Human Pose Estimation using Keypoint RCNN in PyTorch

 Chetan Patil  Vikas Gupta

JUNE 21, 2021

Application Pose PyTorch

Human Pose Estimation is an important research area in the field of Computer Vision. It deals with estimating unique points on the human body, also called **keypoints**. In this blog post, we will discuss one such algorithm for finding keypoints on images containing a human called **Keypoint-RCNN**. The code is written in Pytorch, using the **Torchvision** library.

Assume, you want to build a personal fitness trainer, one that can guide you to strike the right body pose, by analyzing the postures of the body joints. This is where Pose Estimation comes into play.

The idea of Keypoint Detection is to detect interest points or key locations in an image. These could be:

- the facial landmarks (such as nose-tip, eye-corners, face-boundary etc)
- or the body-joints (shoulders, wrists, ankles) in a person
- or the corners and blobs in an image

We have discussed Faster RCNN for Object Detection as well as Mask RCNN for Instance Segmentation in our earlier posts, but let's start at the very beginning here.

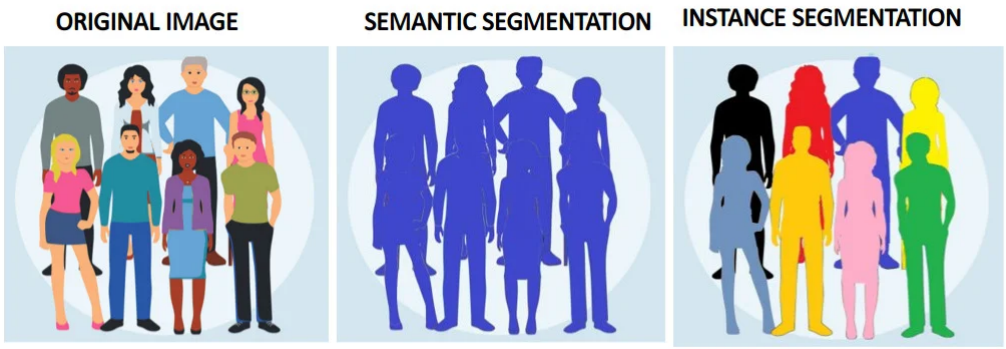
From RCNN to Mask-RCNN

- It all started with RCNN (Region-based Convolutional Neural Networks) evolving into Fast-RCNN, and then, Faster-RCNN.
- Even Faster-RCNN was limited in the sense that it could only detect objects.
- A variant thus followed just a few years later called Mask-RCNN. This variant of Faster-RCNN was published to tackle the problem of Segmentation.

  Mask-RCNN was one of the earlier papers published on Instance Segmentation.

Now, what is Instance Segmentation?
It refers to cases, where every detected object is explicitly segmented.

You also need to know here how Instance Segmentation differs from Semantic Segmentation. Learn more about types of Image segmentation in our blog post. The image below can help understand the difference easily.



Semantic Segmentation vs Instance Segmentation

Well, this is not the end of the story. In the Mask-RCNN paper, the authors also extended the model's capabilities to detect keypoints in the human body. Just a slight modification in the Mask-RCNN presented a new solution for Keypoint Detection.

That brings us to our topic of discussion today, the Keypoint RCNN. Come, let us explore Keypoint Detection, using this modified version of Mask-RCNN.

You already know that Faster-RCNN evolved into Mask-RCNN, and then into Keypoint-RCNN. We will also be discussing the architecture of each, but first let us focus on knowing Keypoint and its applications.

This also constitutes a part of our series on **PyTorch for Beginners**.

Applications of Keypoint Detection

Keypoint Detection has a wide range of applications. Here are a few:

Determining the right body postures of a person during exercise

Body posture check requires finding angles between different key points to predict the posture. Based on this information, one can check whether the angles and elevation of different bones (like arms, legs, back etc.) while exercising is correct or not.

Determining right body postures of a person during exercise

Facial expression detection

Smile Detection using Facial Landmarks

Analyzing the shape of the face to determine its expression, and then using it to understand behaviour and stress levels. For example, it is possible to estimate whether a person is smiling or not, based on the keypoints present on the lips of that person.

