Fine-tuning DINO Model for Pedestrian Detection

Sarvesh Shashikumar

September 24, 2024

Contents

1 Introduction

1.1 Task Description

The primary objective of this project is to fine-tune DINO (DETR with Improved deNoising anchOrboxes), an enhanced version of the original Detection Transformer (DETR) model, which is based on a transformer architecture specifically designed for object detection tasks. DINO introduces several improvements over DETR, including better performance in terms of accuracy and efficiency.

1.2 Dataset Overview

The dataset used for this project comprises 199 image-bounding box pairs. These images feature individuals walking around the IIT Delhi campus. Each image contains at least one bounding box, providing essential information for object detection tasks. The bounding boxes mark the position and size of objects (primarily people) within the images, which is crucial for training and validation.

2 Dataset Preparation

2.1 Splitting the Dataset

The dataset is provided in COCO format, a widely used standard for object detection and segmentation tasks. The first step in data preparation involved loading the annotations stored in JSON format using the `pycocotools` library.

Next, with the help of PyTorch's `DataLoader`, I shuffled the dataset randomly to ensure that it is not biased, and then split it into training and validation subsets. Specifically, 159 images were assigned to the training set, while 40 images were allocated to the validation set (80:20 Split). After splitting, I generated separate annotation files for each subset, and also created separate image directories corresponding to the training and validation data.

For ease of evaluation and fine-tuning, given the time constraints, the directory names and structures are same as the ones maintained by the original authors of DINO.

2.2 Data Annotation

Each image in the dataset is annotated with one or more bounding boxes that define the location of objects within the image. The annotation JSON file contains three key sections:

- **Images:** Contains metadata about each image, including height, width, ID, and file name.
- **Annotations:** Provides detailed information about each bounding box, including:
  - `iscrowd`: Indicates whether the object is a single object or a crowd.
  - `image_id`: The ID of the image corresponding to the bounding box.
  - `bbox`: Coordinates of the bounding box in the format [x, y, width, height].
  - `segmentation`: A polygonal segmentation mask (not used in this case).
  - `category_id`: The ID of the object category (e.g., pedestrian in this case).
  - `id`: Unique ID for the annotation.
  - `area`: The area of the bounding box.
  - `ignore`: Indicates whether the annotation should be ignored.
  - `vis_ratio`: The visibility ratio of the object.
  - `height`: The height of the bounding box.
- **Categories:** Lists the different categories of objects, including their IDs, names, and supercategories.

To verify the correctness of the annotations, I randomly selected a few samples and plotted the images along with their corresponding bounding boxes, ensuring the bounding boxes accurately surround the objects.

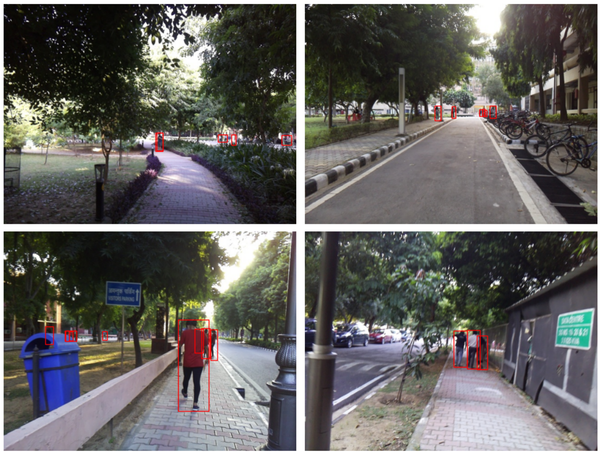


Figure 1: Pretrained model on a random batch

3 DINO Model Setup

3.1 Repository Setup

To fine-tune the DINO model, I cloned the official DINO GitHub repository. After cloning, I installed all the necessary dependencies by running the commands listed in the `requirements.txt` file. These dependencies include libraries for working with deep learning models, handling datasets, specifically those that follow the COCO format, and managing various utilities required for training the model.

3.2 Pre-trained Model

For fine-tuning, I utilized a pre-trained version of the DINO model. Specifically, I downloaded the DINO-4scale model, which is pre-trained with a ResNet-50 backbone, from the Google Drive link provided in the repository. Additionally, I chose a version of the model that had been trained for 33 epochs to ensure improved performance and better generalization during fine-tuning.

4 Evaluation of Pre-trained Model

4.1 Initial Evaluation

The DINO repository is well-structured, and it contains pre-written scripts for most of the necessary tasks, making the evaluation process highly convenient. To evaluate the performance of the pre-trained DINO model (trained for 33 epochs), the following bash command was executed, with the paths to the COCO dataset and the pre-trained model specified:

```
bash scripts/DINO_eval.sh /path/to/your/COCODIR /path/to/your/checkpoint
```

This command automatically loads the pre-trained weights, processes the dataset, and computes the evaluation metrics using the COCO evaluation protocol.

