

✓ 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
        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
        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)

        # project transformer outputs to class labels and bounding boxes
        pred_logits = self.linear_class(h)
        pred_boxes = self.linear_bbox(h).sigmoid()
```

```

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

```

✓ 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.

```

from google.colab import files
from PIL import Image
model = torch.hub.load('facebookresearch/detr', 'detr_resnet50', pretrained=True)
model.eval();

# url = 'http://images.cocodataset.org/val2017/00000039769.jpg'
# im = Image.open(requests.get(url, stream=True).raw) # put your own image

# Upload the image
uploaded = files.upload()

# Open the uploaded image
image_path = list(uploaded.keys())[0] # Get the filename of the uploaded image
im = Image.open(image_path)

# Display the image
im.show()

# 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)

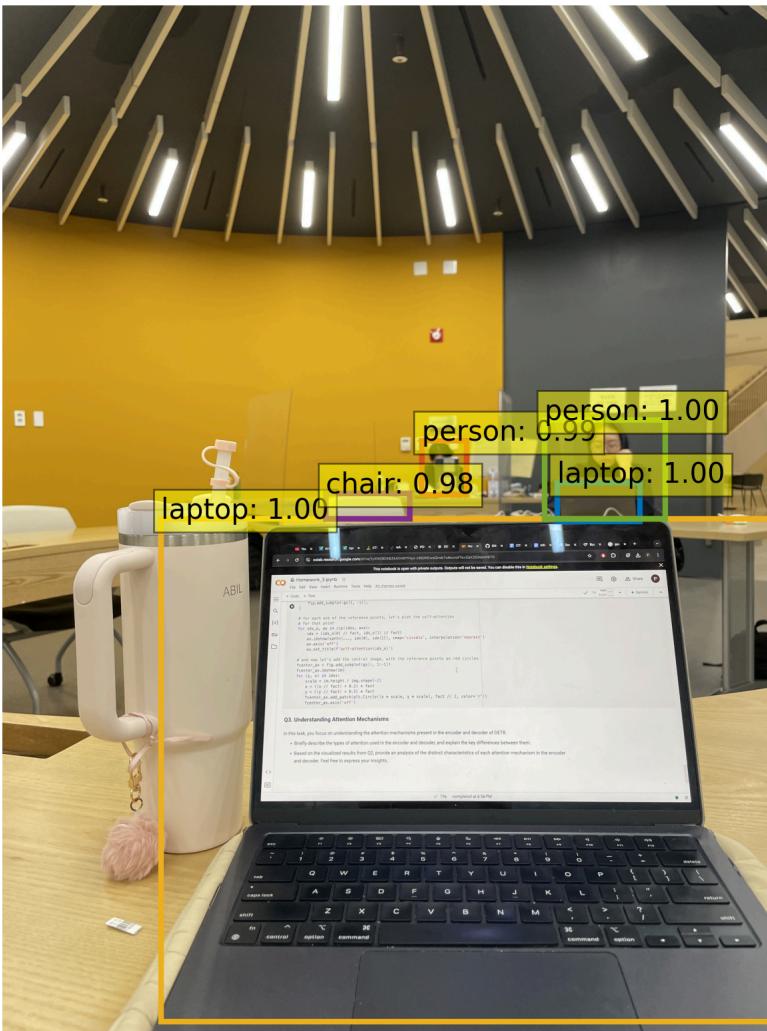
# keep only predictions with 0.7+ confidence
probas = outputs['pred_logits'].softmax(-1)[0, :, :-1]
keep = probas.max(-1).values > 0.9

