Homework 3

Instructions

- This homework focuses on understanding and applying DETR for object detection and attention visualization. It consists of **three questions** designed to assess both theoretical understanding and practical application.
- Please organize your answers and results for the questions below and submit this jupyter notebook as **a .pdf file**.
- **Deadline: 11/14 (Thur) 23:59**

Reference

- End-to-End Object Detection with Transformers (DETR): https://github.com/facebookresearch/detr

Q1. Understanding DETR model

- Fill-in-the-blank exercise to test your understanding of critical parts of the DETR model workflow.

```
from torch import nn
class DETR(nn.Module):
    def __init__(self, num_classes, hidden_dim=256, nheads=8,
                 num_encoder_layers=6, num_decoder_layers=6, num_queries=100):
        super().__init__()

        # create ResNet-50 backbone
        self.backbone = resnet50()
        del self.backbone.fc

        # create conversion layer
        self.conv = nn.Conv2d(2048, hidden_dim, 1)

        # create a default PyTorch transformer
        self.transformer = nn.Transformer(
            hidden_dim, nheads, num_encoder_layers, num_decoder_layers)

        # prediction heads, one extra class for predicting non-empty slots
        # note that in baseline DETR linear_bbox layer is 3-layer MLP
        self.linear_class = nn.Linear(hidden_dim, num_classes + 1)
        self.linear_bbox = nn.Linear(hidden_dim, 4)

        # output positional encodings (object queries)
        self.query_pos = nn.Parameter(torch.rand(num_queries, hidden_dim))

        # spatial positional encodings
        # note that in baseline DETR we use sine positional encodings
        self.row_embed = nn.Parameter(torch.rand(50, hidden_dim // 2))
        self.col_embed = nn.Parameter(torch.rand(50, hidden_dim // 2))

    def forward(self, inputs):
        # propagate inputs through ResNet-50 up to avg-pool layer
        x = self.backbone.conv1(inputs)
        x = self.backbone.bn1(x)
        x = self.backbone.relu(x)
        x = self.backbone.maxpool(x)

        x = self.backbone.layer1(x)
        x = self.backbone.layer2(x)
        x = self.backbone.layer3(x)
        x = self.backbone.layer4(x)

        # convert from 2048 to 256 feature planes for the transformer
        h = self.conv(x)

        # construct positional encodings
        H, W = h.shape[-2:]
        pos = torch.cat([
            self.col_embed[:W].unsqueeze(0).repeat(H, 1, 1),
            self.row_embed[:H].unsqueeze(1).repeat(1, W, 1),
        ], dim=-1).flatten(0, 1).unsqueeze(1)

        # propagate through the transformer
        h = self.transformer(pos + 0.1 * h.flatten(2).permute(2, 0, 1),
                             self.query_pos.unsqueeze(1)).transpose(0, 1)



        # finally project transformer outputs to class labels and bounding boxes
```

```
        pred_logits = self.linear_class(h)
        pred_boxes = self.linear_bbox(h).sigmoid() #for normalizing the size of image

        return {'pred_logits': pred_logits,
                'pred_boxes': pred_boxes}


import torch
from torchvision.models import resnet50
custommodel = DETR(num_classes=5, num_queries=8)
custommodel.eval()
custommodel(torch.rand(1, 3, 800, 800))
```

```
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/transformer.py:379: UserWarning: enable_nested_tensor is True, but self.use_nested_tensc
  warnings.warn(
{'pred_logits': tensor([[[-3.5784e-01, -1.0815e-04,  3.6818e-01,  6.8822e-01, -3.4240e-01,
          -4.4160e-01],
         [-4.3936e-01, -4.6973e-03,  3.4736e-01,  7.4277e-01, -3.2232e-01,
          -4.8741e-01],
         [-4.7062e-01,  1.8902e-02,  3.3566e-01,  6.1343e-01, -1.8369e-01,
          -5.3941e-01],
         [-4.6228e-01, -3.7172e-02,  2.8302e-01,  6.6650e-01, -2.9272e-01,
          -4.4356e-01],
         [-5.2944e-01, -4.8554e-02,  3.1713e-01,  6.6337e-01, -2.7875e-01,
          -4.2134e-01],
         [-3.8838e-01, -7.1909e-02,  3.2587e-01,  6.6539e-01, -2.6635e-01,
          -3.5900e-01],
         [-4.4898e-01, -1.4394e-02,  3.9445e-01,  6.5305e-01, -2.8448e-01,
          -4.5535e-01],
         [-3.9307e-01, -8.7975e-03,  3.6776e-01,  7.9221e-01, -3.0319e-01,
          -4.5660e-01]]]),
 'pred_boxes': tensor([[[0.5486, 0.6910, 0.6938, 0.5235],
         [0.5263, 0.6871, 0.6895, 0.5243],
         [0.5232, 0.6933, 0.6983, 0.4979],
         [0.5211, 0.7002, 0.6900, 0.5387],
         [0.5274, 0.6971, 0.7097, 0.5160],
         [0.5218, 0.6904, 0.6998, 0.5326],
         [0.5280, 0.6979, 0.7048, 0.5204],
         [0.5426, 0.6924, 0.7005, 0.5113]]])}
```

Q2. Custom Image Detection and Attention Visualization

In this task, you will upload an **image of your choice** (different from the provided sample) and follow the steps below:

- Object Detection using DETR
  - Use the DETR model to detect objects in your uploaded image.
- Attention Visualization in Encoder
  - Visualize the regions of the image where the encoder focuses the most.
- Decoder Query Attention in Decoder
  - Visualize how the decoder's query attends to specific areas corresponding to the detected objects.

