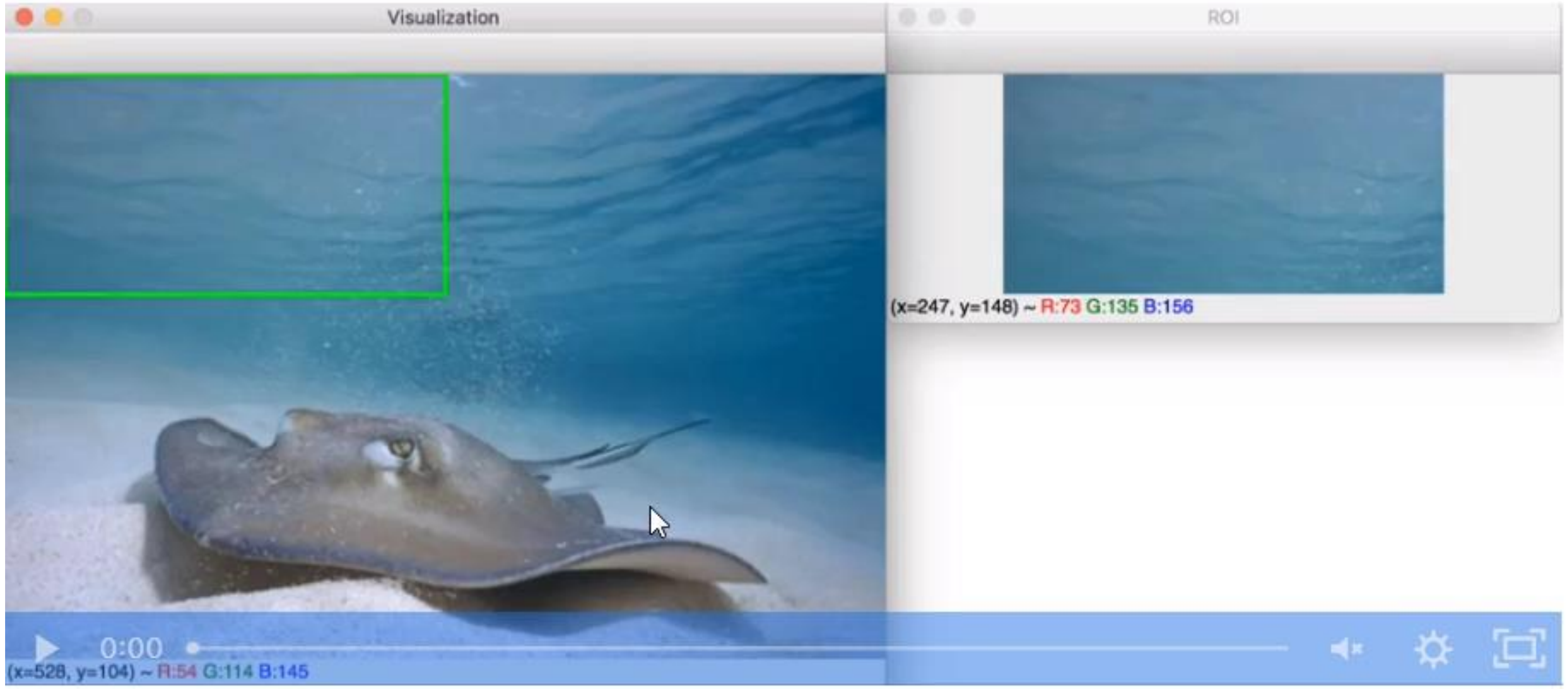


IVPL Intelligent Vision Processing Lab

Turn any deep learning image classifier into an object detector (3)

❖ **Converting classifier to detector**:

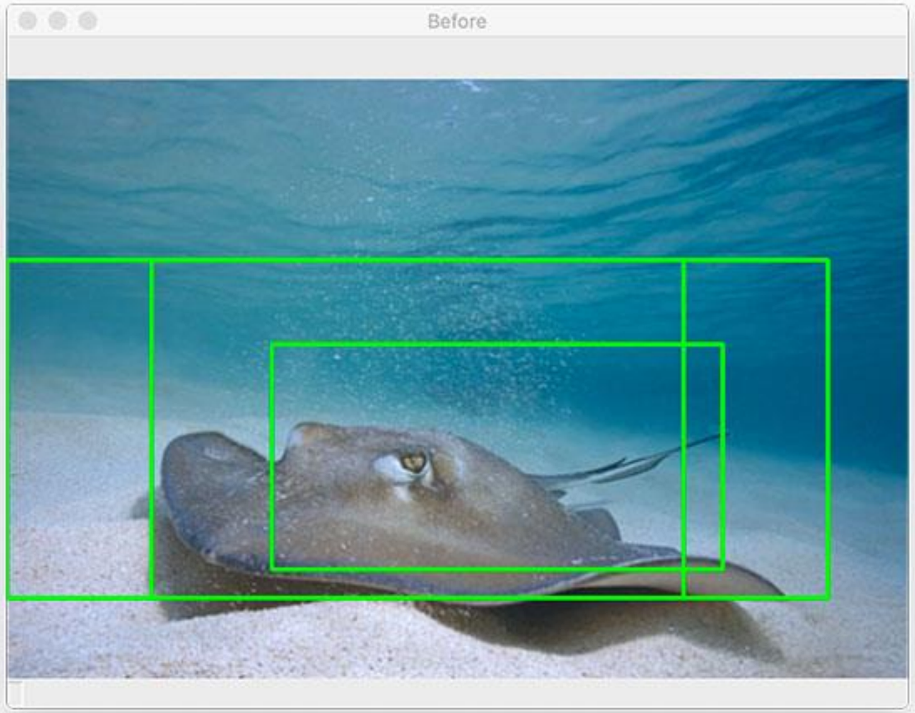
- At each stop of the window we would:
  - **1) Extract the ROI**
  - **2) Pass it through our image classifier (ex., Linear SVM, CNN, etc.)**
  - **3) Obtain the output predictions**
  - **4) Show boxes, label names and so on.**


Visualization
ROI
(x=247, y=148) ~ R:73 G:135 B:156
0:00
(x=528, y=104) ~ R:54 G:114 B:145

Turn any deep learning image classifier into an object detector (4)

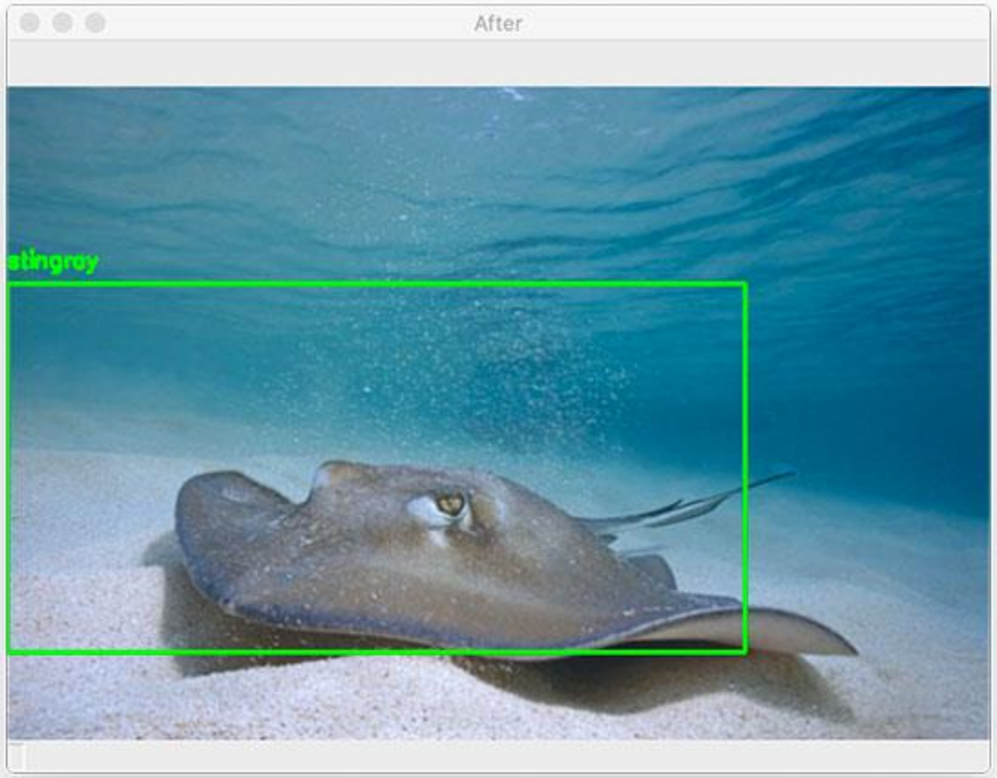
- **3] The final key ingredient we need is *non-maxima suppression.***
  - When performing object detection, our object detector will typically produce multiple, overlapping bounding boxes surrounding an object in an image.
  - It simply implies that as the sliding window approaches an image, our classifier component is returning larger and larger probabilities of a positive detection.