Activity Recognition

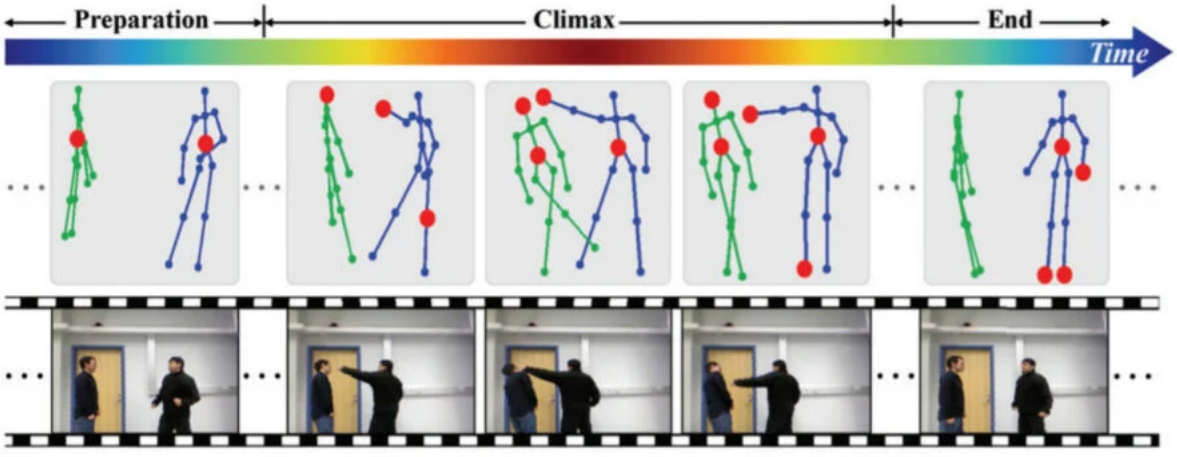


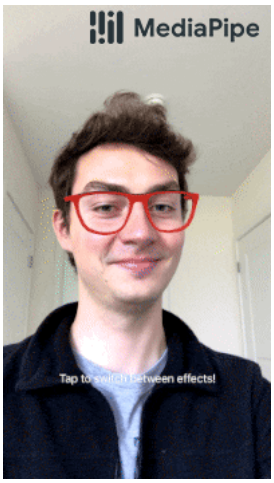
Figure 1: Illustration of the procedure for an action "punching". An action may experience different stages, and involve different discriminative subsets of joints (as the red circles).

Recognising the activity of "Punching"

Monitoring the movement of a child in a cradle, or a crowded place (like the railway station or airport) helps determine and flag any suspicious activity in its vicinity.
Deep Learning models can be trained to predict the keypoint locations and recognize such specified activity, in a given set of consecutive

frames.

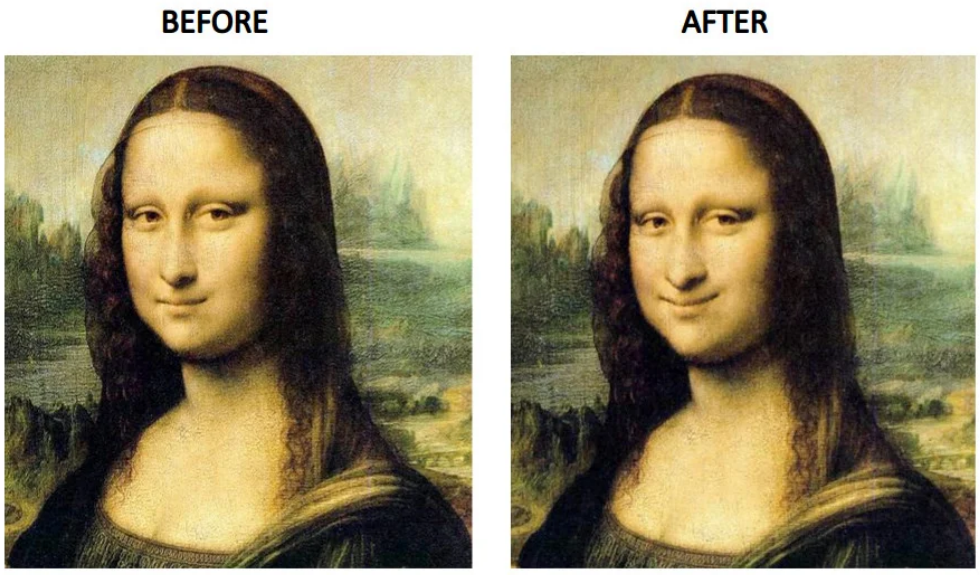
Snapchat like filters



Snapchat like filter

Overlaying a graphical object (eg: a mask, goggles etc) for fun activity, by predicting the 3D locations of the facial landmarks, and then projecting the object on the face.

Photoshop effects



Making Monalisa smile using Least-Squares method

Editing pictures to create a fake smile, or enlarging the eyes to create a buggy-eye effect. The idea here is to warp only a small region of the image, keeping the rest of it intact.
With the help of facial Keypoint Detection, one can get the control points, and then map them to a new set of locations, using the Moving Least Squares method.

Face Morphing

Morphing between faces

Face Morphing is used extensively to morph images of different human characters or objects. The idea here is to create a smooth transition from one face to another.

The process involves:

1. Taking two images, detecting facial landmarks, and then aligning the faces to a standard representation.
2. Next, slowly blend the first and second image to create a smooth transition, from the first to the second image.
   For more details, refer to our post on Face Morphing.