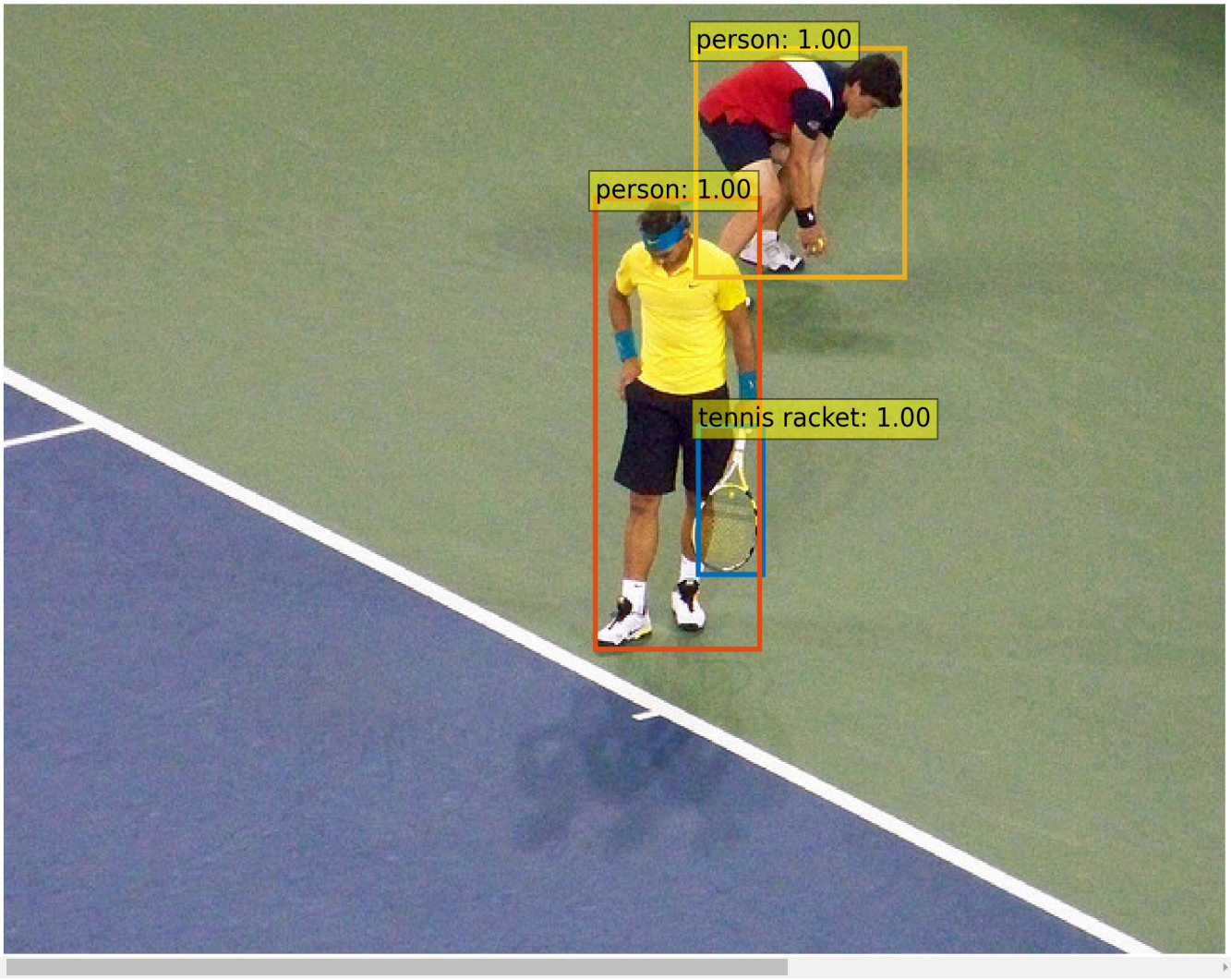
```
import math

from PIL import Image
import requests
import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'

import ipywidgets as widgets
from IPython.display import display, clear_output

import torch
from torch import nn


from torchvision.models import resnet50
import torchvision.transforms as T
torch.set_grad_enabled(False);

# COCO classes
CLASSES = [
    'N/A', 'person', 'bicycle', 'car', 'motorcycle', 'airplane', 'bus',
    'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'N/A',
    'stop sign', 'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse',
    'sheep', 'cow', 'elephant', 'bear', 'zebra', 'giraffe', 'N/A', 'backpack',
    'umbrella', 'N/A', 'N/A', 'handbag', 'tie', 'suitcase', 'frisbee', 'skis',
    'snowboard', 'sports ball', 'kite', 'baseball bat', 'baseball glove',
    'skateboard', 'surfboard', 'tennis racket', 'bottle', 'N/A', 'wine glass',
```

```
    'cup', 'fork', 'knife', 'spoon', 'bowl', 'banana', 'apple', 'sandwich',
    'orange', 'broccoli', 'carrot', 'hot dog', 'pizza', 'donut', 'cake',
    'chair', 'couch', 'potted plant', 'bed', 'N/A', 'dining table', 'N/A',
    'N/A', 'toilet', 'N/A', 'tv', 'laptop', 'mouse', 'remote', 'keyboard',
    'cell phone', 'microwave', 'oven', 'toaster', 'sink', 'refrigerator', 'N/A',
    'book', 'clock', 'vase', 'scissors', 'teddy bear', 'hair drier',
    'toothbrush'
]

# colors for visualization
COLORS = [[0.000, 0.447, 0.741], [0.850, 0.325, 0.098], [0.929, 0.694, 0.125],
          [0.494, 0.184, 0.556], [0.466, 0.674, 0.188], [0.301, 0.745, 0.933]]
# standard PyTorch mean-std input image normalization
transform = T.Compose([
    T.Resize(800),
    T.ToTensor(),
    T.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

# for output bounding box post-processing
def box_cxcywh_to_xyxy(x):
    x_c, y_c, w, h = x.unbind(1)
    b = [(x_c - 0.5 * w), (y_c - 0.5 * h),
         (x_c + 0.5 * w), (y_c + 0.5 * h)]
    return torch.stack(b, dim=1)

def rescale_bboxes(out_bbox, size):
    img_w, img_h = size
    b = box_cxcywh_to_xyxy(out_bbox)
    b = b * torch.tensor([img_w, img_h, img_w, img_h], dtype=torch.float32)
    return b

def plot_results(pil_img, prob, boxes):
    plt.figure(figsize=(16,10))
    plt.imshow(pil_img)
    ax = plt.gca()
    colors = COLORS * 100
    for p, (xmin, ymin, xmax, ymax), c in zip(prob, boxes.tolist(), colors):
        ax.add_patch(plt.Rectangle((xmin, ymin), xmax - xmin, ymax - ymin,
                                   fill=False, color=c, linewidth=3))
        cl = p.argmax()
        text = f'{CLASSES[cl]}: {p[cl]:0.2f}'
        ax.text(xmin, ymin, text, fontsize=15,
                bbox=dict(facecolor='yellow', alpha=0.5))
    plt.axis('off')
    plt.show()
```

