

## ▼ Homework 3

2022320009 이수현

이미지는 픽사베이의 무료 이미지를 사용하였습니다

### 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.

```

1 from torch import nn
2 class DETR(nn.Module):
3     def __init__(self, num_classes, hidden_dim=256, nheads=8,
4                  num_encoder_layers=6, num_decoder_layers=6, num_queries=100):
5         super().__init__()
6
7         # create ResNet-50 backbone
8         self.backbone = resnet50()
9         del self.backbone.fc
10
11        # create conversion layer
12        self.conv = nn.Conv2d(2048, hidden_dim, 1)
13
14        # create a default PyTorch transformer
15        self.transformer = nn.Transformer(
16            hidden_dim, nheads, num_encoder_layers, num_decoder_layers)
17
18        # prediction heads, one extra class for predicting non-empty slots
19        # note that in baseline DETR linear_bbox layer is 3-layer MLP
20        self.linear_class = nn.Linear(hidden_dim, num_classes + 1) #모델이 예측해야하는 총 class수. objet가 없는 :
21        self.linear_bbox = nn.Linear(hidden_dim, 4)
22
23        # output positional encodings (object queries)
24        self.query_pos = nn.Parameter(torch.rand(num_queries, hidden_dim)) #
25
26        # spatial positional encodings
27        # note that in baseline DETR we use sine positional encodings
28        self.row_embed = nn.Parameter(torch.rand(50, hidden_dim // 2))
29        self.col_embed = nn.Parameter(torch.rand(50, hidden_dim // 2))
30
31    def forward(self, inputs):
32        # propagate inputs through ResNet-50 up to avg-pool layer
33        x = self.backbone.conv1(inputs)
34        x = self.backbone.bn1(x)
35        x = self.backbone.relu(x)
36        x = self.backbone.maxpool(x)
37
38        x = self.backbone.layer1(x)
39        x = self.backbone.layer2(x)
40        x = self.backbone.layer3(x)
41        x = self.backbone.layer4(x)
42

```

```

43      # convert from 2048 to 256 feature planes for the transformer
44      h = self.conv(x)
45
46      # construct positional encodings
47      H, W = h.shape[-2:]
48      pos = torch.cat([
49          self.col_embed[:W].unsqueeze(0).repeat(H, 1, 1),
50          self.row_embed[:H].unsqueeze(1).repeat(1, W, 1),
51      ], dim=-1).flatten(0, 1).unsqueeze(1)
52
53      # propagate through the transformer
54      h = self.transformer(pos + 0.1 * h.flatten(2).permute(2, 0, 1),
55                           self.query_pos.unsqueeze(1)).transpose(0, 1)
56
57
58
59      # finally project transformer outputs to class labels and bounding boxes
60      pred_logits = _____
61      pred_boxes = _____
62
63      return {'pred_logits': pred_logits,
64              'pred_boxes': pred_boxes}

```

## takeout messages

[DETR]

- anchor box 없애고 단순한 구조 -> 모델이 더 가벼워지고 빠름
- end to end 구조
  - 기존에는 anchor box -> 후처리(NMS)가 필요
  - end to end 구조로 더 단순해짐
- transformer 기반 구조
  - self attention과 cross attention(encoder-decoder attention)을 이용하여 전반적인 관계와 각 객체간의 연관성 학습
- binary matching에 헝가리안 알고리즘을 사용
- **init** 함수
  - 주요 구조 : 백본 / transformer encoder-decoder / 예측헤드
  - ResNet-50 backbone으로 feature extraction
  - CNN으로 feature를 변환 : self.conv = nn.Conv2d(2048, hidden\_dim, 1)
  - transformer : 인코더와 디코더로 구성
  - 예측 헤드 : class 예측, BBOX 예측 -> 선형 계층 self.linear\_class = nn.Linear(hidden\_dim, num\_classes + 1)  
self.linear\_bbox = nn.Linear(hidden\_dim, 4)
- forward
  - 백본으로 ResNet-50을 통해 이미지에서 특징 추출
  - 특징맵 -> CNN -> transformer input : h = self.conv(x) : 2048차원에서 -> 256차원으로 변환
  - positional encoding 구성
  - transformer
  - 출력 예측 (BBOX, class)

