Towards Seamless Hex-Bug Tracking in Complex Environments: A Detection Transformer (DETR) based Framework

Salil Bhatnagar SALIL.BHATNAGAR@FAU.DE
MSc ACES e mobility
Matriculation Number 23220858
Friedrich-Alexander-Universität Erlangen-Nürnberg

Editor: Salil Bhatnagar

Abstract

The demand for efficient frameworks to track objects and their key-points has risen in parallel with the intricacy of behavioral data and analysis. In this study, the central objective is to develop and implement a simple yet effective framework for tracking the head positions of artificial hex-bugs: small robotic entities that navigate intricate surroundings. The proposed method introduces an efficient approach for real-time tracking of these artificial hex-bugs. This is achieved through the adaptation of a framework based on Detection Transformer (DETR) [1], enabling direct retrieval of regressed head coordinates. Subsequently, a straightforward yet efficient associative algorithm is applied to rectify any inaccuracies in tracking. This study further delves into a comprehensive exploration of various parameters and their impact on the training process and overall performance while utilizing the DETR framework. Furthermore, an in-depth performance comparison is presented, incorporating diverse backbones within the DETR framework [3].

Keywords: Object tracking, Tracking as points, Object detection, Keypoint detection, Detection Transformer, DETR, Computer Vision

1. Introduction

In the domain of computer vision and object tracking, the demand for accurate, efficient, and adaptive frameworks has risen in parallel with the behavioral data and analysis. This is further fueled by the ever-evolving challenges posed by complex environments where objects of interest interact dynamically. Apart from the technical challenges posed by object tracking, its applications are remarkably diverse and have found relevance in numerous fields.

Object tracking is an indispensable tool in surveillance and security systems, enabling the monitoring of individuals and objects in real-time, thus enhancing public safety. In robotics, it plays a pivotal role in enabling machines to navigate and interact with their environments autonomously. It is also instrumental in ecological studies, allowing scientists to track the movements of wildlife and study their behaviors in their natural habitats. Furthermore, it is used to track microorganisms for scientific and medical research. These diverse applications signifies the importance of developing robust and efficient tracking frameworks.

This study presents an exploration into the realm of object tracking, with a specific focus on artificial hex-bugs – small robotic entities that navigate intricate surroundings. The central aim is to introduce a simple and efficient framework that tackles the complexities of tracking for such entities, especially their distinctive head positions.

The proposed framework in this study capitalizes on the advancements in deep learning, particularly leveraging a version of the Detection Transformer (DETR) architecture [1]. Unlike traditional object tracking techniques, the approach presented here combines detection, regressing head location and associating tracking with previous frames, thereby enhancing the overall efficiency and accuracy of tracking hex-bugs. The ultimate goal is to provide a seamless and reliable means to track the intricate movements of artificial hex-bugs, even in challenging and dynamic environments. This report is structured as follows:

- Section II elaborates on the used framework, detailing the modifications made to the DETR architecture and the subsequent incorporation of the associative algorithm. These changes have been adopted for the specific problem of hex-bug tracking in complex environments.
- Section III presents an extensive experimental evaluation, analyzing the impact of various parameters on the training process and overall tracking performance. Furthermore, it offers a comprehensive performance comparison by evaluating different backbone architectures within the DETR framework [1].
- Section IV concludes the paper by summarizing the contributions.
- Lastly, Appendix section includes the coding snippets for the training pipeline as well as full code for the associative algorithm. It also lists the optimum hyperparameters which yielded the most best results. Further, detailed code can be found at my GitHub repository : https://github.com/SalilBhatnagarDE/Hexbug_tracking.

2. Methodology

2.1 DETR Architecture

The DETR architecture [1], as shown in Figure 1, is remarkably straightforward. It comprises three primary elements, which I will elaborate further: a CNN backbone for obtaining a concise feature representation, an encoder-decoder transformer [10], and a basic feed-forward network (FFN) for generating the ultimate detection prediction.

For the implementation of the DETR framework, I have utilized the Hugging Face implementation (https://huggingface.co/docs/transformers/main/model_doc/detr), which is very well-documented and easy to implement. Custom changes can be made very easily using the configuration file of the mentioned repository.

2.1.1 CNN Backbone

I have conducted experiments with three distinct types of backbones to evaluate the performance of each. All of the backbones belong to the ResNet [4] family of CNN backbones. ResNet (Residual Network) is a family of deep convolutional neural networks known for