In this section, we show-case how to load a model from hub, run it on a custom image, and print the result. Here we load the simplest model (DETR-R50) for fast inference. You can swap it with any other model from the model zoo.

```
model = torch.hub.load('facebookresearch/detr', 'detr_resnet50', pretrained=True)
model.eval();

url = 'http://images.cocodataset.org/val2017/000000384670.jpg'
im = Image.open(requests.get(url, stream=True).raw) # put your own image

# mean-std normalize the input image (batch-size: 1)
img = transform(im).unsqueeze(0)

# propagate through the model
outputs = model(img)

# keep only predictions with 0.7+ confidence
probas = outputs['pred_logits'].softmax(-1)[0, :, :-1]
keep = probas.max(-1).values > 0.9

# convert boxes from [0; 1] to image scales
bboxes_scaled = rescale_bboxes(outputs['pred_boxes'][0, keep], im.size)

plot_results(im, probas[keep], bboxes_scaled)
```

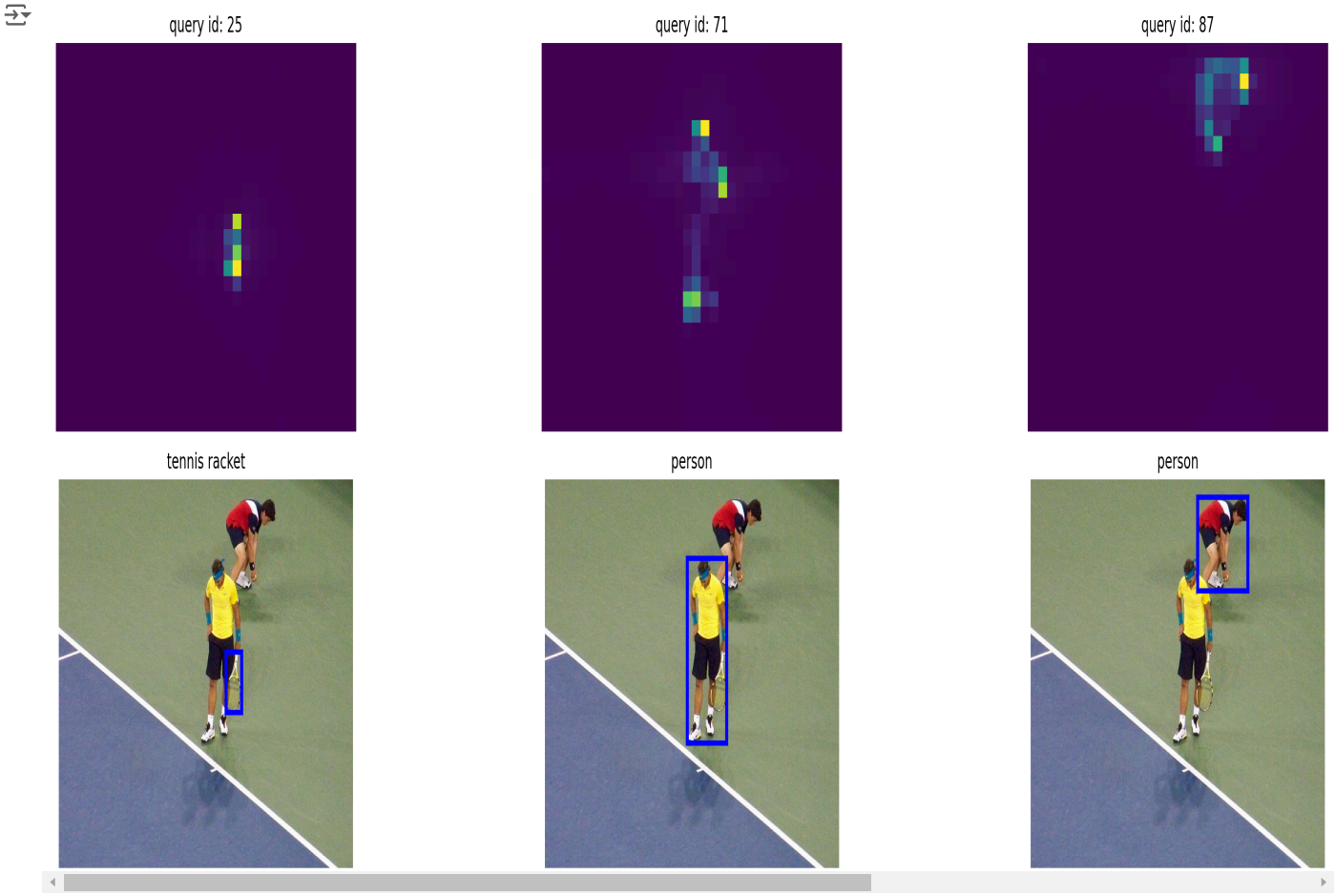
```
Using cache found in /root/.cache/torch/hub/facebookresearch_detr_main
```



Here we visualize attention weights of the last decoder layer. This corresponds to visualizing, for each detected objects, which part of the image the model was looking at to predict this specific bounding box and class.

```
# use lists to store the outputs via up-values
conv_features, enc_attn_weights, dec_attn_weights = [], [], []

hooks = [
    model.backbone[-2].register_forward_hook(
        lambda self, input, output: conv_features.append(output)
    ),
    model.transformer.encoder.layers[-1].self_attn.register_forward_hook(
        lambda self, input, output: enc_attn_weights.append(output[1])
    ),
    model.transformer.decoder.layers[-1].multihead_attn.register_forward_hook(
        lambda self, input, output: dec_attn_weights.append(output[1])
    ),
]

# propagate through the model
outputs = model(img) # put your own image

for hook in hooks:
    hook.remove()

# don't need the list anymore
conv_features = conv_features[0]
enc_attn_weights = enc_attn_weights[0]
dec_attn_weights = dec_attn_weights[0]


# get the feature map shape
h, w = conv_features['0'].tensors.shape[-2:]

fig, axs = plt.subplots(ncols=len(bboxes_scaled), nrows=2, figsize=(22, 7))
colors = COLORS * 100
for idx, ax_i, (xmin, ymin, xmax, ymax) in zip(keep.nonzero(), axs.T, bboxes_scaled):
    ax = ax_i[0]
```

```
    ax.imshow(dec_attn_weights[0, idx].view(h, w))
    ax.axis('off')
    ax.set_title(f'query id: {idx.item()}')
    ax = ax_i[1]
    ax.imshow(im)
    ax.add_patch(plt.Rectangle((xmin, ymin), xmax - xmin, ymax - ymin,
                               fill=False, color='blue', linewidth=3))
    ax.axis('off')
    ax.set_title(CLASSES[probas[idx].argmax()])
fig.tight_layout()
```




```
# output of the CNN
f_map = conv_features['0']
print("Encoder attention:      ", enc_attn_weights[0].shape)
print("Feature map:            ", f_map.tensors.shape)
```