## ▼ 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.

```

1 import math
2
3 from PIL import Image
4 import requests
5 import matplotlib.pyplot as plt
6 %config InlineBackend.figure_format = 'retina'
7
8 import ipywidgets as widgets
9 from IPython.display import display, clear_output
10
11 import torch
12 from torch import nn
13
14
15 from torchvision.models import resnet50
16 import torchvision.transforms as T
17 torch.set_grad_enabled(False);
18
19 # COCO classes
20 CLASSES = [
21     'N/A', 'person', 'bicycle', 'car', 'motorcycle', 'airplane', 'bus',
22     'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'N/A',
23     'stop sign', 'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse',
24     'sheep', 'cow', 'elephant', 'bear', 'zebra', 'giraffe', 'N/A', 'backpack',
25     'umbrella', 'N/A', 'N/A', 'handbag', 'tie', 'suitcase', 'frisbee', 'skis',
26     'snowboard', 'sports ball', 'kite', 'baseball bat', 'baseball glove',
27     'skateboard', 'surfboard', 'tennis racket', 'bottle', 'N/A', 'wine glass',
28     'cup', 'fork', 'knife', 'spoon', 'bowl', 'banana', 'apple', 'sandwich',
29     'orange', 'broccoli', 'carrot', 'hot dog', 'pizza', 'donut', 'cake',
30     'chair', 'couch', 'potted plant', 'bed', 'N/A', 'dining table', 'N/A',
31     'N/A', 'toilet', 'N/A', 'tv', 'laptop', 'mouse', 'remote', 'keyboard',
32     'cell phone', 'microwave', 'oven', 'toaster', 'sink', 'refrigerator', 'N/A',
33     'book', 'clock', 'vase', 'scissors', 'teddy bear', 'hair drier',
34     'toothbrush'
35 ]
36
37 # colors for visualization
38 COLORS = [[0.000, 0.447, 0.741], [0.850, 0.325, 0.098], [0.929, 0.694, 0.125],
39             [0.494, 0.184, 0.556], [0.466, 0.674, 0.188], [0.301, 0.745, 0.933]]
40 # standard PyTorch mean-std input image normalization
41 transform = T.Compose([
42     T.Resize(800),
43     T.ToTensor(),
44     T.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
45 ])
46
47 # for output bounding box post-processing
48 def box_cxcywh_to_xyxy(x):
49     x_c, y_c, w, h = x.unbind(1)
50     b = [(x_c - 0.5 * w), (y_c - 0.5 * h),
51           (x_c + 0.5 * w), (y_c + 0.5 * h)]
52     return torch.stack(b, dim=1)
53
54 def rescale_bboxes(out_bbox, size):
55     img_w, img_h = size
56     b = box_cxcywh_to_xyxy(out_bbox)
57     b = b * torch.tensor([img_w, img_h, img_w, img_h], dtype=torch.float32)
58     return b
59
60 def plot_results(pil_img, prob, boxes):

```

```

61 plt.figure(figsize=(16,10))
62 plt.imshow(pil_img)
63 ax = plt.gca()
64 colors = COLORS * 100
65 for p, (xmin, ymin, xmax, ymax), c in zip(prob, boxes.tolist(), colors):
66     ax.add_patch(plt.Rectangle((xmin, ymin), xmax - xmin, ymax - ymin,
67                               fill=False, color=c, linewidth=3))
68     cl = p.argmax()
69     text = f'{CLASSES[cl]}: {p[cl]:0.2f}'
70     ax.text(xmin, ymin, text, fontsize=15,
71             bbox=dict(facecolor='yellow', alpha=0.5))
72 plt.axis('off')
73 plt.show()
74
75

