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.

In [1]:
```
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)

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

Discussion and Takeaway Messages

Overview of the DETR Model

The DETR (DEtection TRansformer) model uses a Transformer-based architecture for object detection tasks, moving away from traditional CNN-based architectures. The key components of the DETR workflow are:

- **Backbone**: A CNN model like ResNet-50 is used as a feature extractor, removing the final fully connected layer (`fc`) to output feature maps.
- **Transformer**: The core of the DETR model is the transformer, which takes these features and processes them for object detection.
- **Prediction Heads**: These heads predict both class labels and bounding box coordinates from the output of the transformer.
- **Positional Encoding**: DETR employs both spatial and object queries for encoding positional information, which is crucial for the transformer to learn object relationships.

Key Takeaways

- **End-to-End Model**: DETR is an end-to-end model that eliminates the need for anchor boxes and NMS (Non-Maximum Suppression), which are typically required in traditional object detection pipelines. This makes the model simpler and more efficient to train.

- **Transformer for Object Detection**: The use of transformers in object detection is a novel approach that enables DETR to capture global dependencies in an image, which is often difficult for CNNs alone. This leads to better object relationships and more accurate detection.

- **Positional Encoding**:

  - **Spatial Positional Encoding**: DETR uses sine and cosine functions for row and column embeddings, which are essential to preserve spatial information of objects within the image.
  - **Query Positional Encoding**: The object queries are used to focus on specific parts of the image, with each query attending to a different object.
- **Advantages over Traditional Models**:

  - **No Anchor Boxes**: Traditional models rely on hand-crafted anchor boxes, which can be complex to tune. DETR avoids this by learning object detection directly from the data, relying on a fixed set of object queries.
  - **Global Context**: The transformer's ability to attend to all parts of the image simultaneously gives it a unique advantage in modeling relationships across objects in the scene.
- **Challenges**:

  - **Training Time**: DETR typically requires a longer training time compared to traditional models due to the complexity of the transformer-based architecture.
  - **Performance on Small Objects**: While DETR performs well for large objects, it can sometimes struggle with small objects or those that are occluded.

Conclusion

The DETR model is a significant step forward in simplifying object detection by replacing convolutions and region proposal networks (RPNs) with a transformer. Although the model has some challenges, such as requiring longer training times, its end-to-end nature and ability to capture long-range dependencies in images are promising features that could lead to further advancements in object detection.

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.

In [2]:
```
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):
```

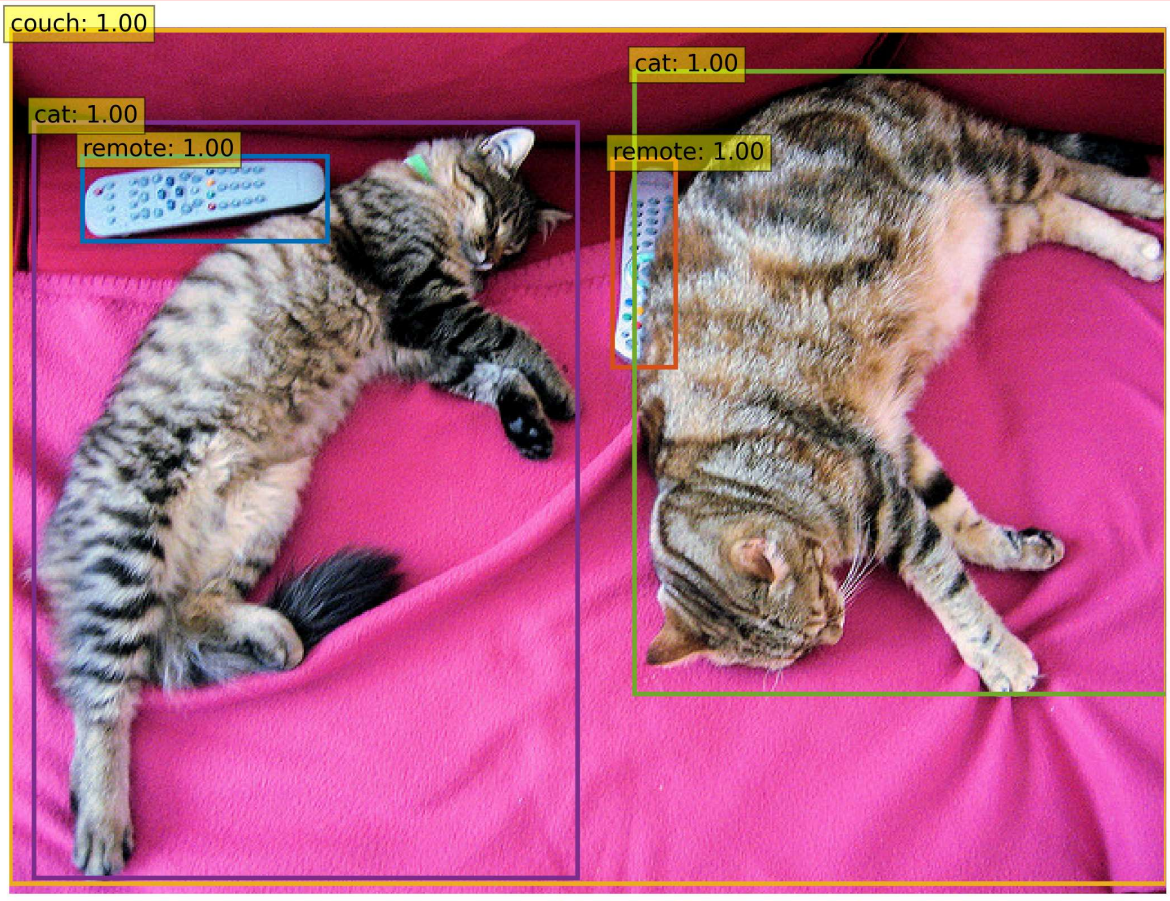
```
    x_c, y_c, w, h = x.unbind(1)
    b = [(x_c - 0.5 * w), (y_c - 0.5 * h),
         (x_c + 0.5 * w), (y_c + 0.5 * h)]
    return torch.stack(b, dim=1)