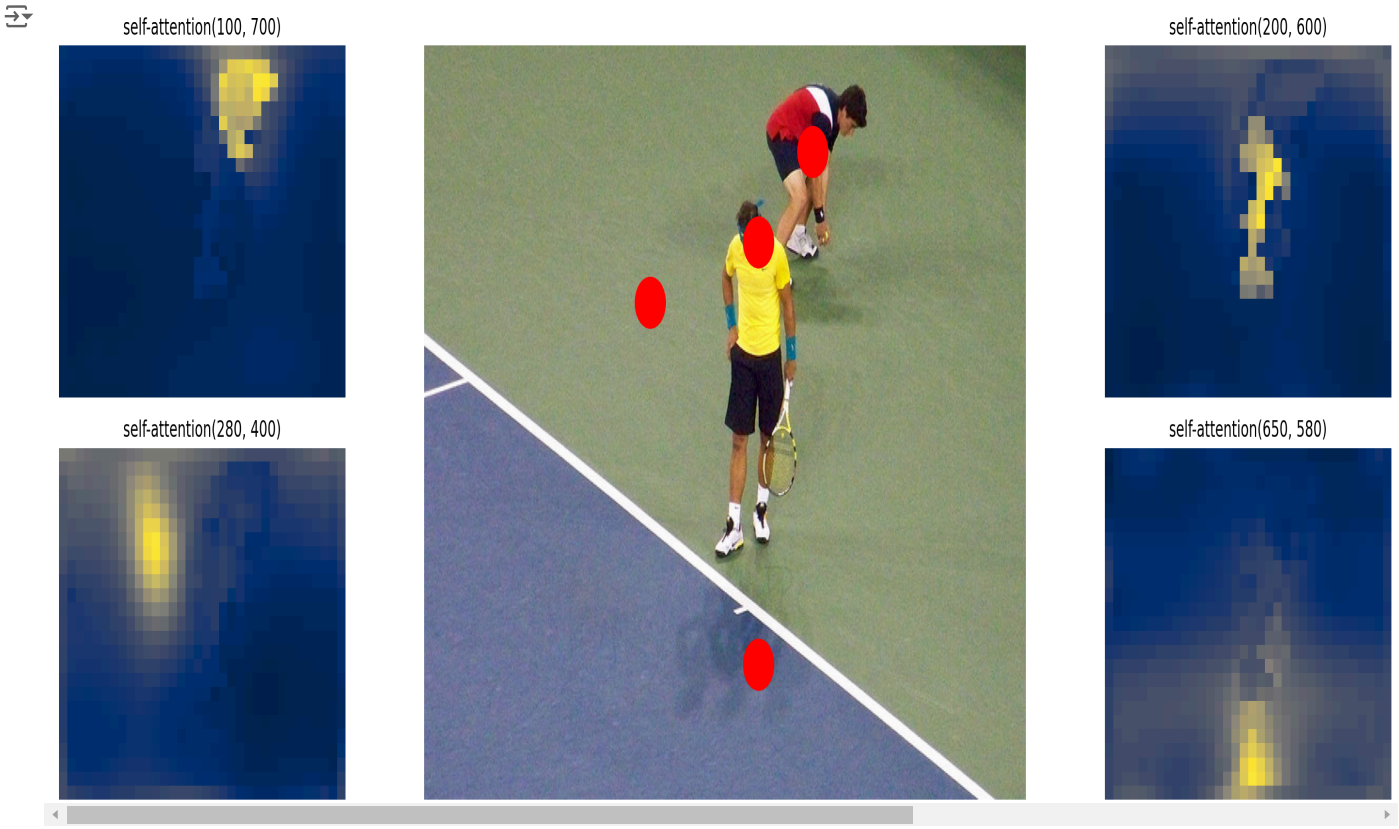
```
Encoder attention:       torch.Size([850, 850])
Feature map:             torch.Size([1, 2048, 25, 34])
```

```
# get the HxW shape of the feature maps of the CNN
shape = f_map.tensors.shape[-2:]
# and reshape the self-attention to a more interpretable shape
sattn = enc_attn_weights[0].reshape(shape + shape)
print("Reshaped self-attention:", sattn.shape)
```

```
Reshaped self-attention: torch.Size([25, 34, 25, 34])
```

```
# downsampling factor for the CNN, is 32 for DETR and 16 for DETR DC5
fact = 32

# let's select 4 reference points for visualization
idxs = [(100, 700), (280, 400), (200, 600), (650, 580),]

# here we create the canvas
fig = plt.figure(constrained_layout=True, figsize=(25 * 0.7, 8.5 * 0.7))
# and we add one plot per reference point
gs = fig.add_gridspec(2, 4)
axs = [
    fig.add_subplot(gs[0, 0]),
    fig.add_subplot(gs[1, 0]),
    fig.add_subplot(gs[0, -1]),
    fig.add_subplot(gs[1, -1]),
]
```

```
# for each one of the reference points, let's plot the self-attention
# for that point
for idx_o, ax in zip(idxs, axs):
    idx = (idx_o[0] // fact, idx_o[1] // fact)
    ax.imshow(sattn[..., idx[0], idx[1]], cmap='cividis', interpolation='nearest')
    ax.axis('off')
    ax.set_title(f'self-attention{idx_o}')

# and now let's add the central image, with the reference points as red circles
fcenter_ax = fig.add_subplot(gs[:, 1:-1])
fcenter_ax.imshow(im)
for (y, x) in idxs:
    scale = im.height / img.shape[-2]
    x = ((x // fact) + 0.5) * fact
    y = ((y // fact) + 0.5) * fact
    fcenter_ax.add_patch(plt.Circle((x * scale, y * scale), fact // 2, color='r'))
    fcenter_ax.axis('off')
```



Q3. Understanding Attention Mechanisms

In this task, you focus on understanding the attention mechanisms present in the encoder and decoder of DETR.

