R12942103
潘品齊

Homework #1
Object Detection

Architecture of the Object Detector

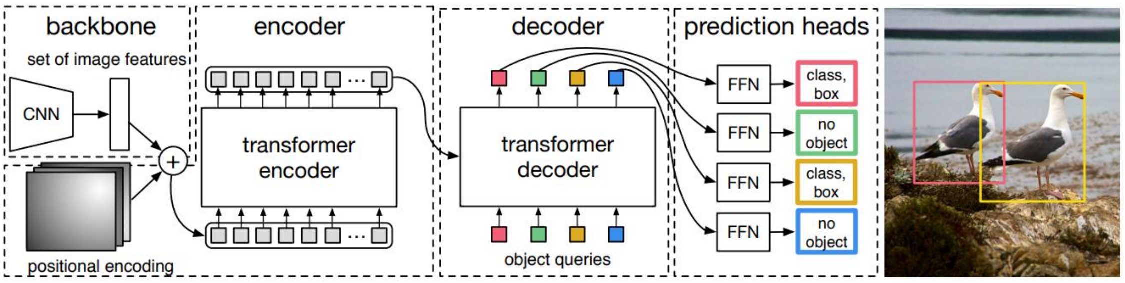

Fig.1: Overview of the DETR object detection architecture.

Our object detection model follows a transformer-based architecture, as shown in Fig.1. This architecture simplifies the detection pipeline by removing traditional hand-crafted components such as anchor generation and non-maximum suppression (NMS). It predicts object bounding boxes and their corresponding classes directly. Training code and pretrained models are available at https://github.com/PANpinchi/CVPDL_HW1_PANpinchi.

- **Backbone**
  We use a CNN backbone (e.g., ResNet-50) to extract image features, which are further processed by a transformer encoder. The CNN generates a low-resolution activation map from the input image, which is then augmented with positional encodings. These positional encodings allow the model to capture spatial information in a permutation-invariant manner.
- **Transformer Encoder**
  The transformer encoder models global relationships across the entire image using self-attention mechanisms. This stage is crucial as it helps the model learn interactions between all pixels of the image. The encoder consists of multiple layers of multi-head attention blocks followed by feed-forward neural networks (FFN).
- **Transformer Decoder**
  The transformer decoder processes a fixed number of learned object queries. These queries attend to the encoded image features and output embeddings, which represent potential object detections. Each object query corresponds to a single detection or "no object."
- **Prediction Heads**
  The output embeddings are passed through feed-forward networks (FFN) to predict the class labels and bounding boxes for the detected objects. The FFN outputs either the predicted class and bounding box or a special "no object" class. Since the transformer operates in a permutation-invariant manner, no post-processing like NMS is required.

This architecture allows the model to directly predict a fixed number of bounding boxes and object classes in a parallel and end-to-end manner, streamlining the detection process.

Implementation Details

- **Dataset Conversion and Preparation**

Before starting the training process, we converted our dataset to COCO format using the script convert2coco.py. This script processes the dataset by extracting bounding box annotations and formatting them into a JSON file that follows the COCO dataset structure. The script supports both training and validation splits and outputs JSON files for each. The key steps in the conversion process include:

  - Loading bounding box annotations from .txt files for each image.
  - Extracting bounding box coordinates and image dimensions.
  - Formatting these annotations into COCO-compatible JSON format.

- **Training**

Once the conversion was completed, the dataset was organized and prepared for training. The training phase was carried out using main.py, which implemented the full pipeline for model training. We employed the AdamW Optimizer with an initial learning rate of 0.001 and weight decay of 0.01. The loss function selected was the Cross Entropy Loss, which is well-suited for segmentation tasks. We set the batch size to 16 and trained the model for a total of 50 epochs. To ensure optimal learning, the learning rate was scheduled to decay by half every 10,000 steps. All training data was resized to 512x512 and normalized before being passed into the network. Data augmentation techniques such as random cropping and horizontal flipping were applied to further enhance the model's robustness. For hardware acceleration, we utilized an NVIDIA TITAN RTX GPU to train the entire model using the PyTorch framework.