```

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.

```

1 model = torch.hub.load('facebookresearch/detr', 'detr_resnet50', pretrained=True)
2 model.eval();
3
4 #원래 코드 url = 'http://images.cocodataset.org/val2017/000000039769.jpg'
5 url = 'https://media.istockphoto.com/id/2148372432/ko/%EC%82%AC%EC%A7%84/%EA%B1%B0%EC%8B%A4-%EC%86%8C%ED%8C%8C%
6 #이미지 출처는 픽사베이(무료이미지)
7 im = Image.open(requests.get(url, stream=True).raw) # put your own image
8
9 # mean-std normalize the input image (batch-size: 1)
10 img = transform(im).unsqueeze(0)
11
12 # propagate through the model
13 outputs = model(img)
14
15 # keep only predictions with 0.7+ confidence
16 probas = outputs['pred_logits'].softmax(-1)[0, :, :-1]
17 keep = probas.max(-1).values > 0.9
18
19 # convert boxes from [0: 1] to image scales
20 bboxes_scaled = rescale_bboxes(outputs['pred_boxes'][0, keep], im.size)
21
22 # mean-std normalize the input image (batch-size: 1)
23 img = transform(im).unsqueeze(0)
24
25 # propagate through the model
26 outputs = model(img)
27
28 # keep only predictions with 0.7+ confidence
29 probas = outputs['pred_logits'].softmax(-1)[0, :, :-1]
30 keep = probas.max(-1).values > 0.9
31
32 # convert boxes from [0: 1] to image scales
33 bboxes_scaled = rescale_bboxes(outputs['pred_boxes'][0, keep], im.size)
34
35 # mean-std normalize the input image (batch-size: 1)
36 img = transform(im).unsqueeze(0)
37
38 # propagate through the model
39 outputs = model(img)
40
41 # keep only predictions with 0.7+ confidence
42 probas = outputs['pred_logits'].softmax(-1)[0, :, :-1]
43 keep = probas.max(-1).values > 0.9
44
45 # convert boxes from [0: 1] to image scales

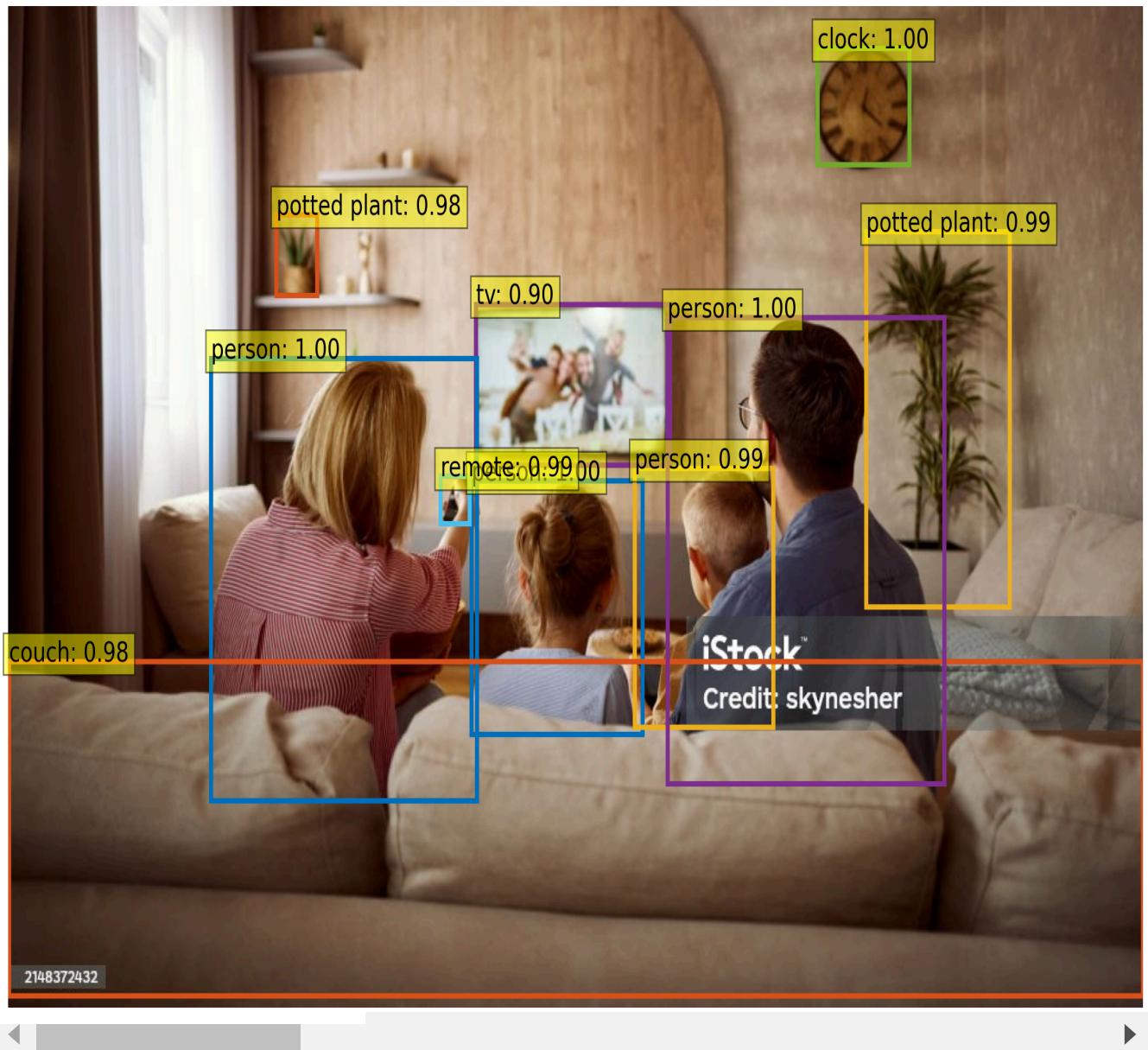
```

```

46 bboxes_scaled = rescale_bboxes(outputs['pred_boxes'][0, keep], im.size)
47
48 plot_results(im, probas[keep], bboxes_scaled)