- Briefly describe the types of attention used in the encoder and decoder, and explain the key differences between them.
- Based on the visualized results from Q2, provide an analysis of the distinct characteristics of each attention mechanism in the encoder and decoder. Feel free to express your insights.

1) Encoder attention vs Decoder attention

Encoder only utilizes a self attention. For each query, which is a compressed information of image patch, attention mechanisms calculates the similarity with other image patches, aggregate the feature of each image patches and add it to the original query. This is the process of adaptively selecting reference feature to enhance the query feature.

On the other hand, decoder utilizes cross attention additionally. Let there are N queries, each of which having d dimension, thus having the total shape of $N \times d$ (learnable parameters). First, empty query slots are enhanced with position & relation between other query slots by self attention. Then, each slots attends to all the key values from image feature tokens and are added with aggregated results, by cross attention. This self-attention and cross-attention block is repeated for several times.

Using the image feature encoded from backbone architecture and tranformer encoder, each query predicts box and classifies the object, making a set. Since every query has different position, they might attend to different region in the image, thus will try to detect objects in different regions. However, as the decoder layer is stacked, there might be some dominant object query. Other queries might attend to dominant query, considering it as the reference query.

Therefore, there might be some "anchors" that model itself finds.

2) Analysis of Encoder attention mechanism

To analyze the encoder attention mechanism, we can check if high attention weights are assigned for similar part of image.

```
# downsampling factor for the CNN, is 32 for DETR and 16 for DETR DC5
fact = 32

# let's select 4 reference points for visualization
idxs = [(100, 700), (280, 400), (200, 600), (400, 600),]

# here we create the canvas
fig = plt.figure(constrained_layout=True, figsize=(25 * 0.7, 8.5 * 0.7))
# and we add one plot per reference point
gs = fig.add_gridspec(2, 4)
axs = [
    fig.add_subplot(gs[0, 0]),
    fig.add_subplot(gs[1, 0]),
    fig.add_subplot(gs[0, -1]),
    fig.add_subplot(gs[1, -1]),
]

# for each one of the reference points, let's plot the self-attention
# for that point
for idx_o, ax in zip(idxs, axs):
    idx = (idx_o[0] // fact, idx_o[1] // fact)
    ax.imshow(sattn[..., idx[0], idx[1]], cmap='cividis', interpolation='nearest')
    ax.axis('off')
    ax.set_title(f'self-attention{idx_o}')

# and now let's add the central image, with the reference points as red circles
fcenter_ax = fig.add_subplot(gs[:, 1:-1])
fcenter_ax.imshow(im)
for (y, x) in idxs:
    scale = im.height / img.shape[-2]
    x = ((x // fact) + 0.5) * fact
    y = ((y // fact) + 0.5) * fact
    fcenter_ax.add_patch(plt.Circle((x * scale, y * scale), fact // 2, color='r'))
    fcenter_ax.axis('off')
```