- **Evaluation**

After training, we evaluated the model using the test.py script. This script loads the trained model and performs object detection on the validation set. It outputs a JSON file containing the predicted bounding boxes and class labels. The results are then compared against ground truth annotations to compute performance metrics such as:

  - mAP (50-95): Mean Average Precision across various IoU thresholds.
  - AP50 and AP75: Average Precision at specific IoU thresholds.

Table of your performance for validation set (mAP, AP_{50}, AP_{75}

The table below summarizes the performance metrics achieved on the validation set. The metrics include the mean Average Precision (mAP) across different Intersection over Union (IoU) thresholds (mAP@[IoU=0.50:0.95]) as well as AP scores at specific IoU thresholds (AP_{50}, AP_{75}).

Metrics (Simple baseline: 0.35 mAP, Strong baseline: 0.45 mAP)			
Calculation Method	mAP (50-95)	AP_{50}	AP_{75}
mmdetection	0.208	0.386	0.195
eval.py	0.5053	-	-
eval_ours.py	0.5053	0.7537	0.5329

- **Explanation of Results:**

eval.py: This script was provided by the teaching assistant, and it calculates only the mAP@[0.50:0.95] score, which evaluates the average precision across IoU thresholds between 0.50 and 0.95. Our model achieves a solid mAP of 0.5053 using this method, indicating strong general detection performance.

eval_ours.py: We extended the functionality of the eval.py script into eval_ours.py. In addition to computing mAP@[0.50:0.95], we also included calculations for AP at IoU thresholds of 0.50 (AP50) and 0.75 (AP75), which offer a more granular view of the model's precision. Notably, our model achieves a high AP50 of 0.7537 and a respectable AP75 of 0.5329, further showcasing its capability to detect objects accurately at various thresholds.

mmdetection: The mmdetection framework was used during the training phase to compute the validation metrics. However, the scores here differ significantly from those calculated using the eval_ours.py script, particularly the mAP@[0.50:0.95] score of 0.208, which is much lower than our eval_ours.py score of 0.5053. This discrepancy can be attributed to differences in implementation between mmdetection and eval_ours.py, such as varying thresholds or handling of edge cases during IoU calculation. Additionally, mmdetection might weigh smaller or medium-sized objects differently, which contributes to lower AP scores for these categories.

This comparison highlights the importance of understanding the calculation methods behind performance metrics, as variations in these methods can lead to significant differences in the reported results.