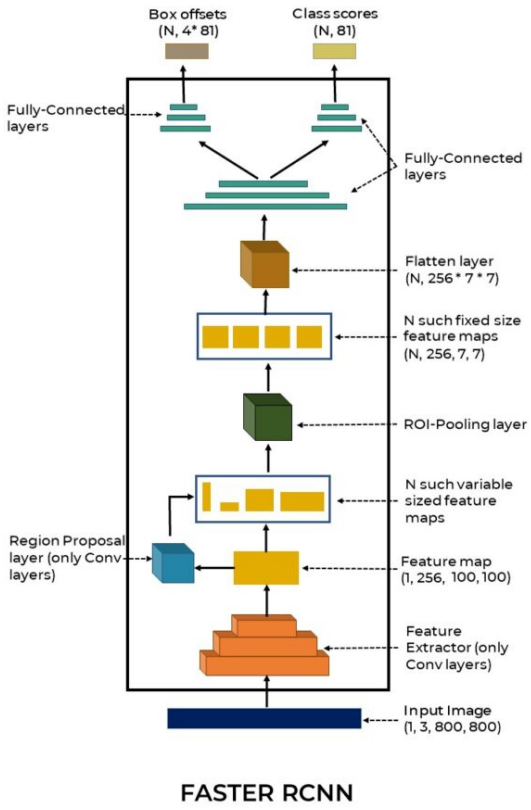
Evolution of Keypoint **RCNN Architecture**

You have seen how Keypoint RCNN followed Mask RCNN, which in turn came after Faster RCNN. So, when introducing Keypoint RCNN, a brief overview of its predecessors is important.

Let us assume these variables for this post:

1. `N` is the number of objects proposed by the Region-Proposal Layer.
2. `C` is the number of classes present in the MS-COCO dataset, which is 80.
3. `K` is the number of keypoints per person, which is 17.

Faster RCNN



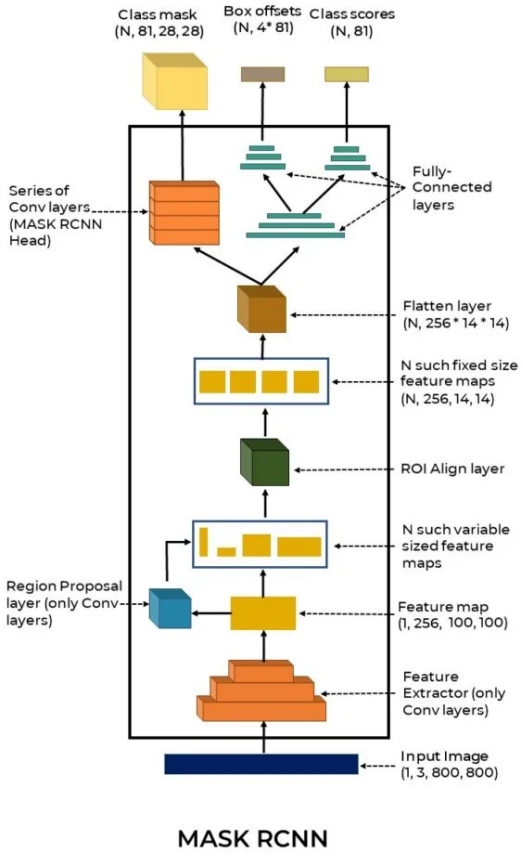
Architecture of Faster-RCNN

The architecture of Faster-RCNN, as you can see in the above image, has numerous layers.

- The Region-Proposal-Layer predicts the rough location of `N` number of objects detected in the feature map. These variable-sized regions are then individually passed to the ROI-Pooling Layer.
- The ROI-Pooling Layer resizes the feature-map (proposed by the Region Proposal Network) to a fixed size by simply quantising the variable sized feature map to a fixed size grid and then picking out the max-values from the variable sized map and placing them in the fixed grid.
  - In our case, the fixed size is `[256, 7, 7]` (`[channels, height, width]`).
  - This is done because the succeeding layers are all Fully Connected and need fixed-input features.
- A Fully-Connected (FC) Layer follows the ROI-Pooling layer. This layer is further split into two separate FC-blocks
  - one for predicting the class-scores for the proposed object, with output size `[N, C]`
  - another for adjusting the box-coordinates for the proposed object, with output size `[N, 4 *C]` (each class is associated with a bounding-box, represented by `[x_center, y_center, width, height]`)

As RPN proposed `N` objects, you will have `N` such class-scores and bounding-box predictions.

Mask RCNN



Architecture of Mask-RCNN

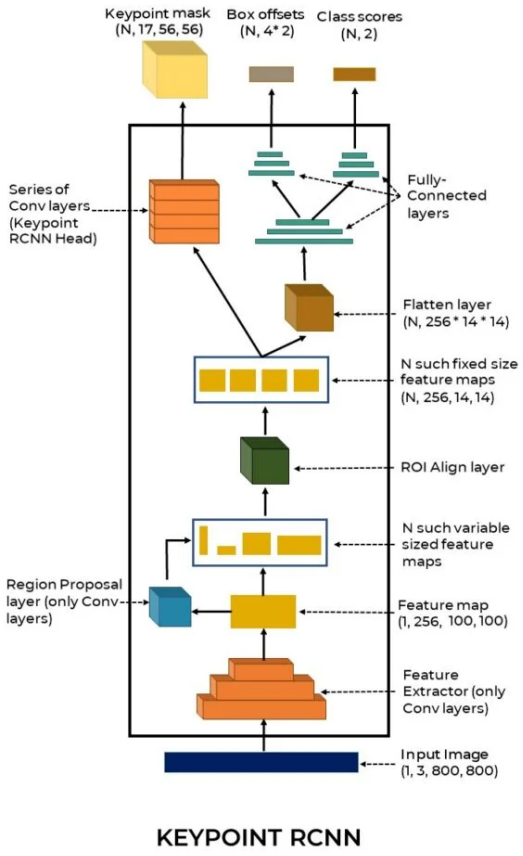
The architecture of Mask is very similar to that of Faster-RCNN, with only a minor addition and some modification of layers.