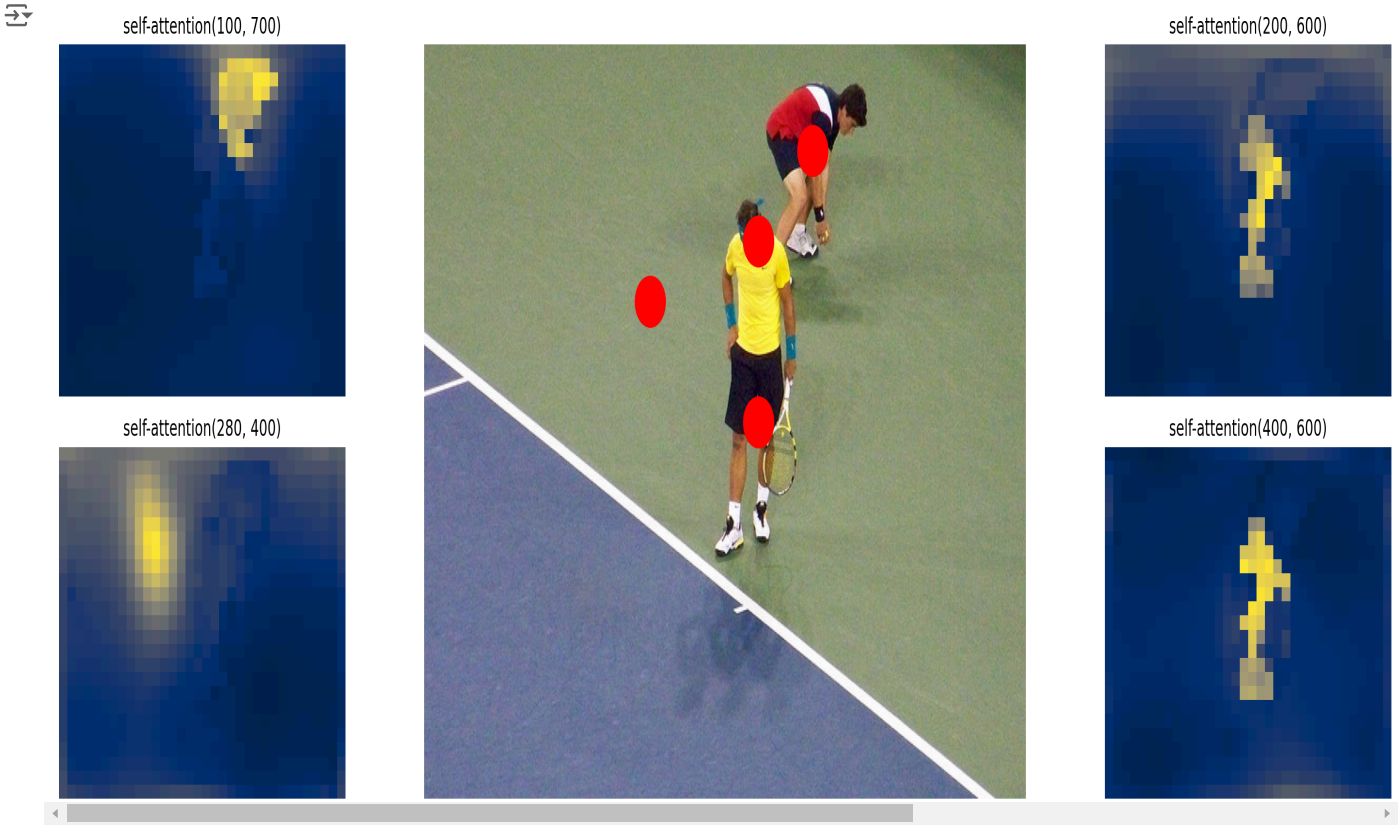


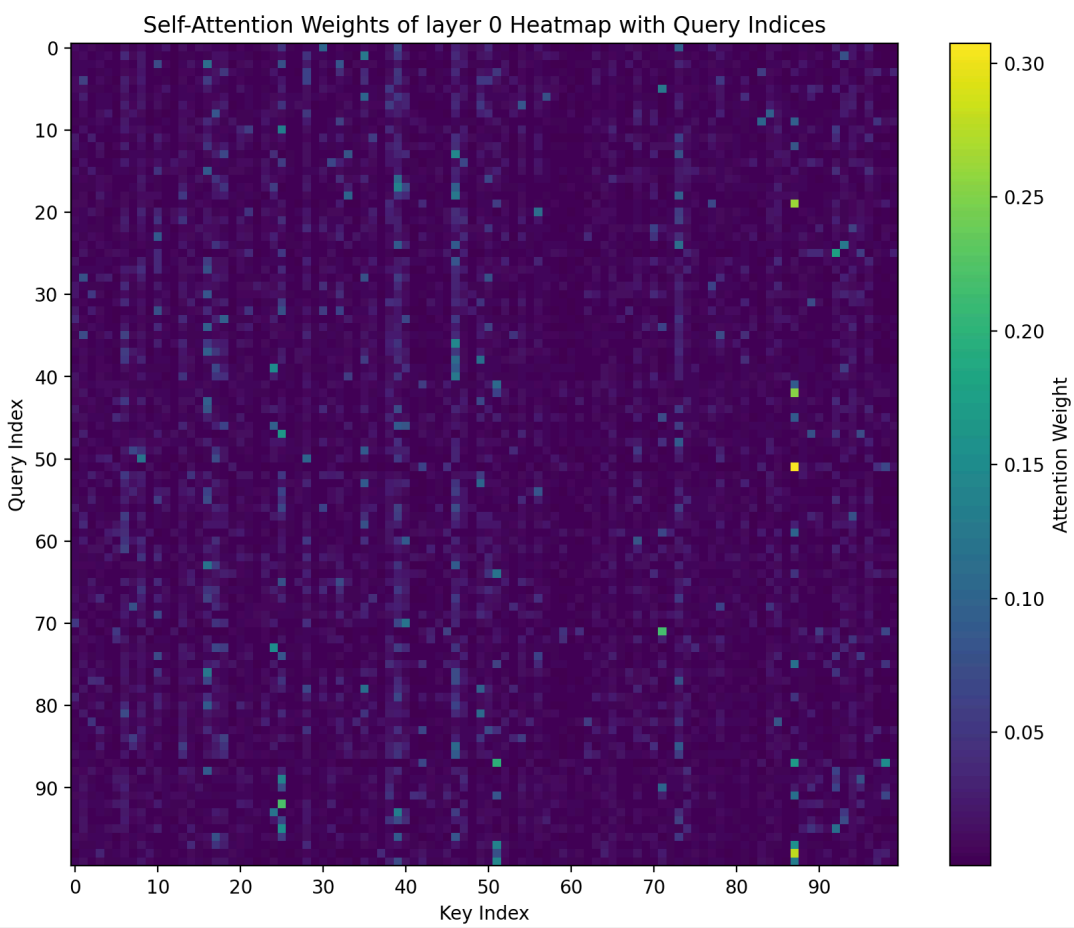
Image patch for 'person' with red shirts attends to the other area of same object. This is also valid for other objects. Self attention enhances the query feature, using highly correlated value patch.

3) Analysis of Decoder attention mechanism

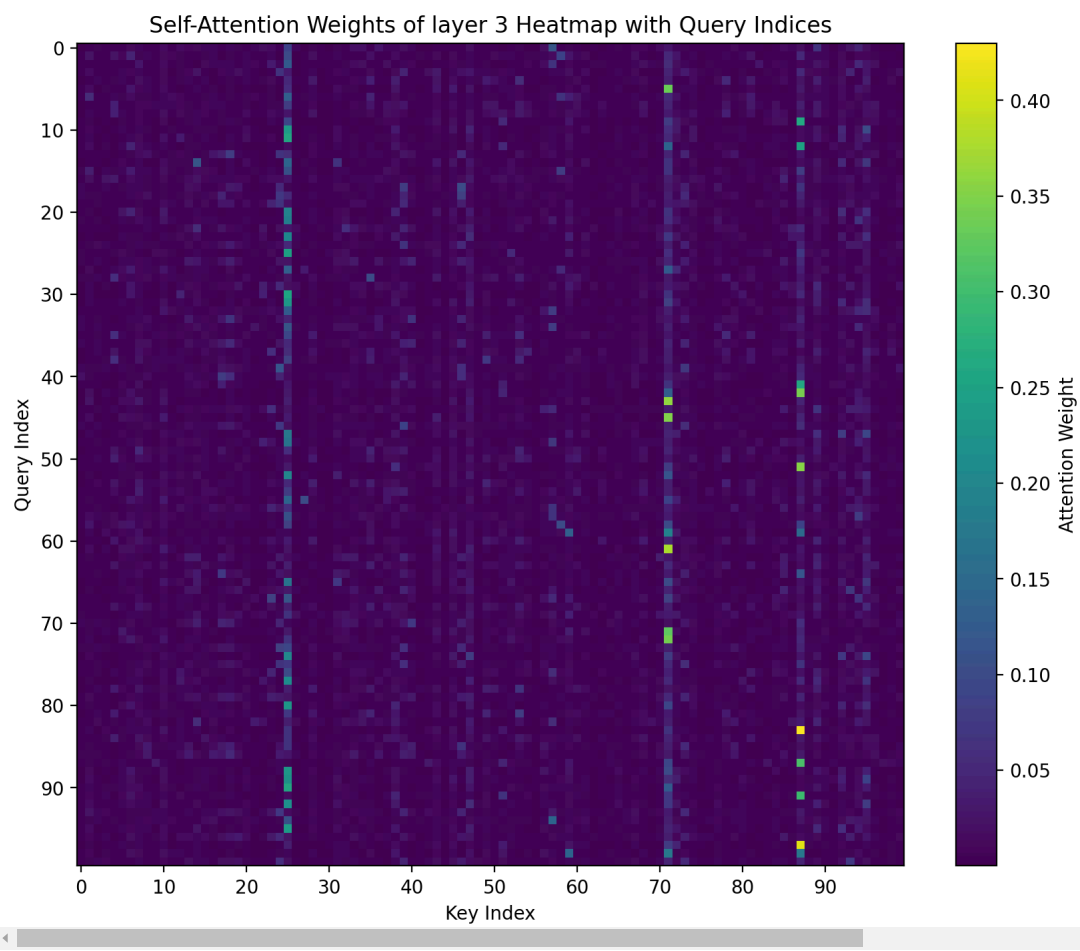
To analyze the decoder attention mechanism, we can check two things : i) How each query token attends to other token ii) How the query token attends to the region of each image.