# 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 /root/.cache/torch/hub/facebookresearch_detr_main
Choose Files | IMG_3416.jpg
• IMG_3416.jpg(image/jpeg) - 2704305 bytes, last modified: 11/14/2024 - 100% done
Saving IMG_3416.jpg to IMG_3416.jpg
```



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]

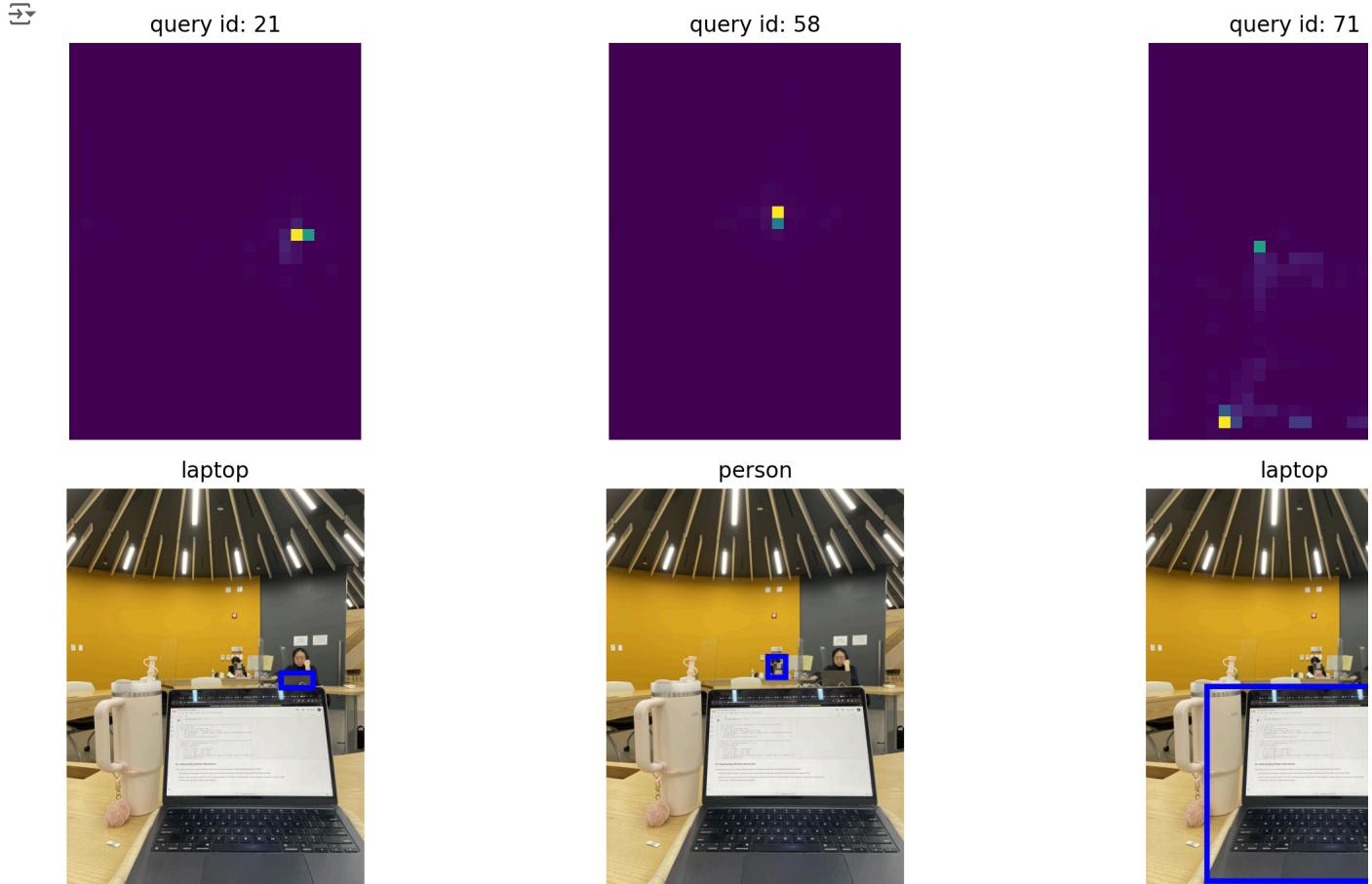
# get the feature map shape
h, w = conv_features['0'].tensors.shape[-2:]

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, 34, 25])

```

```

# 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([34, 25, 34, 25])

```

```

# 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 = [(800, 250), (700, 300), (600, 400), (500, 500),]