1. It uses the ROI-Align Layer instead of the ROI-Pooling layer because of its higher accuracy Unlike Segmentation, Object Detection requires high-level information. Because the ROI-Pooling layer discards some information during quantization, it cannot be used. The authors of Mask-RCNN thus came up with the ROI-Align layer. t Instead of quantization, ROI-Align uses bilinear-interpolation to fill up the values in the fixed-size featuremap, from the variable-sized one.
2. The output of ROI-Align is passed to another branch called the Mask-RCNN head (see the above image).
    - This branch is basically a series of convolutional layers, with final output size [N, C, 28, 28].
    - This output represents N number of class-wise masks, with C (80) channels of size [28, 28]. (Each of the C channels corresponds to a specific class (like bus, person, train etc).

Keypoint RCNN

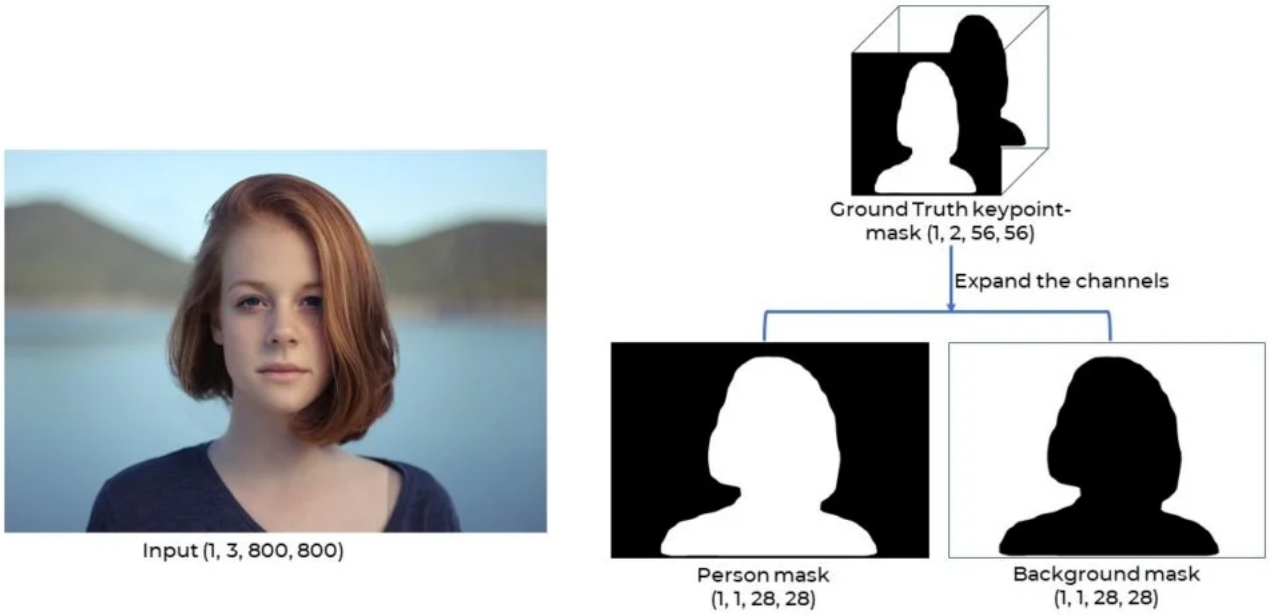
The architecture of Keypoint RCNN resembles the Mask-RCNN. They just differ in the output size, and the way the keypoints are encoded in the keypoint mask.

Note that the COCO Dataset offers keypoints only for the person class. So, here we'll discuss Keypoint Detection only in that context.