<decoder query self attention>

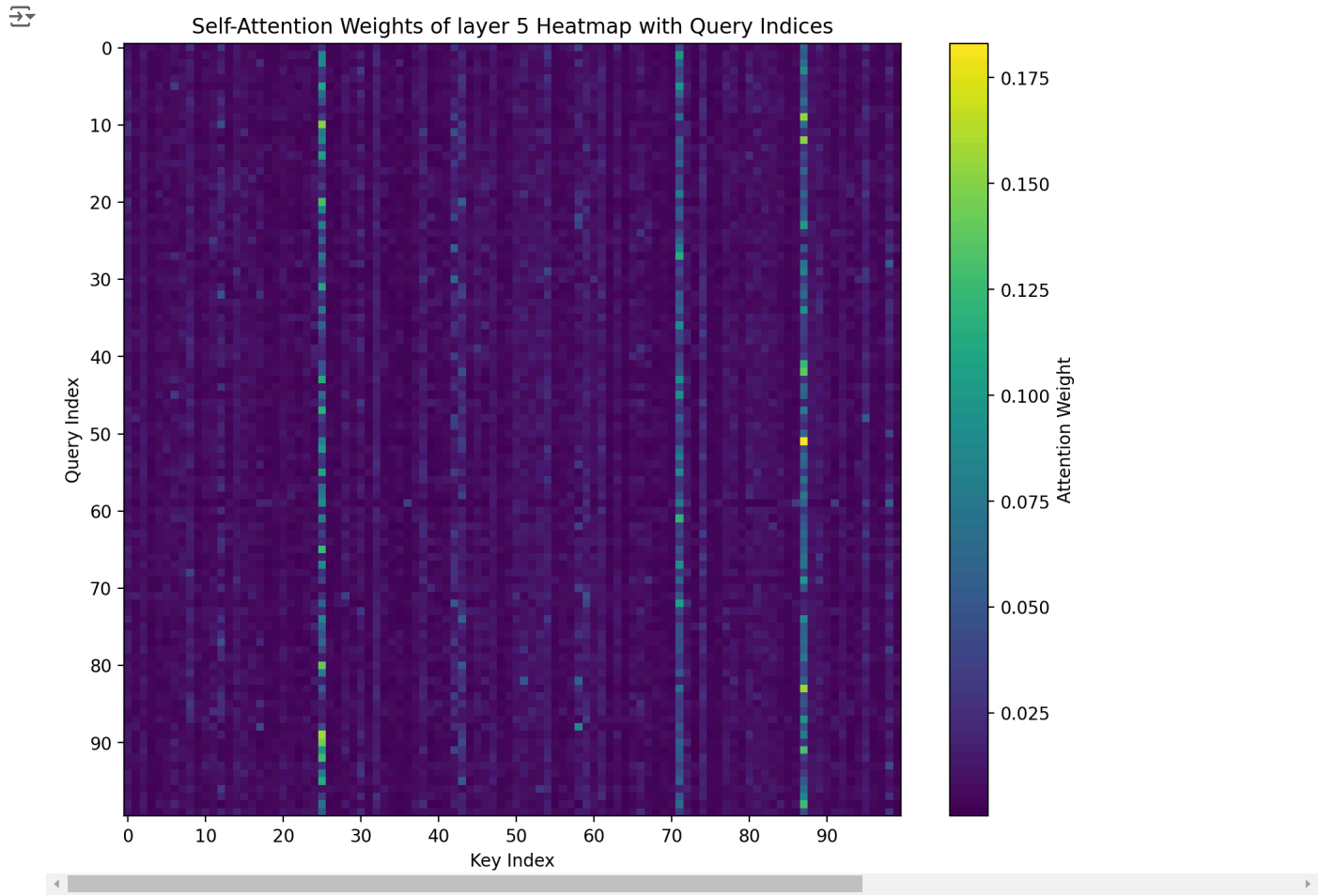
```
# use lists to store the outputs via up-values
dec_sattn_weights_layer0, dec_sattn_weights_layer3, dec_sattn_weights_layer5 = [], [], []

hooks2 = [
    model.transformer.decoder.layers[-5].self_attn.register_forward_hook(
        lambda self, input, output: dec_sattn_weights_layer0.append(output[1])
    ),

    model.transformer.decoder.layers[-3].self_attn.register_forward_hook(
        lambda self, input, output: dec_sattn_weights_layer3.append(output[1])
    ),

    model.transformer.decoder.layers[-1].self_attn.register_forward_hook(
        lambda self, input, output: dec_sattn_weights_layer5.append(output[1])
    ),
]

# propagate through the model
outputs = model(img) # put your own image

for hook in hooks2:
    hook.remove()
dec_sattn_weights_layer0 = dec_sattn_weights_layer0[0]
dec_sattn_weights_layer3 = dec_sattn_weights_layer3[0]
dec_sattn_weights_layer5 = dec_sattn_weights_layer5[0]


# plot the self-attention weight of decoder layer
weights = dec_sattn_weights_layer0[0]

plt.figure(figsize=(10, 8))
plt.imshow(weights, cmap='viridis', aspect='auto')
plt.colorbar(label='Attention Weight')

# add query number
plt.xlabel('Key Index')
plt.ylabel('Query Index')
plt.title('Self-Attention Weights of layer 0 Heatmap with Query Indices')

# Query 및 Key 인덱스 레이블 설정
plt.xticks(range(0, 100, 10))
plt.yticks(range(0, 100, 10))

plt.show()
```




```
# plot the self-attention weight of decoder layer
weights = dec_sattn_weights_layer3[0]

plt.figure(figsize=(10, 8))
plt.imshow(weights, cmap='viridis', aspect='auto')
plt.colorbar(label='Attention Weight')

# add query number
plt.xlabel('Key Index')
plt.ylabel('Query Index')
plt.title('Self-Attention Weights of layer 3 Heatmap with Query Indices')

# Query 및 Key 인덱스 레이블 설정
plt.xticks(range(0, 100, 10))
plt.yticks(range(0, 100, 10))

plt.show()
```




```
# plot the self-attention weight of decoder layer
weights = dec_sattn_weights_layer5[0]

plt.figure(figsize=(10, 8))
plt.imshow(weights, cmap='viridis', aspect='auto')
plt.colorbar(label='Attention Weight')

# add query number
plt.xlabel('Key Index')
plt.ylabel('Query Index')
plt.title('Self-Attention Weights of layer 5 Heatmap with Query Indices')

# Query 및 Key 인덱스 레이블 설정
plt.xticks(range(0, 100, 10))
plt.yticks(range(0, 100, 10))

plt.show()
```