4.2 Results (Pre-trained Model)

The following are the Average Precision (AP) and Average Recall (AR) metrics for different Intersection over Union (IoU) thresholds, object sizes, and maximum detections per image, as obtained from the evaluation of the pre-trained model:

- **Average Precision (AP):**
  - IoU = 0.50:0.95 (all areas, max 100 detections): 0.508
  - IoU = 0.50 (all areas, max 100 detections): 0.860

  - IoU = 0.75 (all areas, max 100 detections): 0.570
  - IoU = 0.50:0.95 (small areas, max 100 detections): 0.413
  - IoU = 0.50:0.95 (medium areas, max 100 detections): 0.632
  - IoU = 0.50:0.95 (large areas, max 100 detections): 0.707

- **Average Recall (AR):**
  - IoU = 0.50:0.95 (all areas, max 1 detection): 0.122
  - IoU = 0.50:0.95 (all areas, max 10 detections): 0.548
  - IoU = 0.50:0.95 (all areas, max 100 detections): 0.618
  - IoU = 0.50:0.95 (small areas, max 100 detections): 0.548
  - IoU = 0.50:0.95 (medium areas, max 100 detections): 0.715
  - IoU = 0.50:0.95 (large areas, max 100 detections): 0.782

The pre-trained model demonstrated strong performance, especially in larger object detections.

5 Fine-tuning the Model

5.1 Fine-tuning Setup

Once the dataset was properly split into training and validation directories, the fine-tuning process was initiated using the following bash command:

```
bash scripts/DINO_train.sh --pretrain_model_path checkpoint0033_4scale.pth \
--finetune_ignore "label_enc.weight class_embed"
```

The `finetune_ignore` parameter is used to avoid training the label encoder and class embedding layers, as recommended by the DINO developers. This allows for a more focused fine-tuning process by freezing parts of the model that are already well-optimized.

Before initiating training, modifications were made to the `DINO_4scale.py` configuration file to suit the specific requirements of the task. Since this project focuses solely on detecting pedestrians, the `num_classes` and `dn_labelbook_size` parameters were adjusted to 2 (one for the pedestrian category and one for the background).

Additionally, batch size and number of epochs were configured in the same file to optimize the training process based on the available computational resources.

5.2 Challenges and Solutions

Fine-tuning the model initially did not produce satisfactory results. Although the training seemed to progress well, the final evaluation metrics were considerably lower, with performance metrics falling more than 50% below that of the pre-trained model.

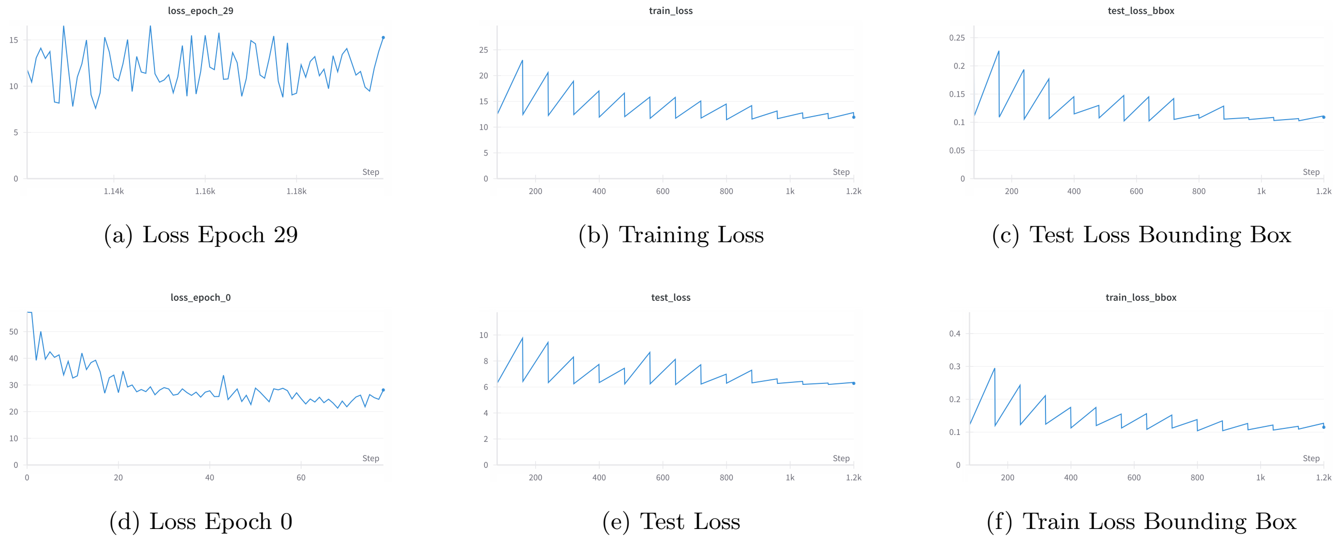
After investigating the issue, it became apparent that fine-tuning the entire model led to suboptimal results. To address this, I froze the backbone layers of the model and fine-tuned only the top layers. This significantly improved the performance, producing results comparable to the pre-trained model.