Architecture of Keypoint RCNN

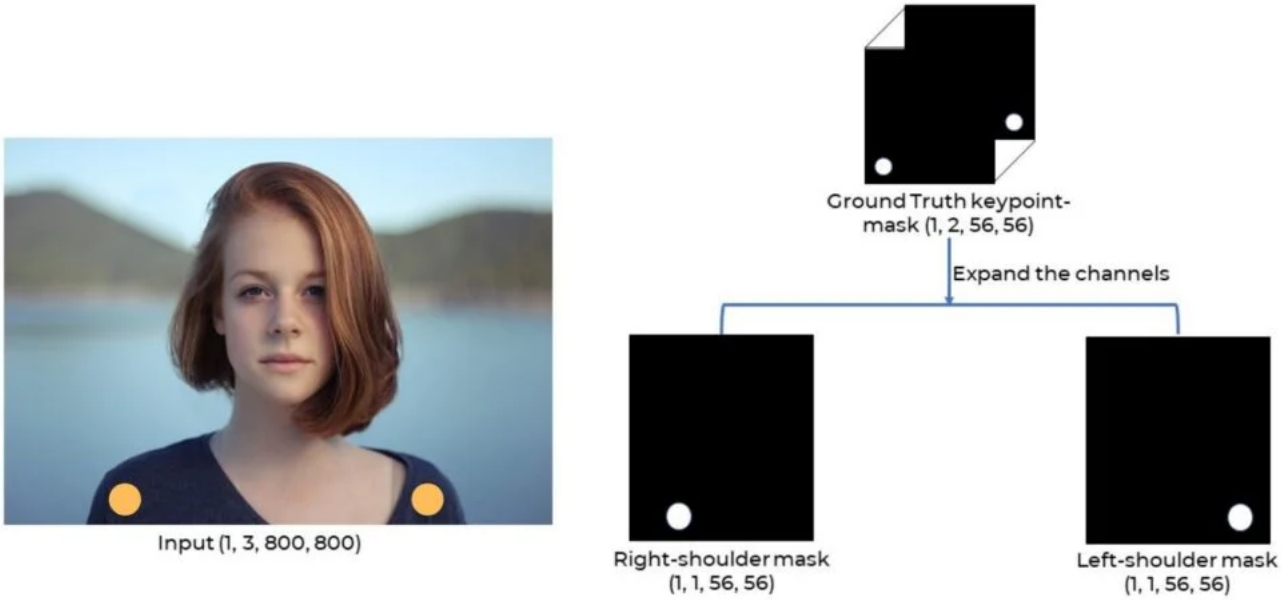
Keypoint RCNN slightly modifies the existing Mask RCNN, by one-hot encoding a keypoint (instead of the whole mask) of the detected object. Let's take a slight detour to understand how the keypoints are encoded, with a visual example. Consider we are solving a Person-Background segmentation problem using Mask-RCNN.



Class-wise output feature map from Mask RCNN

In this case our ground-truth class mask will be of size `[1, 2, 28, 28]`. The 2 channels represent a channel for the person and the background class. As we see above, each channel is responsible for highlighting a specific class. We can also think of a similar approach to encoding a keypoint.

Consider we are trying to estimate the keypoint locations of a person's left shoulder and right shoulder.
Can we relate this problem to the above image? Definitely. Following is an image to encode the keypoints in the output mask.



Encoding the keypoints in the output mask

As you see, we were able to figure out a way to highlight the locations for the left-shoulder and right-shoulder in the keypoint mask. Well, this is the encoding we were talking about. Now let's quickly jump back to the nitty-gritties of Keypoint-RCNN for humans. Before the detour, we mentioned that the output of Keypoint-RCNN is a slightly modified version Mask-RCNN's output

- therefore output from Keypoint-RCNN is now sized `[N, K=17, 56, 56]`.
- Each of the K channels corresponds to a specific keypoint (for eg: left-elbow, right-ankle etc).

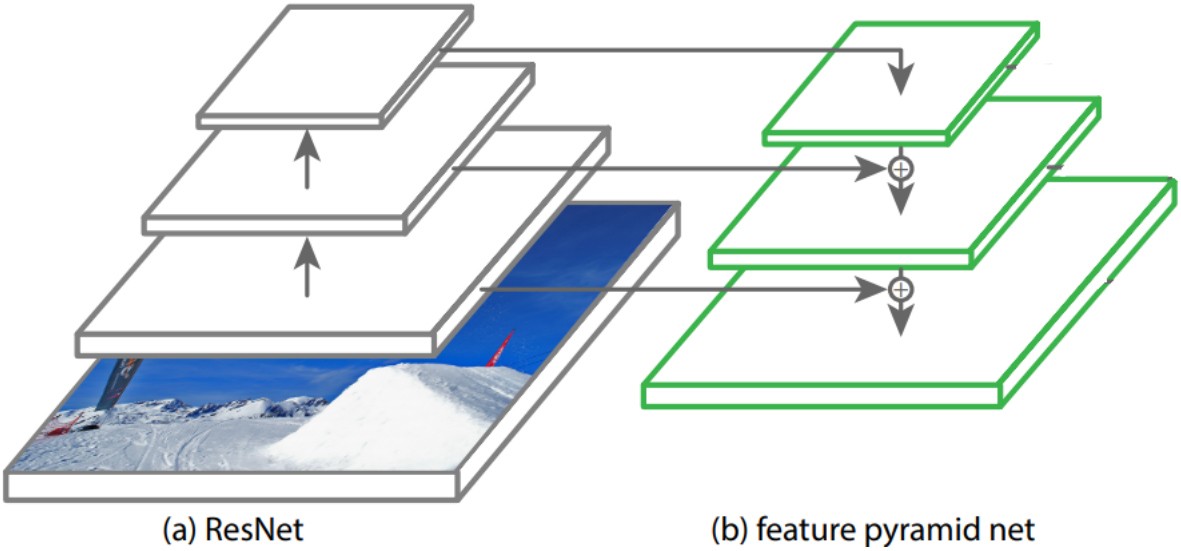
Please note that in Keypoint-RCNN, as you're dealing with keypoints of only the person class, therefore:

- The final class-scores will be of size `[N, 2]`
  - one for background
  - the other for the person class
- Similarly, the box-predictions will be sized `[N, 2 * 4]`.