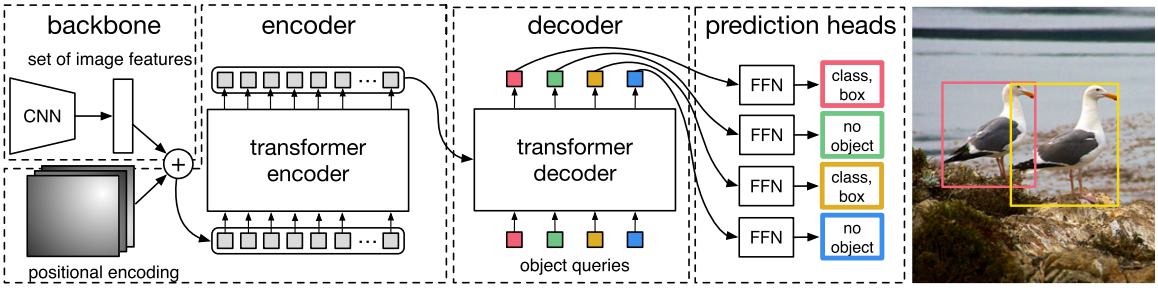

Figure 1: DETR uses a conventional CNN backbone to learn a 2D representation of an input image. The model flattens it and supplements it with a positional encoding before passing it into a transformer encoder. A transformer decoder then takes as input a small fixed number of learned positional embeddings, which are called as object queries, and additionally attends to the encoder output. Each output embedding of the decoder is then passed to a shared feed forward network (FFN) that predicts either a detection (class and bounding box) or a "no object" class.

its ability to train very deep networks effectively by using residual connections. I have experimented with ResNet50 [4], ResNet50 with Dilated Convolutions, and ResNet101.

2.1.2 Transformer encoder

The transformer encoder is same as in the DETR detection paper [1]. Each encoder layer adheres to a standard architecture, consisting of a multi-head self-attention module and a feed-forward network (FFN) [10]. Since the Transformer architecture is invariant to permutations, DETR framework enhance it by incorporating fixed positional encodings [2] that are added to the input of each attention layer.

2.1.3 Transformer Decoder

The transformer decoder is similar to the DETR detection paper [1]. In this DETR model, in each decoder layer, the model concurrently decodes the N objects. The N object queries undergo transformation into an output embedding by the decoder. Subsequently, they are independently decoded into box coordinates and class labels by a feed-forward network (as described in the next subsection), resulting in N final predictions. Through self- and encoder-decoder attention [10], the model conducts global reasoning across all objects, considering pairwise relationships between them. This approach enables the model to utilize the entire image as context while making predictions.

2.1.4 Prediction feed forward network (FFN) as a direct regressor for hex-bug head coordinates

The final prediction is computed by a 3-layer perceptron with ReLU activation function and hidden dimension d, and a linear projection layer. The FFN predicts the normalized

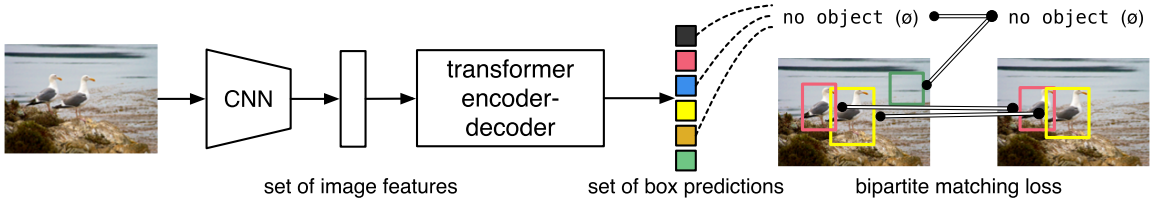

Figure 2: During training, bipartite matching uniquely assigns predictions with ground truth boxes. Prediction with no match should yield a "no object" class.

center coordinates, height, and width of the box w.r.t. the input image, and the linear layer predicts the class label using a softmax function.

In the standard implementation, we get four values out of bounding box coordinates predictions. These are: normalized center coordinates of the bounding boxes and the width and height of the bounding boxes.

In my implementation, I have used box coordinates (x, y) as the location of the head of hex-bugs and then (w, h) as the fixed height and width for the bounding box, which is assumed to be the same for all of the hex-bug detections. The weight and height used are 70x70, which almost covers the head of the hex-bug and gives the network the context to learn. In this way, I am able to directly regress the hex-bug locations as an output from the FFN network. This approach is straightforward and works well for me, as one can see in the results section.

2.2 Hungarian Matching and Loss Function

DETR infers a fixed-size set of N predictions in a single pass through the decoder, where N is set to be significantly larger than the typical number of objects in an image.

I have limited this to a value of 5 since in all of the training and test videos the maximum number of hex-bugs is 4.

2.2.1 Hungarian Matching Algorithm (Bipartite Matching)

One of the main difficulties of training is to match predicted objects (class, position, size) with respect to the ground truth. The DETR loss produces an optimal bipartite matching [5] between predicted and ground truth objects, and then optimize object specific losses, as shown in Figure 2. The Hungarian algorithm [5] is used for this bipartite matching. The details of the algorithm can be found in the DETR detection paper [1].

I have modified the weight factors for the calculation of Hungarian cost function so to achieve optimum results for my problem of hex-bug tracking.

There are many parameters defined for Hungarian matching algorithm. The parameters are namely :

1. Class cost - Relative weight of the classification error in the Hungarian matching cost.
2. Bounding Box cost - Relative weight of the L1 error of the bounding box coordinates in the Hungarian matching cost.