  →There's only *one* object there, and we somehow need to collapse/remove the extraneous bounding boxes.



Turn any deep learning image classifier into an object detector (5)

- **How to suppress** non-maxima, which collapses weak, overlapping bounding boxes in favor of the more confident ones?



[After non-maxima suppression (NMS) has been applied]

IVPL Intelligent Vision Processing Lab

Turn any deep learning image classifier into an object detector (6)

❖ **Combining traditional computer vision with deep learning to build an object detector**

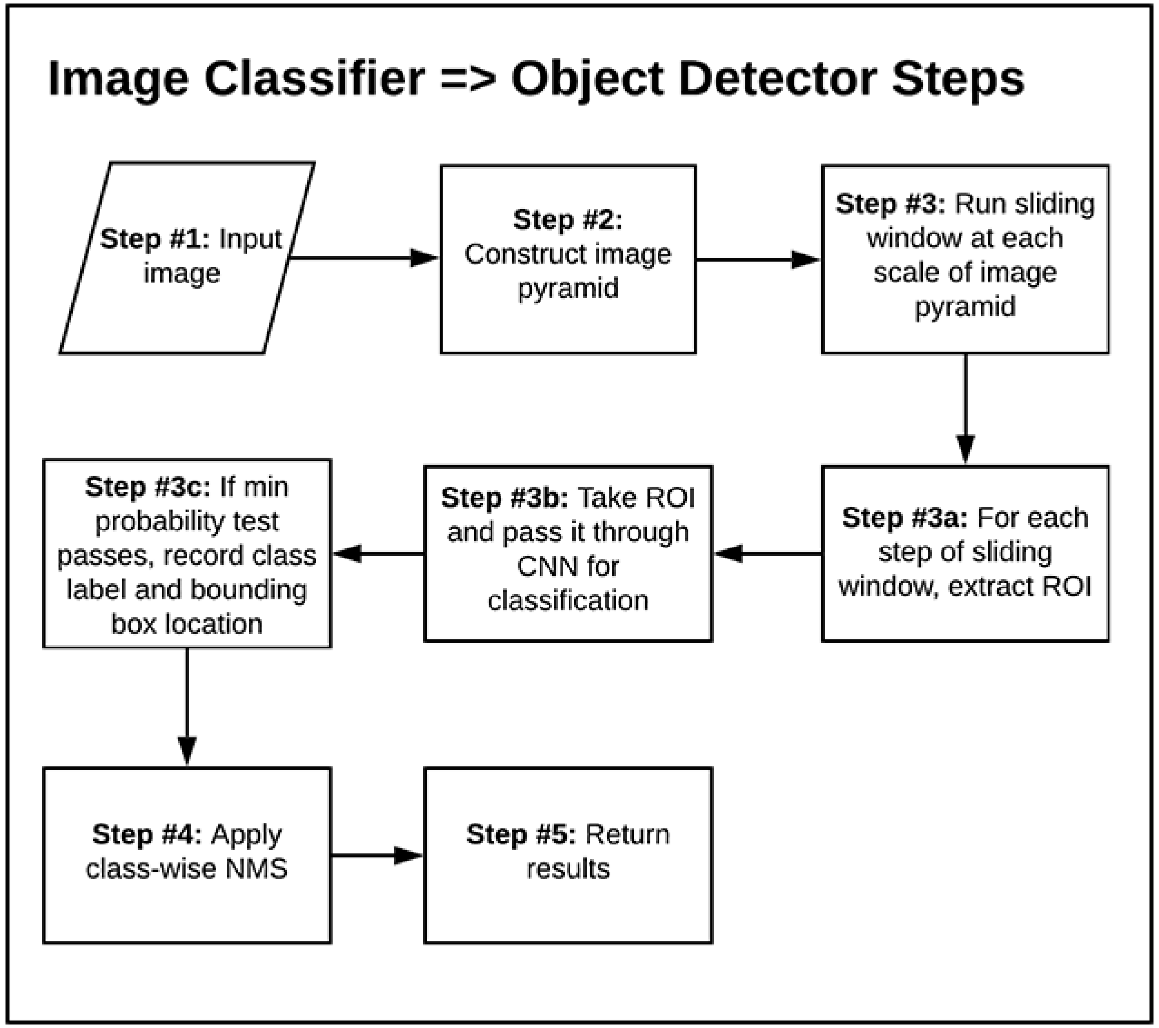


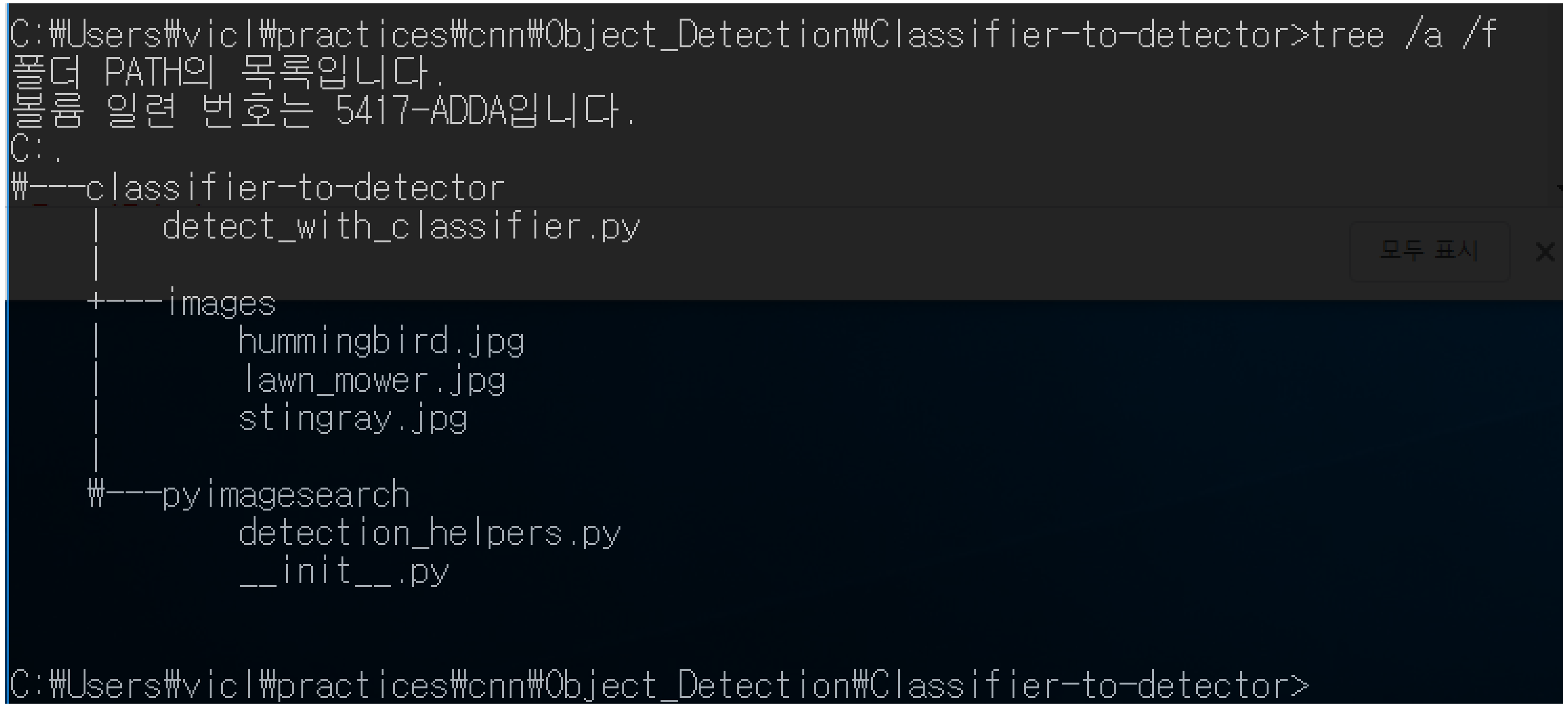
Image Classifier => Object Detector Steps
Step #1: Input image
Step #2: Construct image pyramid
Step #3: Run sliding window at each scale of image pyramid
Step #3a: For each step of sliding window, extract ROI
Step #3b: Take ROI and pass it through CNN for classification
Step #3c: If min probability test passes, record class label and bounding box location
Step #4: Apply class-wise NMS
Step #5: Return results