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.

In [3]:
```
model = torch.hub.load('facebookresearch/detr', 'detr_resnet50', pretrained=True
model.eval();

url = 'http://images.cocodataset.org/val2017/000000039769.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)

# 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)

# 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)
```

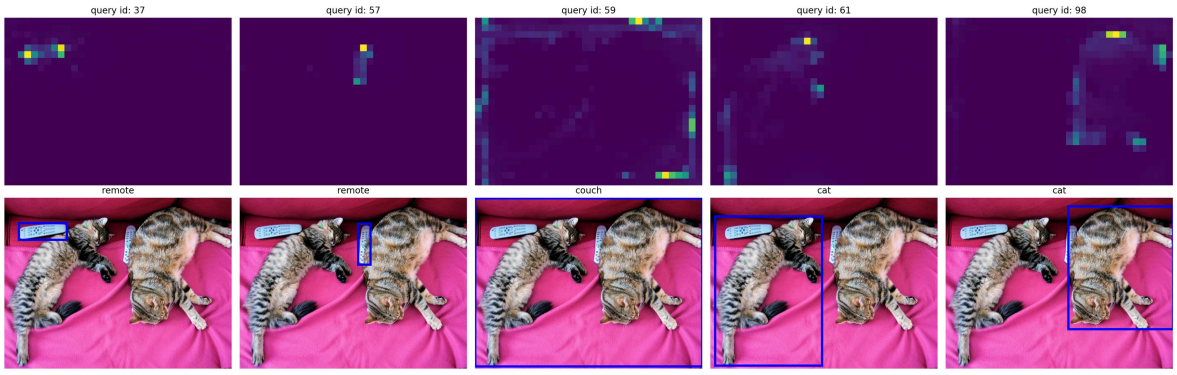
```
Using cache found in C:\Users\papib/.cache\torch\hub\facebookresearch_detr_main
C:\Users\papib\miniconda3\envs\d2l\lib\site-packages\torchvision\models\_utils.p
y:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may b
e removed in the future, please use 'weights' instead.
  warnings.warn(
C:\Users\papib\miniconda3\envs\d2l\lib\site-packages\torchvision\models\_utils.p
y:223: UserWarning: Arguments other than a weight enum or `None` for 'weights' ar
e deprecated since 0.13 and may be removed in the future. The current behavior is
equivalent to passing `weights=ResNet50_Weights.IMAGENET1K_V1`. You can also use
`weights=ResNet50_Weights.DEFAULT` to get the most up-to-date weights.
  warnings.warn(msg)
```



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.