3. GIoU cost - Relative weight of the generalized IoU loss of the bounding box in the Hungarian matching cost.

I have conducted experiments with these parameters, and in the results section, I present the results with different parameters and how changing these will affect the convergence of the training.

2.2.2 Loss

The second step is to compute the loss function for all pairs matched in the previous step. We define the loss similarly to the losses of common object detectors, i.e. a linear combination of a negative log-likelihood for class prediction, a L1 box loss and the generalized IoU loss [7].

Further details to the loss function can be referred in DETR detection paper [1].

There are many parameters defined for loss. The parameters are namely:

1. Bounding box loss coefficient - Relative weight of the L1 bounding box loss in the object detection loss.

2. GIoU loss coefficient - Relative weight of the generalized IoU loss in the loss.

3. EoS coefficient - Relative classification weight of the 'no-object' class in the loss.

I have conducted experiments with these parameters, and in the results section, I present the results with different parameters and how changing these will affect the convergence of the training.

2.2.3 Auxiliary loss

To stabilize the training and enable the network to learn the number of objects in a given image, I incorporate auxiliary losses [8] within the decoder branch. Following each decoder layer, an auxiliary loss is computed and employed during training, ensuring that the network converges steadily towards the minimum. The implementation of auxiliary loss is facilitated using the Hugging Face [9], and their significance and impact on the study's results will be demonstrated in subsequent sections.

2.3 Associative Algorithm

To ensure accurate hex-bug predictions within a single video, I developed a custom associative algorithm tailored specifically to the task of locating hex-bug heads. Although I explored other available algorithms such as Sorting and DeepSort [11], they prove susceptible to errors related to object motion. Consequently, creating a custom associative algorithm emerges as a viable solution.

Here's how my algorithm functions:

1. Determining Hex-bug Count: The algorithm begins by leveraging the trained DETR detection model to calculate the number of hex-bugs in a video, typically consisting of multiple frames (e.g., 100 frames).

2. Mode-Based Allocation: Based on the mode (most frequently occurring count) of hex-bugs detected in each frame, the algorithm assigns a fixed number of hex-bugs to every frame in the video. For instance, if the DETR model identifies 2 hex-bugs in 80 frames and only 1 in the remaining 20 frames, the associative algorithm assumes 2 hex-bugs in each frame as ground truth.

3. Correction of Missed and Extra Detections: The algorithm then proceeds to rectify any errors in detection. If a hex-bug is missed in one frame but present in neighboring frames, the algorithm generates new instances by estimating the missed hex-bug's position. In cases where a hex-bug remains undetected for a frame, the algorithm generates a new instance of the missing hex-bug by averaging its location from both the previous and future frames where it is detected. Similarly, if there are extra detections, it deletes these instances.

4. Association and Tracking: In each frame, the algorithm compares the current hex-bug predictions to those in previous frames. It establishes associations between hex-bugs, effectively tracking them across frames by assigning unique IDs. This tracking ensures that each hex-bug is consistently identified throughout the video. The association is done based on the minimum distance metric and implemented as a greedy solution to find the matching.

This custom associative algorithm offers a straightforward yet effective means of improving model accuracy. There is a huge boost in performance after the implementation of this associative algorithm since all the missed ones and extra detections are being corrected by this algorithm. Thus, this algorithm is a part of all the Experiments as explained in next sections. In the Appendix A.4, I present the code for the algorithm.

2.4 Implementation

2.4.1 Pretrained models

Fully pretrained DETR models are readily available on the Facebook HuggingFace page [3], and are publicly accessible. These models have undergone training on ImageNet with 1000 classes and can be directly fine-tuned on a custom dataset. Alternatively, there is an option to employ only a pretrained backbone while training the transformer head from scratch on the custom dataset. For my research, I found that using a fully pretrained DETR model led to faster convergence.

I also conducted experiments by comparing different scenarios, including freezing various layers and making all layers trainable. I will provide detailed insights into these performances in the Experiments section of my study.

2.4.2 Learning Rates and Optimizer

In the training of DETR, I employ the AdamW [6] optimizer. Specifically, I set the initial learning rate for the transformer to 10^{-4}, while the learning rate for the backbone was set to 10^{-5}, accompanied by a weight decay of 10^{-4}. This configuration closely adheres to the recommendations outlined in the DETR detection paper [1], which is widely regarded as an optimal approach to training. The rationale behind using a lower learning rate for the

backbone as opposed to the transformer head is rooted in the fact that the backbone has already been pre-trained to extract meaningful representations from images. Retraining it with the same learning rate leads to training instability and fluctuation.

2.4.3 Data Augmentation

I incorporate various data augmentation techniques, including Color Jitter, Gaussian Blur, Re-scaling, Random Crop and Color Inversion, to facilitate the model in learning the most generalized representations.