Turn any deep learning image classifier into an object detector (7)

- **Step #1**: Input an image
- **Step #2**: Construct an image pyramid
- **Step #3**: For each scale of the image pyramid, run a sliding window
  - **Step #3a:** For each stop of the sliding window, extract the ROI
  - **Step #3b:** Take the ROI and pass it through our CNN originally trained for image classification
  - **Step #3c:** Examine the probability of the top class label of the CNN, and if meets a minimum confidence, record (1) the class label and (2) the location of the sliding window
- **Step #4**: Apply class-wise non-maxima suppression to the bounding boxes
- **Step #5**: Return results to calling function

Actual Practices and Applications (1)

❖ Project structure



```
C:₩Users₩vicl₩practices₩cnn₩Object_Detection₩Classifier-to-detector>tree /a /f
폴더 PATH의 목록입니다.
볼륨 일련 번호는 5417-ADDA입니다.
C:.
₩---classifier-to-detector
    |   detect_with_classifier.py
    |
    +---images
    |       hummingbird.jpg
    |       lawn_mower.jpg
    |       stingray.jpg
    |
    ₩---pyimagesearch
            detection_helpers.py
            __init__.py


C:₩Users₩vicl₩practices₩cnn₩Object_Detection₩Classifier-to-detector>
```

- two helper functions:
  - image_pyramid : Assists in generating copies of our image at different scales so that we can find objects of different sizes
  - `sliding_window` : Helps us find where in the image an object is by sliding our classification window from left-to-right (column-wise) and top-to-bottom (row-wise)

IVPL Intelligent Vision Processing Lab

Actual Practices and Applications (2)

- **Classifier**:
  - A pre-trained ResNet50 CNN using ImageNet (1000 classes)

- Source Analysis
  - detection_helpers.py

```
# import the necessary packages
import imutils

def sliding_window(image, step, ws):
        # slide a window across the image
        for y in range(0, image.shape[0] - ws[1], step):
                for x in range(0, image.shape[1] - ws[0], step):
                        # yield the current window
                        yield (x, y, image[y:y + ws[1], x:x + ws[0]])
```

• **image**: The *input image* that we are going to loop over and generate windows from. This input image may come from the output of our image pyramid.
• **step**: Our *step size,* which indicates how many pixels we are going to "skip" in both the *(x, y)* directions. Normally, we would *not* want to loop over each and every pixel of the image (i.e., step=1), as this would be computationally prohibitive if we were applying an image classifier at each window. Instead, the step size is determined on a per-dataset basis and is tuned to give optimal performance based on your dataset of images. In practice, it's common to use a step of 4 to 8 pixels. Remember, the smaller your step size is, the more windows you'll need to examine.
• **ws**: The *window size* defines the width and height (in pixels) of the window we are going to extract from our image. If you scroll back to **Figure**, the *window size* is equivalent to the *dimensions* of the *green* box that is sliding across the image.