Visualization and discussion

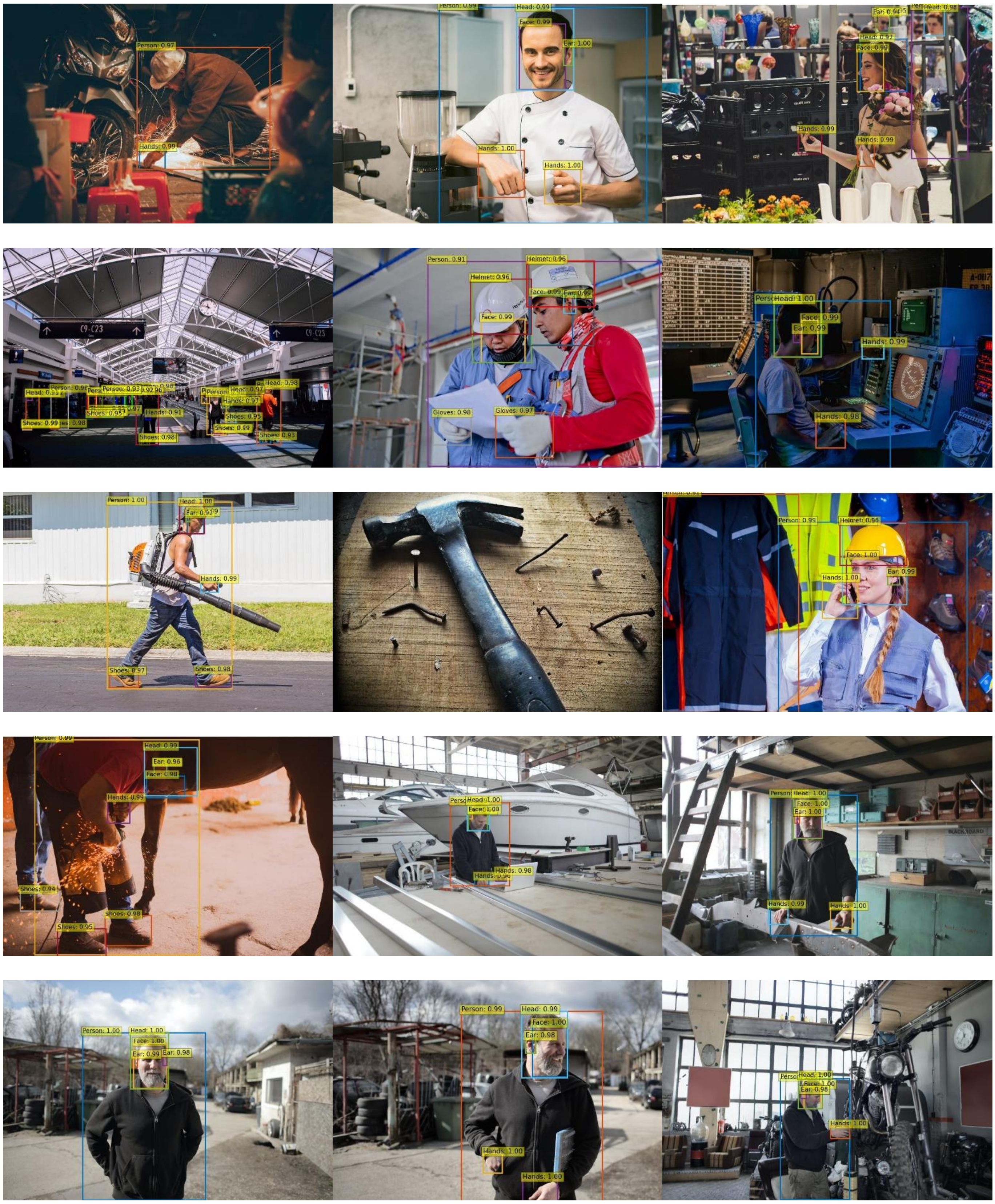

Fig. 2: Object Detection Results Across Various Environments

The images in Fig. 2 illustrate the results of the object detection model applied to various real-world scenarios. The model demonstrates strong performance in detecting objects such as hands, helmets, heads, and other safety equipment with high confidence scores.

- **Key Observations**
  1. **High Confidence Detections**: The bounding boxes in Fig. 2 are drawn with confidence scores displayed above 90% for most detected objects, indicating the model's accuracy in detecting critical objects like hands, helmets, and shoes. For example, in the construction site images, objects such as helmets, heads, and shoes are detected with scores above 0.90, which are essential for ensuring worker safety.

  2. **Multiple Objects in Complex Scenes**: The model successfully identifies multiple objects in complex scenes, such as workers in construction sites or machinery operators in the field, as shown in Fig. 2. This highlights the model's ability to handle challenging environments where multiple instances of the same class (e.g., shoes or helmets) occur.

  3. **Small Objects and Long-Tail Categories**: The model's performance on smaller objects or rare categories like tools, as seen in Fig. 2, shows variability. In some cases, these objects are detected correctly with high confidence, while in others, the confidence is lower or the detection is missed entirely. This variability could be attributed to the "long-tail effect," where categories with fewer examples in the training data, such as tools, are not detected as reliably as more frequent categories.

  4. **Missed and False Positive Detections**: Some instances of false positives and missed detections are evident, particularly for smaller or occluded objects. In Fig. 2, for instance, the model occasionally fails to detect smaller objects like parts of the machinery or misclassifies parts of the equipment as a person or tool.

- **Discussion on the Long-Tail Effect**

The long-tail effect refers to the issue where certain object categories are underrepresented in the dataset, leading to poorer detection performance for these categories. In Fig. 2, we observe that frequent categories like "Person" and "Shoes" are detected with high precision and confidence, whereas less frequent categories like "Tools" may not be detected as reliably. This discrepancy could be improved by employing techniques like class-balanced sampling or additional data augmentation focused on rare categories during training.

These results, as presented in Fig. 2, confirm that the model performs well in detecting common objects with high confidence, but there is room for improvement in handling rare categories and smaller objects.