- In Color Jitter, adjustments in brightness, contrast, and saturation are allowed to vary within a range of -20 percent to +50 percent, while the hue can fluctuate within a range of -0.1 to 0.1.
- Gaussian blur is applied using a kernel size of 5, with the sigma ranging from 0.1 to 3.0. After conducting numerous experiments and analyses, these values were determined to be optimal for the hex-bug dataset at hand.
- Color inversion is also valuable in training, as it helps diversify the dataset and exposes the model to different variations of the same image. This can enhance the model's ability to recognize objects under various lighting conditions and color schemes.
- Additionally, I implement scale augmentation, resizing input images such that the shortest side falls within a range of at least 480 to at most 800 pixels, while the longest side is capped at a maximum of 1333 pixels [12].
- To enhance the learning of global relationships through the self-attention mechanism of the encoder, random crop augmentations are also applied during training, contributing to improved performance.

2.4.4 Training

For all experiments and training procedures, the FAU TinyGPU Cluster is utilized, harnessing the power of 4 Nvidia A100 GPUs, each equipped with 40 GB of RAM. Given the transformer's flexibility in accepting various input sizes, I directly fed the images into the DETR model without any resizing. To optimize GPU usage and memory, the number of images per GPU was set to 14, resulting in a total batch size of 56.

During training, an Early Stopping criterion is applied, primarily based on the validation loss. A patience parameter is set at 15, which means that if the model's performance did not improve for 15 consecutive iterations, the training process would halt. To facilitate monitoring and recording the training progress, a Tensor board logger is employed. Additionally, the best weights of the model are saved for later use, ensuring that the model with the highest validation performance could be retrieved for further evaluation or deployment.

3. Results and Experiments

3.1 Dataset

The dataset is accessible through the Traco team's Git repository at :

- https://github.com/ankilab/traco_external

It comprises 100 videos, each containing approximately 100 frames. These videos exhibit significant variability, encompassing diverse backgrounds, varying lighting conditions, different quantities of hex-bugs, and a range of hex-bug colors.

To introduce even greater diversity, I included a subset of videos recorded under different background conditions within the dataset. In Figure 3, my custom made videos and annotations can be seen.

Figure 3: Examples of custom videos and these were annotated using Traco annotation tool and used as additional training videos.

The dataset is then divided into training and validation sets, allocated at 75 percent and 25 percent, respectively, ensuring a similar distribution of videos in both partitions. The performance metric on the validation set will be detailed in subsequent sections and nonetheless mentioned the score metric has been shown for the validation set. Additionally, a small test set is available for final evaluation, consisting of 5 videos, and it contributes to the Traco leader-board at https://traco.anki.xyz/leaderboard.php.

3.2 Evaluation Metric

An evaluation metric is devised that imposes penalties for deviations between predictions and ground truth. These penalties are computed based on the disparities between the predicted coordinates (x', y') generated by the trained DETR detection model and the actual ground truth values (x, y). The greater the disparity between the prediction and ground truth, the more substantial the penalty incurred. The penalty is equal to the distance between ground truth and prediction coordinates. These penalties are calculated for each frame and aggregated for both the validation and test sets individually. In addition to the penalties mentioned above, there are some more penalties, as mentioned below.

- Excessive Hex-bug Detections: If more than 4 hex-bugs are detected in a single frame, a substantial penalty of 1,000,000 is added to the score.
- Incorrect Number of Frames : In cases of an incorrect number of frames, a penalty of 10,000 is added for each misjudged frame.

- Incorrect Number of Detections: In cases of an incorrect number of detections, a penalty of 1,000 is added for each misjudged detection.

3.3 Results with different backbones

Table 1 provides a comprehensive overview of the results achieved by employing different backbones as feature extractors. It is crucial to emphasize that all the results presented in the table are based on the utilization of meticulously fine-tuned hyperparameters as mentioned in Appendix A. These optimal hyperparameters remain consistent throughout the results presented in Table 1. Detailed discussions on the impact of these hyperparameters on the training process are elaborated in subsequent sections. Figure 4 presents some of the qualitative results after training of DETR-ResNet50 framework.

For a complete reference to the final and optimum hyperparameters utilized, please refer to Appendix A.

Model	Frames Per Second	#params	Score/Cumulative penalty
DETR-R50	28	41M	95258
DETR-R50-DC5	12	41M	86582
DETR-R101	20	60M	93552

Table 1: Comparison with ResNet-50, ResNet-50 Dilated Convolutions (DC) and ResNet-101 backbones on the TRACO validation set. All the models are trained with same hyperparameters mentioned in Appendix A and using the same early stopping criteria on validation loss. The lower the score metric, the better the model performance. The associative algorithm is applied in all of these experiments.