```

→ Downloading: "<https://github.com/facebookresearch/detr/zipball/main>" to /root/.cache/torch/hub/main.zip  
 /usr/local/lib/python3.10/dist-packages/torchvision/models/\_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated  
 warnings.warn(  
 /usr/local/lib/python3.10/dist-packages/torchvision/models/\_utils.py:223: UserWarning: Arguments other than a weight enum or `N  
 warnings.warn(msg)  
 Downloading: "<https://download.pytorch.org/models/resnet50-0676ba61.pth>" to /root/.cache/torch/hub/checkpoints/resnet50-0676ba61  
 100%|██████████| 97.8M/97.8M [00:00<00:00, 202MB/s]  
 Downloading: "<https://dl.fbaipublicfiles.com/detr/detr-r50-e632da11.pth>" to /root/.cache/torch/hub/checkpoints/detr-r50-e632da1  
 100%|██████████| 159M/159M [00:01<00:00, 163MB/s]



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.

```

1 # use lists to store the outputs via up-values
2 conv_features, enc_attn_weights, dec_attn_weights = [], [], []
3
4 hooks = [
5     model.backbone[-2].register_forward_hook(
6         lambda self, input, output: conv_features.append(output)
7     ),
8     model.transformer.encoder.layers[-1].self_attn.register_forward_hook(
9         lambda self, input, output: enc_attn_weights.append(output[1])
10    ),

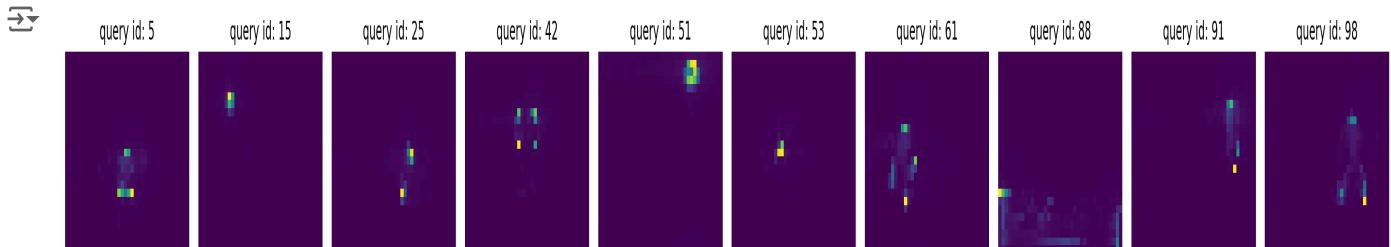
```

```

11     model.transformer.decoder.layers[-1].multihead_attn.register_forward_hook(
12         lambda self, input, output: dec_attn_weights.append(output[1]))
13     ),
14 ]
15
16 # propagate through the model
17 outputs = model(img) # put your own image
18
19 for hook in hooks:
20     hook.remove() #inference 이후에는 hook 제거. 불필요한 연산 제거
21
22 # don't need the list anymore
23 conv_features = conv_features[0]
24 enc_attn_weights = enc_attn_weights[0]
25 dec_attn_weights = dec_attn_weights[0]

1 # get the feature map shape
2 h, w = conv_features['0'].tensors.shape[-2:]
3
4 fig, axs = plt.subplots(ncols=len(bboxes_scaled), nrows=2, figsize=(22, 7))
5 colors = COLORS * 100
6 for idx, ax_i, (xmin, ymin, xmax, ymax) in zip(keep.nonzero(), axs.T, bboxes_scaled):
7     ax = ax_i[0]
8     ax.imshow(dec_attn_weights[0, idx].view(h, w))
9     ax.axis('off')
10    ax.set_title(f'query id: {idx.item()}')
11    ax = ax_i[1]
12    ax.imshow(im)
13    ax.add_patch(plt.Rectangle((xmin, ymin), xmax - xmin, ymax - ymin,
14                               fill=False, color='blue', linewidth=3))
15    ax.axis('off')
16    ax.set_title(CLASSES[probas[idx].argmax()])
17 fig.tight_layout()