Now that you have a brief overview of its architecture, let's check out the Keypoint-RCNN model, provided by Torchvision.

Torchvision's Keypoint Detection API

Torchvision has a pretrained Keypoint Detection model, in its detection module. The model is built on top of the ResNet-50 FPN (Feature Pyramid Network) backbone. Feature Pyramid Network is the concept of fusing feature maps at multiple scales to preserve information at multiple levels. This backbone architecture was also used in RetinaNet (which introduced Focal-Loss).

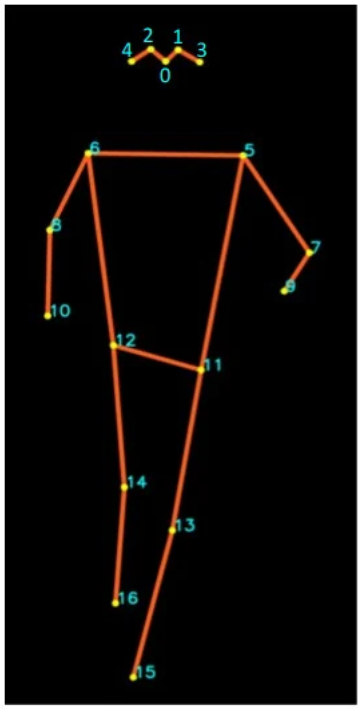


ResNet Feature Pyramid Network

The Keypoint RCNN is trained on the MS-COCO (Common Objects in Context) dataset, which offers different annotation types for Object Detection, Segmentation and Image Captioning. Keypoint Detection falls in the same list.

Note that originally COCO offered `80` classes for Detection and Segmentation. However, for Keypoint Detection, the annotations are offered only for the person class. The Keypoint-RCNN available in Torchvision is particularly trained to identify key points in a person, so you'll run inference on it.

Index	Key point
0	Nose
1	Left-eye
2	Right-eye
3	Left-ear
4	Right-ear
5	Left-shoulder
6	Right-shoulder
7	Left-elbow
8	Right-elbow
9	Left-wrist
10	Right-wrist
11	Left-hip
12	Right-hip
13	Left-knee
14	Right-knee
15	Left-ankle
16	Right-ankle



Keypoints available in COCO-dataset for humans

Download Code To easily follow along this tutorial, please download code by clicking on the button below. It's FREE!

Download Code

```
# create a model object from the keypointrcnn_resnet50_fpn class
model = torchvision.models.detection.keypointrcnn_resnet50_fpn(pretrained=True)
# call the eval() method to prepare the model for inference mode.
model.eval()

# create the list of keypoints.
keypoints = ['nose','left_eye','right_eye',\
'left_ear','right_ear','left_shoulder',\
'right_shoulder','left_elbow','right_elbow',\
'left_wrist','right_wrist','left_hip',\
'right_hip','left_knee', 'right_knee', \
'left_ankle','right_ankle']
```

Input Output Format

Input to the model a tensor of size `[batch_size, 3, height, width]`. Note that the original image should be normalized (i.e. the pixel values in the image should range between 0 and 1).

Do this by using the classes: `transforms.Compose()` and `transforms.ToTensor()`, which are available in the `transforms` module of Torchvision.

Once the preprocessing is done, simply pass the preprocessed input to the model to get the output (all the postprocessing such as Non-Max-Supression, getting keypoint locations from the keypoint mask is done in the `keypointrcnn_resnet50_fpn` class).

Please refer to roi_heads.py for the postprocessing code.

```
# import the transforms module
from torchvision import transforms as T

# Read the image using opencv
img_path ="./images/image_1.jpg"
img = cv2.imread(img_path)

# preprocess the input image
transform = T.Compose([T.ToTensor()])
```

```
img_tensor = transform(img)

# forward-pass the model
# the input is a list, hence the output will also be a list
output = model([img_tensor])[0]
```

The variable `output` is a dictionary, with the following keys and values:

boxes – A tensor of size `[N, 4]`, where `N` is the number of objects detected.

labels – A tensor of size `[N]`, depicting the class of the object.

- This is always 1 because each detected box belongs to a person.
- 0 stands for the background class.

scores – A tensor of size `[N]`, depicting the confidence score of the detected object.

keypoints – A tensor of size `[N, 17, 3]`, depicting the 17 joint locations of `N` number of persons. Out of 3, the first two numbers are the coordinates `x` and `y`, and the third one depicts the visibility.

- 0, when keypoint is invisible
- 1, when keypoint is visible

keypoints_scores – A tensor of size `[N, 17]`, depicting the score for all the keypoints, for each detected person.