Actual Practices and Applications (3)

- **image_pyramid** function

```
def image_pyramid(image, scale=1.5, minSize=(224, 224)):
        # yield the original image
        yield image
        # keep looping over the image pyramid
        while True:
                # compute the dimensions of the next image in the pyramid
                w = int(image.shape[1] / scale)
                image = imutils.resize(image, width=w)
                # if the resized image does not meet the supplied minimum
                # size, then stop constructing the pyramid
                if image.shape[0] < minSize[1] or image.shape[1] < minSize[0]:
                                break
                # yield the next image in the pyramid
                yield image
```

• **image**: The *input image* for which we wish to generate multi-scale representations.
• **scale**: *scale factor* controls how much the image is resized at each layer. Smaller scale values yield more layers in the pyramid, and larger scale values yield fewer layers.
• **`minSize`**: Controls the *minimum size* of an output image (layer of our pyramid). This is important because we could effectively construct progressively smaller scaled representations of our input image infinitely. Without a minSize parameter, our while loop would continue forever (which is *not* what we want).

Actual Practices and Applications (4)

- detect_with_classifier.py

```
# initialize variables used for the object detection procedure
WIDTH = 600
PYR_SCALE = 1.5
WIN_STEP = 16
ROI_SIZE = eval(args["size"])
INPUT_SIZE = (224, 224)
```

•WIDTH: Given that the selection of images for testing (refer to the *"Project Structure"* section) are all slightly different in size, we set a constant width here for later resizing purposes. By ensuring our images have a consistent starting width, we know that the image will fit on our screen.
•PYR_SCALE: Our image pyramid *scale factor*. This value controls how much the image is resized at each layer. Smaller scale values yield more layers in the pyramid, and larger scales yield fewer layers. The fewer layers you have, the faster the overall object detection system will operate, potentially at the expense of accuracy.
•WIN_STEP: Our sliding window *step size,* which indicates how many pixels we are going to "skip" in both the *(x, y)* directions. Remember, the smaller your step size is, the more windows you'll need to examine, which leads to a slower overall object detection execution time. In practice, I would recommend trying values of 4 and 8 to start with (depending on the dimensions of your input and your minSize).
•ROI_SIZE: Controls the aspect ratio of the objects we want to detect; if a mistake is made setting the aspect ratio, it will be nearly impossible to detect objects. Additionally, this value is related to the image pyramid minSize value — giving our image pyramid generator a means of exiting. As you can see, this value comes directly from our --size command line argument.
•INPUT_SIZE: The classification *CNN dimensions*. Note that the tuple defined here heavily depends on the CNN you are using (in our case, it is ResNet50).

Actual Practices and Applications (5)

```
# load our network weights from disk
print("[INFO] loading network...")
model = ResNet50(weights="imagenet", include_top=True)
# load the input image from disk, resize it such that it has the
# has the supplied width, and then grab its dimensions
orig = cv2.imread(args["image"])
orig = imutils.resize(orig, width=WIDTH)
(H, W) = orig.shape[:2]

# initialize the image pyramid
pyramid = image_pyramid(orig, scale=PYR_SCALE, minSize=ROI_SIZE)
# initialize two lists, one to hold the ROIs generated from the image
# pyramid and sliding window, and another list used to store the
# (x, y)-coordinates of where the ROI was in the original image
rois = []
locs = []
# time how long it takes to loop over the image pyramid layers and
# sliding window locations
start = time.time()
```

- rois : Holds the regions of interest (ROIs) generated from pyramid + sliding window output.
- locs: Stores the *(x, y)*-coordinates of where the ROI was in the original image.

IVPL Intelligent Vision Processing Lab

Actual Practices and Applications (6)

```
# loop over the image pyramid
for image in pyramid:
        # determine the scale factor between the *original* image
        # dimensions and the *current* layer of the pyramid
        scale = W / float(image.shape[1])
        # for each layer of the image pyramid, loop over the sliding
        # window locations
        for (x, y, roiOrig) in sliding_window(image, WIN_STEP, ROI_SIZE):
                # scale the (x, y)-coordinates of the ROI with respect to the
                # *original* image dimensions
                x = int(x * scale)
                y = int(y * scale)
                w = int(ROI_SIZE[0] * scale)
                h = int(ROI_SIZE[1] * scale)
                # take the ROI and preprocess it so we can later classify
                # the region using Keras/TensorFlow
                roi = cv2.resize(roiOrig, INPUT_SIZE)
                roi = img_to_array(roi)
                roi = preprocess_input(roi)
                # update our list of ROIs and associated coordinates
                rois.append(roi)
                locs.append((x, y, x + w, y + h))
```