```



## 디코더 attention

### 1. self attention

- 디코더가 객체 쿼리들 사이의 상호작용을 학습
- 객체 쿼리들이 서로 독립적으로 작용하지 않음
- 각 쿼리가 예측하려는 객체가 중복되지 않음

### 2. encoder - decoder attention

- 디코더의 객체 쿼리가 인코더가 출력한 특징 맵과 상호작용
- 디코더의 각 쿼리는 인코더의 특징 맵에서 자신이 집중해야 하는 부분에 대한 정보를 출력
- 이 과정에서 디코더가 특정 개체에 집중하여 BBOX와 class를 예측

## 디코더 attention 시각화 분석

- 위의 보라색 이미지
  - 각각의 쿼리 ID의 디코더 attention 가중치를 시각화
  - 밝은 부분(노란색)이 디코더가 해당 객체에 집중하고 있는 부분
  - id 25는 person에 집중하고 있는데 위 아래 사진을 비교해보면 위의 사진에서 밝은 부분이 사람
- 아래의 이미지
  - 디코더의 예측결과로, 각 쿼리 ID마다 특정 개체에 집중하여 BBOX와 class를 출력
  - person, potted plant, tv, clock, remote, couch을 분류
- 특정 객체 쿼리에 대해 이미지 내에서 어떤 영역에 집중하는지를 나타낸다
- 디코더의 self attention으로 각각의 개체가 중복되지 않고 탐지
- 각 쿼리 id에 따라 디코더는 이미지에서 특정 객체에 집중하며 각 객체에 파란색 BBOX와 class를 분류
- 각 id별로 위의 보라색 이미지와 아래의 이미지를 매칭해보면, 각 id의 객체를 탐지하는데 노란색 부분이 가장 큰 도움을 주었다고 해석할 수 있다.
- id 25를 예로 들면 노란색 부분을 보고 'person'이라고 판단했다고 이해 (다른 부분도 봤지만 노란색 부분이 가장 많은 도움이 되었다 정도)

```

1 # output of the CNN
2 f_map = conv_features['0']
3 print("Encoder attention:      ", enc_attn_weights[0].shape) # => 인코더의 self attention 가중치! => 얘를 통해 ?
4 print("Feature map:            ", f_map.tensors.shape)

→ Encoder attention:      torch.Size([950, 950])
  Feature map:            torch.Size([1, 2048, 25, 38])

```

```

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

→ Reshaped self-attention: torch.Size([25, 38, 25, 38])