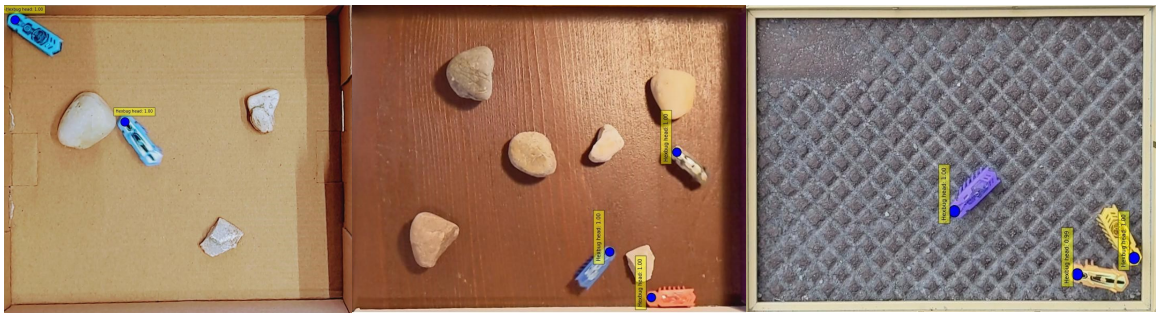

Figure 4: Visualization of predictions by fully trained DETR-ResNet50 on Traco validation dataset. The blue dot represents the head of the hex-bugs and additional yellow box gives the class probability of being a hex-bug.

3.4 Experiments

3.4.1 Effect of Auxiliary losses in decoder

Auxiliary losses [8] can be incorporated at each step within the decoder. The primary purpose of introducing auxiliary losses, specifically in the form of cross-entropy loss for determining the correct number of objects in a frame, is to facilitate network convergence. I conducted experiments both with and without auxiliary losses, the outcomes of which are presented in Figure 5.

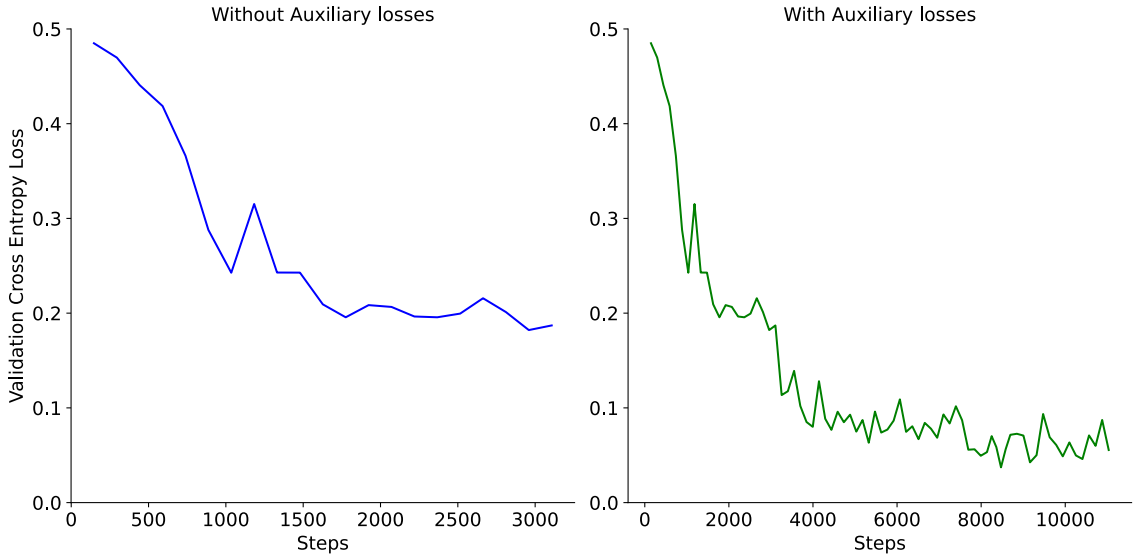

Figure 5: Comparison with and without auxiliary losses. As clearly seen, the validation cross entropy loss reaches minimum when we have auxiliary losses, while it struggles to converge in case of without auxiliary losses in the decoder. The model framework used is DETR-ResNet50.

When auxiliary loss is omitted, the model encounters challenges in converging to the minimum classification cross-entropy loss, resulting in erroneous frame predictions. Consequently, it is strongly recommended to consistently utilize auxiliary losses, and further, to apply them after each step in the decoder. This is clearly evident from the Figure 5.

3.4.2 Effect of different components in Hungarian matching algorithm for hex-bugs detection

The Hungarian matching algorithm [5] serves the purpose of aligning predictions with ground truth based on a cost function. Following the successful matching of predictions with the ground truth labels, the final loss is computed, as explained in the previous sections. This Hungarian cost function incorporates various components, including Class cost, GIoU cost, and Bounding box cost. The recommended weighting for each of these components is delineated in the DETR detection paper. According to the original implementation

[1] [3], the optimal weights for each component in the Hungarian matching cost function are as follows:

- Default Hungarian Cost = Classification Cost + 5 * L1 Bounding Box Cost + 2 * GIoU Cost

However, as mentioned earlier, the above cost function was found to be sub-optimal for hex-bug tracking and detection. Through experimentation, the optimal cost function for the hex-bug detection problem was determined to be as follows:

- Optimum Hungarian Cost for Hex-bug problem = 2 * Classification Cost + 5 * L1 Bounding Box Cost + 2 * GIoU Cost