Scale coordinates

Grab the ROI and preprocess

Update the list of rois and associated locs coordinates

Actual Practices and Applications (7)

```
# check to see if we are visualizing each of the sliding
# windows in the image pyramid
if args["visualize"] > 0:
    # clone the original image and then draw a bounding box
    # surrounding the current region
    clone = orig.copy()
    cv2.rectangle(clone, (x, y), (x + w, y + h),
    (0, 255, 0), 2)
    # show the visualization and current ROI
    cv2.imshow("Visualization", clone)
    cv2.imshow("ROI", roiOrig)
    cv2.waitKey(0)
```

optional visualization

Actual Practices and Applications (8)

```
# show how long it took to loop over the image pyramid layers and
# sliding window locations
end = time.time()
print("[INFO] looping over pyramid/windows took {:.5f} seconds".format(end - start))
# convert the ROIs to a NumPy array
rois = np.array(rois, dtype="float32")
# classify each of the proposal ROIs using ResNet and then show how
# long the classifications took
print("[INFO] classifying ROIs...")
start = time.time()
preds = model.predict(rois)
end = time.time()
print("[INFO] classifying ROIs took {:.5f} seconds".format(end - start))

# decode the predictions and initialize a dictionary which maps class
# labels (keys) to any ROIs associated with that label (values)
preds = imagenet_utils.decode_predictions(preds, top=1)
labels = {}
```

take the ROIs and pass them (in batch) through our pre-trained image classifier (i.e., ResNet) via predict

Decodes the predictions, grabbing only the top prediction for each ROI.

Actual Practices and Applications (9)

```
# loop over the predictions
for (i, p) in enumerate(preds):
        # grab the prediction information for the current ROI
        (imagenetID, label, prob) = p[0]

        # filter out weak detections by ensuring the predicted probability
        # is greater than the minimum probability
        if prob >= args["min_conf"]:
                # grab the bounding box associated with the prediction and
                # convert the coordinates
                box = locs[i]
                # grab the list of predictions for the lTTabel and add the
                # bounding box and probability to the list
                L = labels.get(label, [])
                L.append((box, prob))
                labels[label] = L
```

The bounding box and prob score tuple (value) associated with each class label (key).

Actual Practices and Applications (10)

```
# loop over the labels for each of detected objects in the image
for label in labels.keys():
	# clone the original image so that we can draw on it
	print("[INFO] showing results for '{}'".format(label))
	clone = orig.copy()
	# loop over all bounding boxes for the current label
	for (box, prob) in labels[label]:
		# draw the bounding box on the image
		(startX, startY, endX, endY) = box
		cv2.rectangle(clone, (startX, startY), (endX, endY),
		  (0, 255, 0), 2)
	# show the results *before* applying non-maxima suppression,
	then
	# clone the image again so we can display the results *after*
	# applying non-maxima suppression
	cv2.imshow("Before", clone)
	clone = orig.copy()
```