```

```

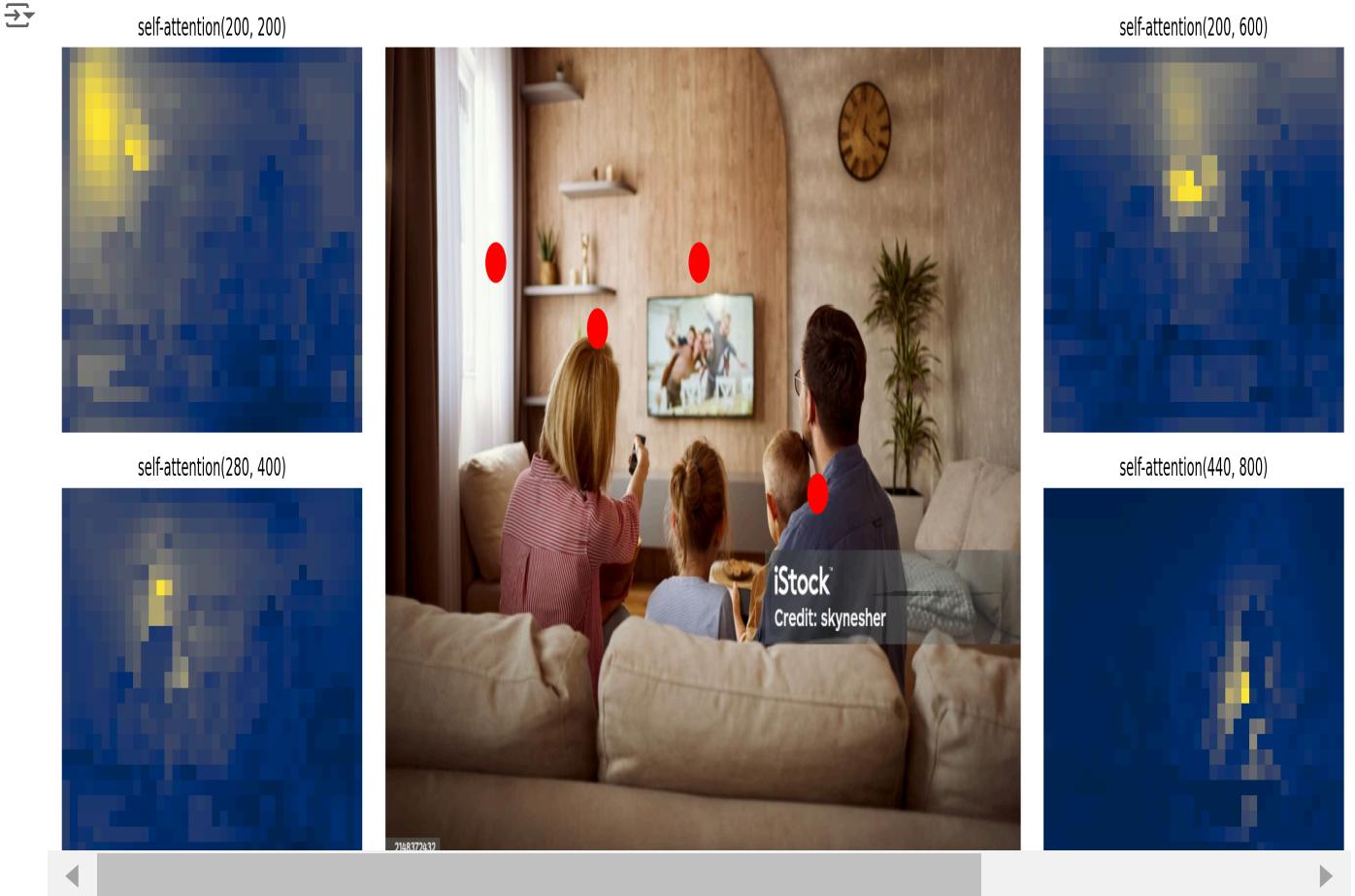
1 # downsampling factor for the CNN, is 32 for DETR and 16 for DETR DC5
2 fact = 32 #다운샘플링 feature map 크기가 32배 작아짐
3
4 # let's select 4 reference points for visualization
5 idxs = [(200, 200), (280, 400), (200, 600), (440, 800),] # => 인코더의 self attention이 집중하는 영역을 시각화함
6
7 # here we create the canvas
8 fig = plt.figure(constrained_layout=True, figsize=(25 * 0.7, 8.5 * 0.7))
9 # and we add one plot per reference point
10 gs = fig.add_gridspec(2, 4)
11 axs = [
12     fig.add_subplot(gs[0, 0]),
13     fig.add_subplot(gs[1, 0]),
14     fig.add_subplot(gs[0, -1]),
15     fig.add_subplot(gs[1, -1]),
16 ]
17
18 # for each one of the reference points, let's plot the self-attention
19 # for that point
20 # 인코더 셀프 어텐션 시각화