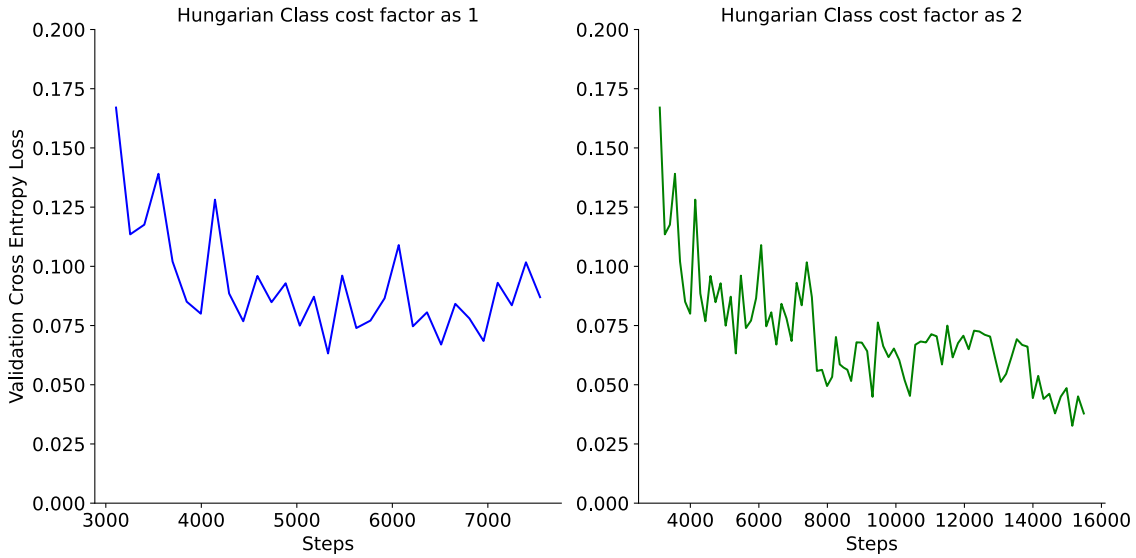

Figure 6: Comparison with class cost factor as 1 and class cost factor as 2. As clearly seen, the validation cross entropy loss reaches minimum when we have class cost factor as 2 in the Hungarian matching algorithm. In above shown example, the class cost factor was increased to 2 after 7300 steps and a sharp decline in the cross entropy validation loss is visible afterwards. The model framework used is DETR-ResNet50.

To attain the optimal loss at the conclusion of training, the classification cost is multiplied by a factor of 2. However, increasing it further to a factor of 3 or 4 resulted in training convergence issues. The outcomes of these experiments are presented in Figure 6.

3.4.3 Effect of Loss ablations

To assess the significance of various components in the matching cost and loss, I conducted experiments involving the activation and deactivation of these components in several models.

The loss comprises four key components: classification loss, L1 bounding box distance loss, Generalized Intersection over Union (GIoU) loss, and auxiliary loss.

Table 2: Effect of different loss components on score metric/ cumulative penalty. I trained several models by turning off L1 loss, or GIoU loss or Auxiliary loss, and observed that L1 gives poor results on its own, but when combined with GIoU improves performance. The associative algorithm is applied in all of these experiments.

class	**L1**	**GIoU**	**Auxiliary**	**Score metric/ Penalty**
X	X	X		152050
X	X			197502
X		X	X	130325
X	X	X	X	95258

Given that classification loss is fundamental for training and cannot be disabled, I conducted experiments by training models without the bounding box distance loss, models without the auxiliary loss, models without the GIoU loss, and compared their performance with the baseline model trained with all the losses. The results are summarized in Table 2.

3.4.4 Effect of classification weight of the 'no-object' class in the loss

I employ a fixed number of 5 queries in the DETR decoder segment to identify hex-bugs. This implies that my implementation is designed to identify a maximum of 5 hex-bugs in a single frame, which is appropriate given that the maximum number of hex-bugs in any training and test video is 4. However, since the number of hex-bugs varies across different videos, there exists a class imbalance factor, particularly regarding the classification weight for the 'no-object' class in the loss function.

Table 3: Below shows the distribution of number of hex-bugs in each of the training videos. This is used to calculate the Eos coefficient : the relative classification weight of the 'no-object' class in the object detection loss.

Number of Hex-bugs	**Number of Training videos**
1	43
2	18
3	22
4	3
Average hex-bugs per videos	**1.83**
Eos coefficient	**1.83/(5-1.83) = 0.578**

To elaborate, let's consider a scenario where all videos contain 2 hex-bugs, but the decoder employs 5 object queries. In such cases, three of the object queries should consistently yield 'no object' predictions, while the remaining two should accurately classify the presence of hex-bugs. To facilitate balanced training across these classes, it is logical to introduce an Eos-coefficient of 0.6. This coefficient is multiplied with the classification loss while

predicting the 'no-object' class. Consequently, this adjustment balances the predictions, ensuring that two object class predictions and three non-object class predictions are effectively managed during training. The calculation of the Eos-coefficient in my implementation is detailed in Table 3.