Loss Function in Keypoint-RCNN

As in Keypoint Detection, each Ground-Truth keypoint is one-hot-encoded, across all the `K` channels, in the featuremap of size `[K=17, 56, 56]`, for a single object. For each visible Ground-Truth, channel wise Softmax (instead of sigmoid), from the final featuremap `[17, 56, 56]`, is used to minimize the Cross Entropy Loss.

$$\frac{-\sum_{h,w}\left[Y_{k,h,w}==1\right]\left(Y_{k,h,w}*\log\left(\text{softmax}\left(\hat{Y}_{k,h,w}\right)\right)\right)}{\sum_{h,w}\left[Y_{k,h,w}==1\right]}$$

Running Inference on a Sample Image

Once you have the output from the model, it is easy to draw keypoints for the detected person.
Keep in mind that most of the detected keypoints (for each person) appear nearly at the same locations.

Have a look at the image below

Multiple keypoints appearing at the same location

As we see above, there are multiple key points located near the same place. To overcome this, you need to filter out the detected person, based on their confidence score. Only key points for objects having a high confidence score are drawn. Don't forget that you also use the keypoint scores to filter out bad keypoints.

Let's check out this filtering function:

```
import matplotlib.pyplot as plt

def draw_keypoints_per_person(img, all_keypoints, all_scores, confs, keypoint_threshold=2, conf_threshold=0.9):
    # initialize a set of colors from the rainbow spectrum
    cmap = plt.get_cmap('rainbow')
    # create a copy of the image
    img_copy = img.copy()
    # pick a set of N color-ids from the spectrum
    color_id = np.arange(1,255, 255//len(all_keypoints)).tolist()[::-1]
    # iterate for every person detected
    for person_id in range(len(all_keypoints)):
      # check the confidence score of the detected person
      if confs[person_id]>conf_threshold:
        # grab the keypoint-locations for the detected person
        keypoints = all_keypoints[person_id, ...]
        # grab the keypoint-scores for the keypoints
        scores = all_scores[person_id, ...]
        # iterate for every keypoint-score
        for kp in range(len(scores)):
            # check the confidence score of detected keypoint
            if scores[kp]>keypoint_threshold:
                # convert the keypoint float-array to a python-list of integers
                keypoint = tuple(map(int, keypoints[kp, :2].detach().numpy().tolist()))
                # pick the color at the specific color-id
                color = tuple(np.asarray(cmap(color_id[person_id])[:-1])*255)
                # draw a circle over the keypoint location
                cv2.circle(img_copy, keypoint, 30, color, -1)
```

```
    return img_copy
```

Use the above function and the earlier input-output variables to draw the keypoints, over the original image.

```
keypoints_img = draw_keypoints_per_person(img, output["keypoints"], output["keypoints_scores"], output["scores"], keypoint_threshold=2)
```

Overlaying Keypoints on the input image

Getting the Skeletal Structure of the Detected Person

You have successfully figured out the key points of the person. But what about the pose? To get that, connect the joints together to form a skeleton-like structure. Following is a list of connections that form such a structure. Note that the skeletal structure will be the same for all detected persons. So, set these connections on a global scope.

Table depicting the connection between two keypoints to form a limb

```
def get_limbs_from_keypoints(keypoints):
  limbs = [
        [keypoints.index('right_eye'), keypoints.index('nose')],
        [keypoints.index('right_eye'), keypoints.index('right_ear')],
        [keypoints.index('left_eye'), keypoints.index('nose')],
        [keypoints.index('left_eye'), keypoints.index('left_ear')],
        [keypoints.index('right_shoulder'), keypoints.index('right_elbow')],
        [keypoints.index('right_elbow'), keypoints.index('right_wrist')],
        [keypoints.index('left_shoulder'), keypoints.index('left_elbow')],
        [keypoints.index('left_elbow'), keypoints.index('left_wrist')],
        [keypoints.index('right_hip'), keypoints.index('right_knee')],
        [keypoints.index('right_knee'), keypoints.index('right_ankle')],
        [keypoints.index('left_hip'), keypoints.index('left_knee')],
        [keypoints.index('left_knee'), keypoints.index('left_ankle')],
        [keypoints.index('right_shoulder'), keypoints.index('left_shoulder')],
        [keypoints.index('right_hip'), keypoints.index('left_hip')],
        [keypoints.index('right_shoulder'), keypoints.index('right_hip')],
        [keypoints.index('left_shoulder'), keypoints.index('left_hip')]
        ]
  return limbs

limbs = get_limbs_from_keypoints(keypoints)
```