6 Fine-tuned Model Evaluation

6.1 Fine-tuned Evaluation

After fine-tuning the model using the improved strategy, the evaluation metrics for the fine-tuned model are as follows:

- **Average Precision (AP):**
  - IoU = 0.50:0.95 (all areas, max 100 detections): 0.326
  - IoU = 0.50 (all areas, max 100 detections): 0.648



(a) Loss Epoch 29

(b) Training Loss

(c) Test Loss Bounding Box

(d) Loss Epoch 0

(e) Test Loss

(f) Train Loss Bounding Box

Figure 2: Loss and Training Metrics Visualization

  - IoU = 0.75 (all areas, max 100 detections): 0.283
  - IoU = 0.50:0.95 (small areas, max 100 detections): 0.250
  - IoU = 0.50:0.95 (medium areas, max 100 detections): 0.464
  - IoU = 0.50:0.95 (large areas, max 100 detections): 0.332

- **Average Recall (AR):**
  - IoU = 0.50:0.95 (all areas, max 1 detection): 0.093
  - IoU = 0.50:0.95 (all areas, max 10 detections): 0.426
  - IoU = 0.50:0.95 (all areas, max 100 detections): 0.614
  - IoU = 0.50:0.95 (small areas, max 100 detections): 0.543
  - IoU = 0.50:0.95 (medium areas, max 100 detections): 0.724
  - IoU = 0.50:0.95 (large areas, max 100 detections): 0.618

While the fine-tuned model's performance is lower than the pre-trained variant, it still provides satisfactory results, especially after employing the base-layer freezing strategy. Further optimization could lead to even better performance.

7 Computational Challenges

One of the most significant challenges encountered during this project was the computational resource limitations. The complexity of the DINO model, combined with the large dataset, made it difficult to run training and evaluation tasks on a local machine.

7.1 Local Setup Limitations

Initially, the project was attempted on a local machine, but due to hardware limitations, only a single batch size evaluation was possible. The memory and processing power required to train the DINO model exceeded the capabilities of the local setup, making it impossible to carry out full-fledged training sessions.

7.2 Google Colab Usage

To overcome the limitations of the local machine, Google Colab was used as an alternative, leveraging the provided NVIDIA T4 GPU. However, while Colab provided a more powerful computing environment, it was not without its own challenges. Halfway through the fine-tuning process, Colab imposed rate limits on my usage, which resulted in an incomplete fine-tuning session and the need to restart the process. The rate limiting, which is a known limitation of using the free tier of Colab, was a significant setback.

7.3 Switching to Kaggle

In response to the issues encountered on Colab, I switched to using Kaggle for the training and fine-tuning processes. Kaggle provided an uninterrupted environment with sufficient resources to continue the task, though it also has limitations in terms of available GPU time and memory.

7.4 Precautionary Measures

As a precaution, I began monitoring and logging key training results using the *Weights and Biases* library. This allowed me to track the progress of the model training, save intermediate results, and avoid data loss due to unexpected interruptions. The use of this tool provided additional security, ensuring that valuable progress was not lost in case of hardware or session limitations.

Despite these challenges, the combined use of multiple cloud platforms and monitoring tools helped in overcoming most computational barriers and achieving the project's goals.

8 Conclusion and Future Work

In this project, the DINO model, an advanced transformer-based object detection algorithm, was fine-tuned for pedestrian detection. Despite facing computational challenges, such as hardware limitations and platform constraints, the model was successfully fine-tuned and evaluated, with satisfactory results. The exploration of techniques like freezing parts of the model and using external resources such as Kaggle helped mitigate these issues and contributed to the overall success of the project.

8.1 Future Work

Looking ahead, there are several potential areas of improvement and exploration for future work:

- **Low-Rank Adaptation (LoRA):** One promising avenue is the integration of Low-Rank Adaptation (LoRA) techniques into the transformer-based architectures. LoRA reduces the number of trainable parameters in the model by decomposing the weight matrices, allowing for more efficient fine-tuning with significantly less computational overhead. This could lead to faster training times and lower resource consumption, making fine-tuning more accessible for individuals with limited hardware resources.

- **Quantization:** Another promising area for future work is the application of quantization techniques. Quantization reduces the precision of the model's weights, which can significantly decrease both memory usage and inference time, with minimal impact on model accuracy. This technique would make the DINO model more suitable for deployment on edge devices or in real-time applications where computational resources are constrained.

- **Model Pruning:** Further investigation into model pruning techniques could also prove beneficial. Pruning removes redundant or less important parameters in the network, leading to a more lightweight model. Combined with quantization and LoRA, this could result in substantial improvements in both model efficiency and speed.

- **Improved Model:** DINO has come a long way since the launch of the model that we used in this task. There are a multiple newer models, including DINOv2, which may significantly outperform DINO. The inclusion of such models will tremendously benefit in the implementation of this task.

By focusing on these areas, future work can aim to make DINO and other transformer-based models more efficient and scalable, paving the way for broader usage in both academic research and practical applications.