```

```

21 for idx_o, ax in zip(idxs, axs):
22     idx = (idx_o[0] // fact, idx_o[1] // fact)
23     ax.imshow(sattn[..., idx[0], idx[1]], cmap='cividis', interpolation='nearest')
24     ax.axis('off')
25     ax.set_title(f'self-attention{idx_o}')
26
27 # and now let's add the central image, with the reference points as red circles
28 fcenter_ax = fig.add_subplot(gs[:, 1:-1])
29 fcenter_ax.imshow(im)
30 for (y, x) in idxs:
31     scale = im.height / img.shape[-2]
32     x = ((x // fact) + 0.5) * fact
33     y = ((y // fact) + 0.5) * fact
34     fcenter_ax.add_patch(plt.Circle((x * scale, y * scale), fact // 2, color='r'))
35     fcenter_ax.axis('off')

```



## 인코더 attention

### 1. self attention

- 디코더가 객체 쿼리들 사이의 상호작용을 학습
- 객체 쿼리들이 서로 독립적으로 작용하지 않음
- 각 쿼리가 예측하려는 객체가 중복되지 않음

## 인코더의 self attention 분석

- fact = 32 : 실제 이미지보다 32배 작아짐
- 좌우의 파랑노랑 사진들 : 참조포인트에 대한 self attention 가중치를 시각화
  - 노랑부분에 attention이 강하게 작용
  - 인코더가 해당 참조 포인트가 특정 이미지 영역과 관련 있다고 판단하는 부분
- 중간 이미지
  - 빨간색 원 : 참조 포인트
  - 각 참조포인트에 대해 self attention 가중치가 계산됨

- (200,600)의 경우 원이 tv 근처에 위치해서 왼쪽 위의 self attention(200,600)에서 tv가 있는 곳이 노란색으로 표시됨

- 인코더는 이미지의 전반적인 문맥을 이해
- 각 참조포인트의 노란색 부분은 인코더가 해당 위치와 관련된 특징을 학습했다고 이해
- 즉, (280,400)의 경우 사람의 머리, 주변 환경등 참조포인트의 위치에 대한 특징을 학습함

## ▼ 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. attention mechanisms of encoder

- 입력 이미지의 feature map을 처리할 때 self attention을 이용
- 입력 sequence의 각 요소가 다른 모든 요소와의 관계를 학습하여 전반적인 정보를 파악
- 즉, 인코더는 이런 매커니즘을 통해 이미지 전체의 특징을 종합적으로 학습할 수 있다.

### 2. attention mechanisms of decoder

- self attention
  - 디코더가 객체 쿼리들 사이의 상호작용을 학습
  - 객체 쿼리들이 서로 독립적으로 작용하지 않음
  - 각 쿼리가 예측하려는 객체가 중복되지 않음
- encoder - decoder attention
  - 디코더의 객체 쿼리가 인코더가 출력한 특징 맵과 상호작용
  - 디코더의 각 쿼리는 인코더의 특징 맵에서 자신이 집중해야 하는 부분에 대한 정보를 출력
  - 이 과정에서 디코더가 특정 객체에 집중하여 BBOX와 class를 예측

### 3. encoder와 decoder의 attention 차이점

- 인코더는 self attention만 사용하고 디코더는 self attention과 encoder-decoder attention을 사용
- 인코더는 self attention만 사용 -> 전체 입력 이미지의 특징을 학습. 이미지의 특징을 요약하고 변환
- 디코더의 self attention -> 쿼리 사이의 관계를 학습. 서로의 관계를 이해하여 객체의 중복이 없음
- 디코더의 encoder-decoder attention -> 디코더의 쿼리가 인코더의 특징 맵에서 특정 객체와 관련된 정보를 추출
- 인코더의 목적은 전반적인 특징을 학습하고 요약하여 디코더에 전달하는 것이고
- 디코더는 인코더에게 받은 특징을 기반으로 object detection하여 class와 BBOX 예측

### 4. Based on the visualized results from Q2, provide an analysis of the distinct characteristics of each attention mechanism in the encoder and decoder

-> 각 시각화 결과 아래에 분석했습니다