3.4.5 Effect of freezing layers in the backbone

Experiments were undertaken to investigate the impact of frozen layers within the backbone versus having all layers in the entire DETR transformer model as trainable. Additionally, various learning rates for the backbone were explored. Figure 7 illustrates the contrast between frozen layers in the ResNet-50 [4] backbone and trainable layers in the ResNet-50 backbone. Similar observations were made with the ResNet-50 Dilated Convolutional network.

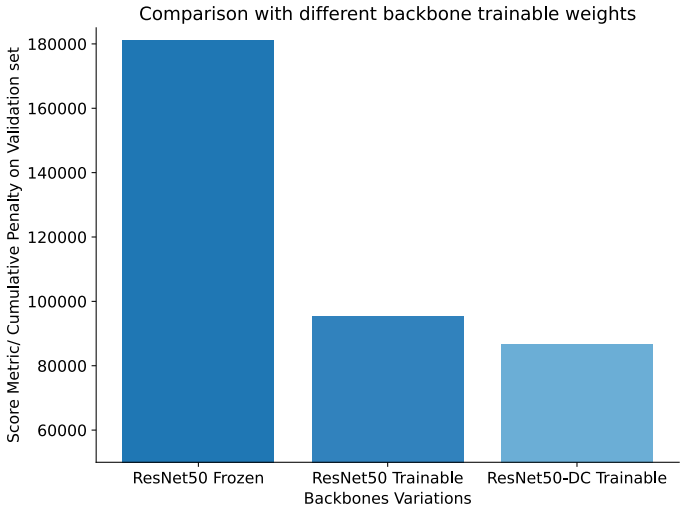

Figure 7: Effect of different trainable layers in Backbones in DETR framework. All the models are trained with same hyperparameters and with similar early stopping criteria but with different trainable layers.

4. Conclusion

In this study, I introduced DETR detection combined with a simple associative algorithm for the purpose of hex-bug tracking, which represents an efficient approach in the realm of object tracking systems. Grounded in transformer architecture [10] and leveraged by bipartite matching loss for direct set prediction, my approach yielded good results, as evidenced by my experiments on the provided dataset.

The combination of DETR [1] and my simple associative algorithm not only showcases effectiveness in its tracking capabilities but also signifies its ease of implementation and adaptable architecture. It is worth noting that my implementation exhibited promising

performance on the Traco leader board, securing the second position, ultimately securing third place in the final testing phase.

Nonetheless, it's important to acknowledge that the deployment of DETR for detectors introduces its own unique set of challenges, particularly with regards to training, optimization, and performance considerations, particularly when dealing with hex-bug like objects in complex environments.

Acknowledgments

I would like to acknowledge Professor Andreas Kist and the Tracking Olympiad organization team (René Groh, Luisa Neubig and Marion Dörrich) for their support for this project.

References

[1] Carion, Nicolas et al. "End-to-End Object Detection with Transformers". In: *European Conference on Computer Vision (ECCV)*. 2020.

[2] Cordonnier, Jean-Baptiste, Loukas, Andreas, and Jaggi, Martin. "On the Relationship Between Self-Attention and Convolutional Layers". In: *International Conference on Learning Representations (ICLR)*. 2020.

[3] Facebook, Research. *DETR: End-to-End Object Detection with Transformers*. https://github.com/facebookresearch/detr. 2020.

[4] He, Kaiming et al. "Deep Residual Learning for Image Recognition". In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[5] Kuhn, Harold W. "The Hungarian Method for the Assignment Problem". In: (1955).

[6] Loshchilov, Ilya and Hutter, Frank. "Decoupled Weight Decay Regularization". In: *International Conference on Learning Representations (ICLR)*. 2017.

[7] Rezatofighi, Hamid et al. "Generalized Intersection over Union". In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019.

[8] Al-Rfou, Rami et al. "Character-level Language Modeling with Deeper Self-Attention". In: *AAAI Conference on Artificial Intelligence*. 2019.

[9] Rogge, Niels. *Hugging Face : Transformers : DETR*. https://huggingface.co/docs/transformers/main/model_doc/detr. 2022.

[10] Vaswani, Ashish et al. "Attention is All You Need". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2017.

[11] Wojke, Nicolai, Bewley, Alex, and Paulus, Dietrich. "Simple online and realtime tracking with a deep association metric". In: *2017 IEEE International Conference on Image Processing (ICIP)*. 2017.

[12] Wu, Yuxin et al. *Detectron2*. https://github.com/facebookresearch/detectron2. 2019.

Appendix A.

A.1 Training details

For the training of DETR, I employed the AdamW [6] optimizer with improved weight decay handling, set at 10^{-4}. Additionally, I implemented gradient clipping with a maximum gradient norm of 0.1. It's important to note that the backbone and the transformers were treated differently in terms of training, and I will delve into the specifics for each.

- Backbone : I utilized a pretrained ResNet-50 [4], ResNet-101, and ResNet-50 with Dilated Convolutions as backbone from Torchvision. To fine-tune the backbone, I set the learning rate to 10^{-5}.