In [4]:
```
# 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]
```

In [5]:
```
# 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_sca
    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()
```



In [6]:
```
# 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])
```

In [7]:
```
# 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])
```

In [8]:
```
# 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 = [(200, 200), (280, 400), (200, 600), (440, 800),]

# 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')
```

Discussion and Takeaways

1. Model Overview:

The model used here is **DETR (Detection Transformer)**, which utilizes a transformer architecture to perform object detection. One of the key advantages of DETR is that it does not require region proposal networks (RPNs) or non-maximum suppression (NMS) in its pipeline. Instead, the model directly predicts bounding boxes and class labels from the feature map. This approach simplifies the detection process by using attention mechanisms to determine where and what the objects are in the image.

2. Object Detection:

Using the **DETR model**, the object detection task is simplified by leveraging transformer-based attention mechanisms. The model outputs bounding boxes and class probabilities for each detected object in the image.

Post-processing:

After the model makes its predictions, the bounding boxes are rescaled to match the original image dimensions. Additionally, predictions with a confidence score greater than 0.9 are retained for display, ensuring that only the most confident detections are considered.

3. Self-Attention and Visualization:

The **self-attention** maps from the transformer encoder provide insight into how the model focuses on different parts of the image. By visualizing these attention maps, we can better understand which areas of the image the model attends to when processing specific parts of the image.

This is particularly useful for interpreting **transformer-based models**, as traditional CNN-based models often lack explicit attention mechanisms. This makes the transformer approach more transparent and interpretable.

4. Attention Visualization:

The **self-attention** visualization shows which spatial locations in the feature map are being attended to by the model when processing different objects. These maps can help explain the model's decision-making process for certain objects, allowing us to understand how the attention mechanism focuses on different areas during object detection.

Decoder Query Attention:

By examining how the **decoder's queries** attend to different areas of the image, we can see which parts of the feature map are most important for object detection predictions. This provides further insights into how the model is attending to the relevant regions when making its final predictions.

5. Interpretable Attention Mechanisms:

The ability to **visualize and interpret attention mechanisms** is crucial for understanding how transformer-based models generate object detection predictions. Unlike traditional CNN-based methods, which might rely on heuristics or complex post-processing steps, transformers like DETR use attention to directly focus on the regions that matter the most.

This transparency is important for improving **model interpretability**, especially in applications where understanding the decision-making process of AI models is essential (e.g., medical image analysis, autonomous driving).

6. Practical Insights:

The **DETR model's direct end-to-end approach** to object detection can be very effective but may require substantial computational resources, especially when performing fine-tuning or inference on high-resolution images. The transformer architecture is computationally demanding, which might make it slower than traditional detection models in some contexts.

Further Improvements:

To improve performance, further work could explore how different architectural tweaks (e.g., using a larger backbone like **ResNet-101**) or training strategies (e.g., **multi-scale detection**) can enhance the detection capabilities of the model. By experimenting with these methods, we could potentially make DETR more efficient and effective for a wider range of applications.

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.

Understanding Attention Mechanisms in DETR

In this section, we will explore the different types of attention used in the encoder and decoder of the DETR model. We'll describe how each attention mechanism works and discuss the key differences between them. Additionally, we will analyze the visualized attention maps from the previous section to gain further insights into their behaviors.

Encoder Attention: Self-Attention

The encoder in DETR uses **self-attention** mechanisms. Self-attention allows each element in the input sequence (in this case, the feature map of the image) to focus on all other elements in the sequence, regardless of their position. This means that each pixel in the feature map can attend to all other pixels, enabling the model to capture long-range dependencies and contextual information across the entire image.

Key Features of Encoder Attention:

- **Self-attention** operates within the encoder, where the key, query, and value are all derived from the same input sequence (the image feature map).
- The goal is to compute a set of attention scores for each position in the sequence, allowing it to attend to (or focus on) relevant parts of the image while processing the current feature.
- In DETR, this allows the model to build a global understanding of the image, with each pixel attending to other pixels to capture spatial relationships across the entire image.