Once you are ready with the joints or connections,

- Use a new function: `draw_skeleton_per_person`. It draws these connections for every detected person.
- Pass the same set of arguments that you did for the `draw_keypoints_per_person` function. The only difference here being that you will be drawing the limbs and not keypoints.
  - A limb is a line joining two keypoints. Since we are drawing/calculating a limb based on the keypoints, it makes sense to assign a confidence for that limb.
    We assign the limb-confidence/limb-score with one of the key points' score. The way we do this is pick the lowest confidence-score of the two keypoints and assign it to limb-score.
  - We can now consider a good limb as the one who's limb-score is greater than the keypoint threshold, which is set to 2.

```
def draw_skeleton_per_person(img, all_keypoints, all_scores, confs, keypoint_threshold=2, conf_threshold=0.9):

    # initialize a set of colors from the rainbow spectrum
    cmap = plt.get_cmap('rainbow')
    # create a copy of the image
    img_copy = img.copy()
    # check if the keypoints are detected
    if len(output["keypoints"])>0:
      # pick a set of N color-ids from the spectrum
      colors = np.arange(1,255, 255//len(all_keypoints)).tolist()[::-1]
```

```
    # iterate for every person detected
    for person_id in range(len(all_keypoints)):
        # check the confidence score of the detected person
        if confs[person_id]>conf_threshold:
          # grab the keypoint-locations for the detected person
          keypoints = all_keypoints[person_id, ...]

          # iterate for every limb
          for limb_id in range(len(limbs)):
            # pick the start-point of the limb
            limb_loc1 = keypoints[limbs[limb_id][0], :2].detach().numpy().astype(np.int32)
            # pick the start-point of the limb
            limb_loc2 = keypoints[limbs[limb_id][1], :2].detach().numpy().astype(np.int32)
            # consider limb-confidence score as the minimum keypoint score among the two keypoint scores
            limb_score = min(all_scores[person_id, limbs[limb_id][0]], all_scores[person_id, limbs[limb_id][1]])
            # check if limb-score is greater than threshold
            if limb_score> keypoint_threshold:
              # pick the color at a specific color-id
              color = tuple(np.asarray(cmap(colors[person_id])[:-1])*255)
              # draw the line for the limb
              cv2.line(img_copy, tuple(limb_loc1), tuple(limb_loc2), color, 25)

  return img_copy
```

Now, you can use the above function to create a skeletal structure of the person.

```
# overlay the skeleton in the detected person
skeletal_img = draw_skeleton_per_person(img, output["keypoints"], output["keypoints_scores"],
output["scores"],keypoint_threshold=2)
```

Overlaying a skeleton on the original image

Evaluation Metric in Keypoint Detection

- Tasks like Object Detection and Segmentation, employ Intersection Over Union as the metric to quantify the similarity between the ground-truth and the predicted box or mask.

- Keypoint Detection uses a metric called Object Keypoint Similarity (OKS), to quantify the closeness of the predicted keypoint-location, with the ground-truth keypoint. This metric ranges between 0 and 1.
  - The closer the predicted keypoint to the ground-truth, the closer will OKS approach 1.

Here's the formula:

$$OKS = \exp\left(-\frac{d_i^2}{2s^2k_i^2}\right)$$

Where `d_i` is the Euclidean distance between predicted and ground-truth, `s` is the object's scale, and `k_i` is a constant for a specific keypoint

Essentially, for `N` number of persons detected:

- You end up with `N` such values for s (as each detected person has its own scale).
- Also, there are `17` unique values for `k`, which remain constant for all the detected samples.

How do we find `s`?
Well, as we pointed out earlier that **s** refers to an object's scale, it simply is the square-root of the object's area. The bigger the object, the lesser should be the penalization, in other words, the better the OKS.

This should make sense. It is okay to predict a keypoint slightly away from the ground-truth keypoint, if the object is big. However, if the object is small, a slight deviation from the ground-truth might land the predicted keypoint out of the body itself. Such cases should be heavily penalized.

How to fix the values for `k`?
Well, as we mentioned earlier, k is a constant factor for each keypoint and it remains the same for all samples. It turns out that k is a measure of standard-deviation of a particular keypoint. Essentially, the value of `k` for keypoints on the face (eyes, ears, nose) have a relatively smaller standard deviation than the keypoints on the body (hips, shoulders, knees).

One beautiful thing about OKS is that it quantifies the same value for all predicted keypoints, at a particular radial distance from the ground-truth. The following visual will make it all clear:

In the above image:

- The green-dot is ground-truth keypoint, and each of the three brown-dots are possible examples for the predicted keypoint.
- You also have three sets of concentric circles, coinciding with the three predicted keypoints.
- All predicted keypoints lying on the:
  - innermost circle (yellow) will be quantified with a value of 0.88.
  - middle circle (red), will be quantified with a value of 0.75
  - outermost circle (blue), will be quantified with a value of 0.64.

Notice how the value of OKS approaches 1 as the predicted keypoint moves closer to the ground-truth keypoint.

OKS serves as a good metric to quantify the closeness of a predicted keypoint with the ground-truth one.

Inference Speed of Keypoint RCNN Tested on Colab and Colab Pro

Check out the following Table of Inference Speed on the Keypoint RCNN Model, tested on three different image sizes, over Google Colab and Colab Pro. Note that the original images are resized beforehand to match the model's input size. The table below thus reflects the exact input sizes fed to the model.

The FPS is calculated over the time interval between the model being fed to the network, and the final output (a dictionary), which we discussed in the Input-Output section.
The time interval excludes the pre-processing and includes the post-processing step.

The FPS shown below averages over 20 images.