```

```

# 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]),

```

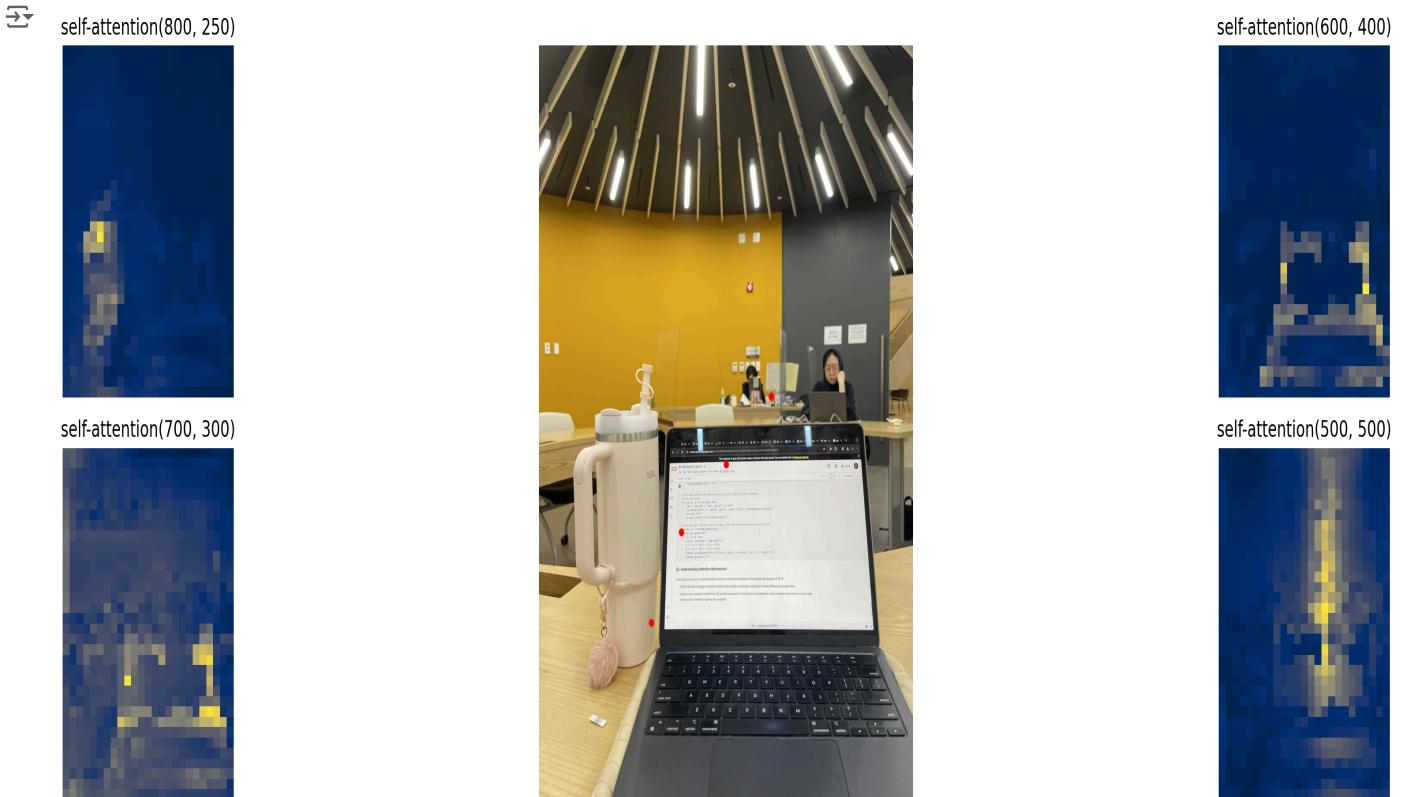
```

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')

```



* Note: I adjusted visualization reference point to get more it to a more bottom part since most of the object is there).

- Note: I adjusted visualization reference point to get more insights (move it to a more bottom part since most of the object is there).

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.
 - In the encoder, the self-attention mechanism is used. This mechanism takes image feature maps (processed by the CNN backbone), then each token (pixel/patch representation) attends to all other tokens to form a contextualized feature map. The goal of self-attention mechanism is to capture global spatial relationships and context within the image. On the other hand, the decoder uses cross-attention mechanism. This mechanism takes both the object queries and the encoder's output (self-attended image features). Then, decoder queries interact with encoder outputs, updating their representations based on the attention to relevant parts of the image. The goal of

cross-attention mechanism is to refine object representations by incorporating context from the image for detection, in another word identifying and localizing objects.

- The encoder self-attention mechanism captures how each pixel feature or patch interacts with others across the image. From the visualizations, we can see areas where attention is particularly focused, such as around object boundaries or regions with significant texture changes. These areas are likely where objects change drastically in appearance or position, and self-attention helps the encoder to capture the long-range relationships. Meanwhile, the decoder's cross-attention mechanism refines the object queries based on their relevance to different parts of the image. The attention weights show how the queries (which represent possible objects) "attend" to different spatial locations in the image. The cross-attention mechanism tends to focus on parts of the image that are relevant to the query (object). This is evident in the visualizations where the attention is localized to the bounding boxes of detected objects.