Decoder Attention: Cross-Attention

In contrast, the **decoder** uses **cross-attention**. This type of attention mechanism involves two distinct sequences of information: one coming from the encoder (the image features) and the other coming from the decoder (the object queries, or object embeddings).

Key Features of Decoder Attention:

- In **cross-attention**, the **queries** come from the decoder, and the **keys and values** come from the encoder's output. This means the decoder uses the features learned by the encoder (global image context) and applies it to the specific object queries to focus on particular areas of the image.
- Cross-attention enables the decoder to selectively focus on relevant parts of the image (as determined by the queries) while maintaining a global understanding of the image from the encoder's self-attention mechanism.
- The main goal of cross-attention in the decoder is to associate the object queries with particular regions in the image, thereby enabling object detection.

Key Differences Between Encoder and Decoder Attention

- **Type of Attention**:
  - **Encoder**: Self-attention, where each position in the sequence (feature map) attends to all other positions within the same sequence (global dependencies in the image).
  - **Decoder**: Cross-attention, where each object query (decoder input) attends to the image feature map produced by the encoder (specific regions in the image, based on object queries).
- **Purpose**:
  - **Encoder Attention**: Aims to create a global representation of the image by allowing each pixel in the feature map to attend to every other pixel, capturing spatial relationships across the entire image.
  - **Decoder Attention**: Uses the context provided by the encoder's attention mechanism and applies it to specific object queries. Each query attends to different parts of the image based on the relevance to the object being detected.
- **Inputs**:
  - **Encoder**: The input is the image feature map, and self-attention is applied to this feature map to build a global context.
  - **Decoder**: The input is a set of object queries that will be matched with parts of the image via cross-attention. These queries are typically initialized randomly or based on learned object representations.

Analysis of Attention Mechanisms Based on Visualization

Encoder Self-Attention

From the visualizations in **Q2**, we can see the following characteristics of the encoder's self-attention:

- **Global Attention**: The self-attention maps are spread across the entire feature map, meaning that each pixel attends to all other pixels in the feature map. This helps capture relationships between different regions in the image, allowing the model to build a holistic representation of the image.
- **Complex Spatial Interactions**: The attention maps often highlight broader regions of the image, demonstrating how the model integrates information from different parts of the image to form a complete understanding. For example, in the case of an object like a "dog," the model might attend to both the body and the face of the dog to understand its full structure.
- **Less Localized**: Since the self-attention mechanism is not bound by a specific region in the image, it may not be as focused on fine-grained object details as the decoder's attention. Instead, it works at a more global level to encode spatial relationships and object features.

Decoder Cross-Attention

From the decoder's cross-attention visualization:

- **Targeted Attention**: Unlike the encoder's self-attention, the decoder's cross-attention is more **focused**. Each query in the decoder attends to a specific region of the image that is relevant to its associated object category. For example, the query for a "person" might focus primarily on regions of the image where a person is located, highlighting that the decoder is trying to match the query with specific parts of the image.
- **Refinement of Object Location**: The cross-attention mechanism is crucial for refining the bounding box prediction. The queries attending to the image ensure that the model detects objects by attending to their corresponding regions. It is more **localized** compared to the self-attention in the encoder.
- **Interaction with Queries**: The attention maps of the decoder often show how different object queries are associated with different regions of the image. This makes sense, as each query corresponds to a specific object that the model is trying to detect. For example, a query for a "cat" might attend mostly to the regions where the cat is present in the image.

Key Insights from Visualization:

- **Encoder Attention**: Global, broad, and contextually rich, self-attention helps the model understand the entire image and build a comprehensive feature map by connecting distant parts of the image.
- **Decoder Attention**: Targeted and specific, cross-attention allows the decoder to focus on particular regions of the image, matching the object queries with the relevant parts of the feature map to refine object localization and classification.

The distinction between these two attention mechanisms is fundamental to DETR's ability to process and detect objects. The encoder builds a global understanding of the image, while the decoder uses that understanding to make precise object detections, focusing attention on specific parts of the image.