Note that, attention score is very low even though key and query are from same token. This is because W_k and W_q differs, which enables the model to focus on "important information" rather than just focusing on "similar information".

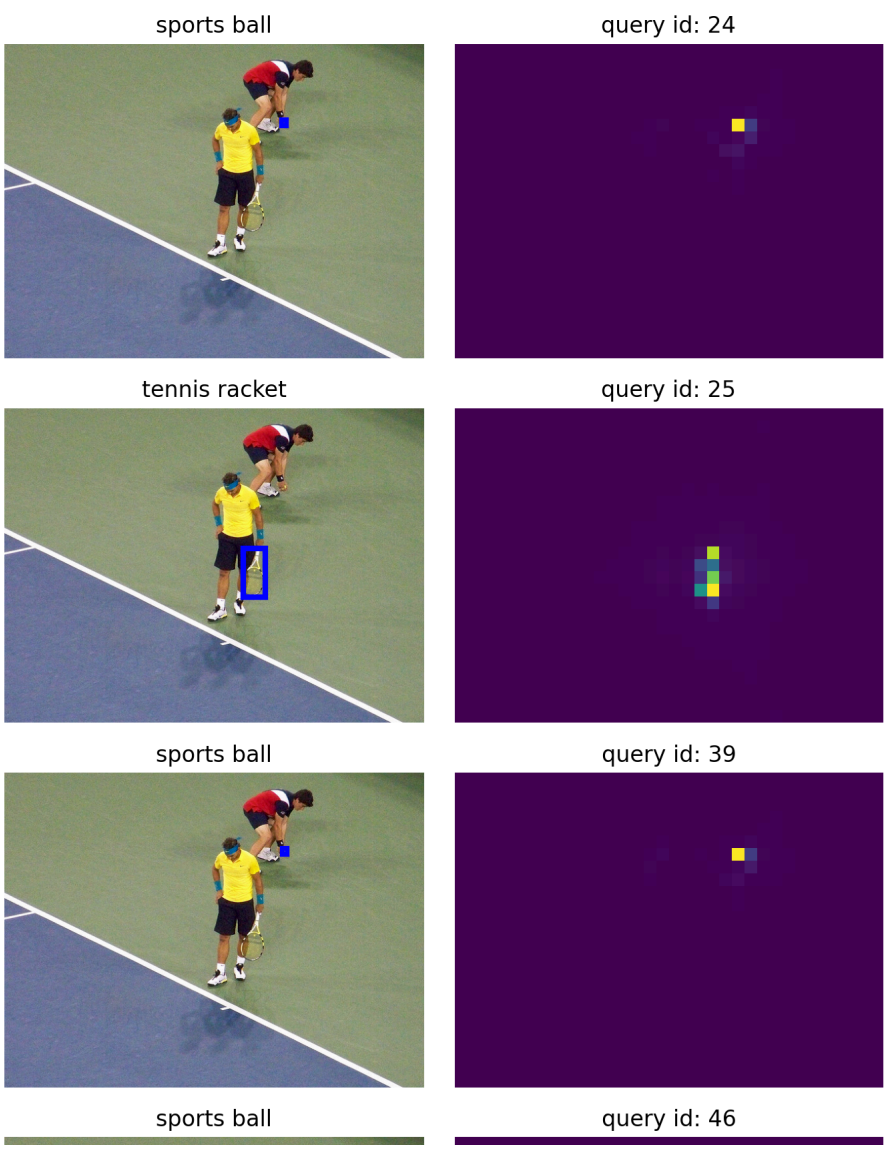
As the layer goes deeper, we can observe that almost every queries have high attention score to Key = {25, 71, 81}. This means, those keys are used as "anchors" or as important reference point, when predicting the bounding box and probability.

<comparing the preidctions from relevent token set>

- keep : if ith token has confidence prediction, True

Almost every object queries predict similar objects. This can be considered as the evidence of anchoring effect.

```
keep2 = probas.max(-1).values >= 0.03
bboxes_scaled2 = rescale_bboxes(outputs['pred_boxes'][0, keep2], im.size)


# get the feature map shape
h, w = conv_features['0'].tensors.shape[-2:]

fig, axs = plt.subplots(ncols=2, nrows=len(bboxes_scaled2), figsize=(7, 22))
colors = COLORS * 100
for idx, ax_i, (xmin, ymin, xmax, ymax) in zip(keep2.nonzero(), axs, bboxes_scaled2):
    ax = ax_i[0]
    ax.imshow(im)
    ax.add_patch(plt.Rectangle((xmin, ymin), xmax - xmin, ymax - ymin,
                               fill=False, color='blue', linewidth=3))
    ax.axis('off')
    ax.set_title(CLASSES[probas[idx].argmax()])
    ax = ax_i[1]
    ax.imshow(dec_attn_weights[0, idx].view(h, w))
    ax.axis('off')
    ax.set_title(f'query id: {idx.item()}')
fig.tight_layout()
```


sports ball
query id: 24
tennis racket
query id: 25
sports ball
query id: 39
sports ball
query id: 46