annotate all bounding boxes for the current label

visualize the before/after applying NMS

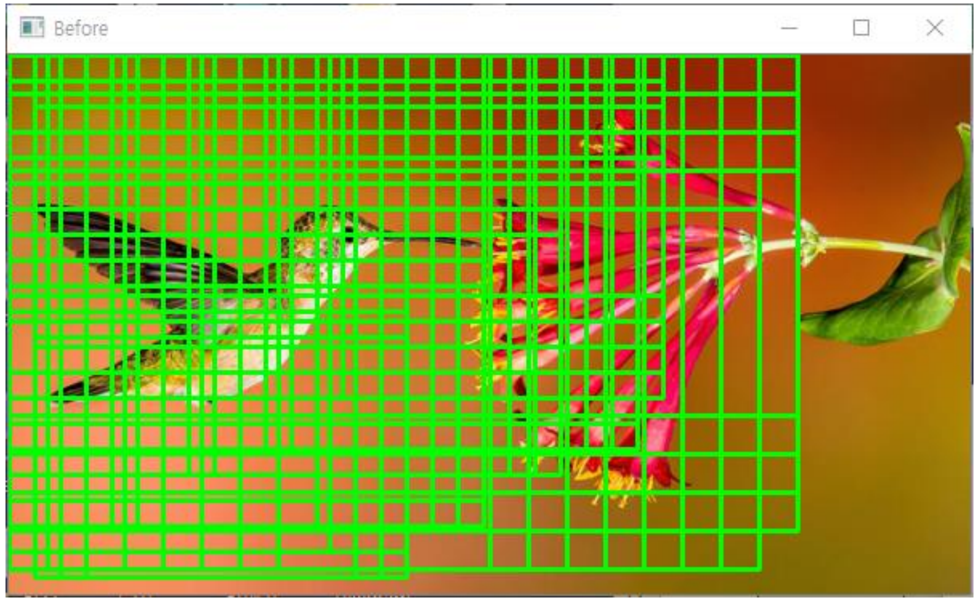
Actual Practices and Applications (11)

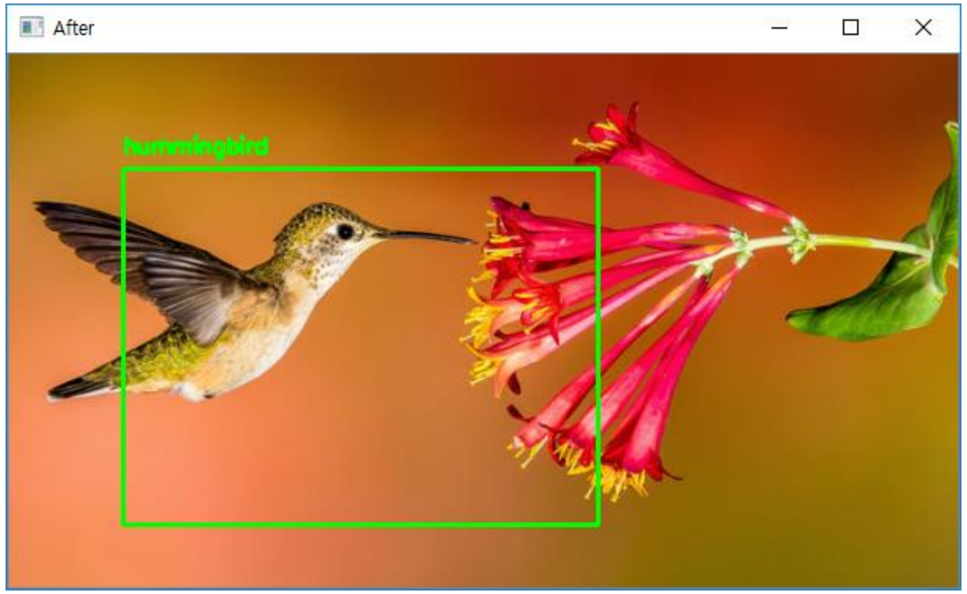
```
# extract the bounding boxes and associated prediction
# probabilities, then apply non-maxima suppression
boxes = np.array([p[0] for p in labels[label]])
proba = np.array([p[1] for p in labels[label]])
boxes = non_max_suppression(boxes, proba)
# loop over all bounding boxes that were kept after applying
# non-maxima suppression
for (startX, startY, endX, endY) in boxes:
        # draw the bounding box and label on the image
        cv2.rectangle(clone, (startX, startY), (endX, endY),
        (0, 255, 0), 2)
        y = startY - 10 if startY - 10 > 10 else startY + 10
        cv2.putText(clone, label, (startX, y),
        cv2.FONT_HERSHEY_SIMPLEX, 0.45, (0, 255, 0), 2)

# show the output after apply non-maxima suppression
cv2.imshow("After", clone)
cv2.waitKey(0)
```

annotate bounding box rectangles and labels on the “after” NMS image

Actual Practices and Applications (12)

❖ Some results with the developed detector





Intelligent Vision Processing Lab

Thank you for your attention.!!!
QnA

http://ivpl.sookmyung.ac.kr