- Transformer : The transformer was trained with a learning rate of 10^{-4}. Additionally, I applied additive dropout of 0.1 after each multi-head attention and feedforward network (FFN) before layer normalization.

- GPU : For all experiments and training procedures, the FAU TinyGPU Cluster is utilized, harnessing the power of 4 Nvidia A100 GPUs, each equipped with 40 GB of RAM. Given the transformer's flexibility in accepting various input sizes, I directly fed the images into the DETR model without any resizing.

- All models were trained with N = 5 decoder query slots, contributing to the overall configuration that enabled the successful training of DETR for our specific task.

A.2 Hyperparameters yielding optimum results

In below section, I present you the configuration file which yields best results on the hex-bug dataset :

"auxiliary loss": true, "bbox cost": 5, "bbox loss coefficient": 10, "class cost": 2, "d model": 256, "decoder attention heads": 8, "decoder ffn dim": 2048, "decoder layers": 6, "dropout": 0.1, "encoder attention heads": 8, "encoder ffn dim": 2048, "encoder layers": 6, "eos coefficient": 0.578, "giou cost": 2, "giou loss coefficient": 2, "id2label": "0": "LABEL 0", "init std": 0.02, "init xavier std": 1.0, "is encoder decoder": true, "num channels": 3, "num hidden layers": 6, "num queries": 5, "position embedding type": "sine", "scale embedding": false, "torch dtype": "float32"

A.3 PyTorch Code : Training Pipeline

As previously discussed, I leveraged the Hugging Face DETR open-source implementation for this project. I highly recommend adopting this implementation [9] [3], as it offers ease of use and facilitates seamless customization. The comprehensive documentation can be accessed at https://huggingface.co/docs/transformers/main/model_doc/detr.

In the subsequent sections, I will provide concise code snippets that will prove invaluable for constructing a tailored pipeline. Further, detailed code can be found at my GitHub repository : https://github.com/SalilBhatnagarDE/Hexbug_tracking.

A.3.1 Importing libraries and making custom dataset class

```
if __name__ == '__main__':
    import torchvision
    import os
    import torch
    import torchvision.transforms as transforms
    from torchvision.transforms import GaussianBlur
    import numpy as np
    import pytorch_lightning as pl
    from transformers import DetrForObjectDetection
    import torch
    import pickle
    from torch.utils.data import DataLoader
    from pytorch_lightning.loggers import TensorBoardLogger
    from pytorch_lightning import Trainer
    from pytorch_lightning.callbacks import EarlyStopping, ModelCheckpoint

    #Making custom dataset class
    class CocoDetection(torchvision.datasets.CocoDetection):
        def __init__(self, img_folder, processor, train):
            ann_file = os.path.join(img_folder, "new_train1.json" if train else "custom_val.json")
            super(CocoDetection, self).__init__(img_folder, ann_file)
            self.processor = processor
            self.train = train

        def __getitem__(self, idx):
            # read in PIL image and target in COCO format
            img, target = super(CocoDetection, self).__getitem__(idx)

            # preprocess image and target (converting target to DETR format,
            # resizing + normalization of both image and target)
            image_id = self.ids[idx]
            target = {'image_id': image_id, 'annotations': target}
            encoding = self.processor(images=img, annotations=target, return_tensors="pt")
            pixel_values = encoding["pixel_values"].squeeze()
            target = encoding["labels"][0]  # remove batch dimension

            # Data Augmentation applied
            if self.train ==True:
                p = np.random.rand()
                if p<=0.5:
                    transform = transforms.Compose([
                        transforms.ColorJitter(brightness=(0.8,1.5), contrast=(0.8,1.5),
                                               saturation=(0.8,1.5), hue=(-0.1,0.1))])
                    pixel_values = transform(pixel_values)

                if p>0.5:
                    noise_transform = transforms.Compose([
                        transforms.RandomApply([transforms.GaussianBlur(
                            kernel_size=5, sigma=(0.1, 3.0))])
                    ])
                    pixel_values = noise_transform(pixel_values)

            return pixel_values, target



    # Loading Preprocessor
    # Preprocessor downloaded from :
```

A.3.2 Defining data-loaders and making custom Pytorch Lightning class for training and evaluation pipeline

```
    # : DetrImageProcessor.from_pretrained("facebook/detr-resnet-50")
    with open("/home/hpc/iwb3/iwb3013h/Traco/org_size/Full_trainable/"
              "five_queries/auxloss/bboxloss_inc/processor.pkl", "rb") as f:
        processor = pickle.load(f)

    # Making datasets and dataloaders
    train_dataset = CocoDetection(img_folder='/./Traco/org_size/train', processor=processor,
                                  train=True)
    val_dataset = CocoDetection(img_folder='/./Traco/org_size/val', processor=processor,
                                train=False)

    def collate_fn(batch):
      pixel_values = [item[0] for item in batch]
      encoding = processor.pad(pixel_values, return_tensors="pt")
      labels = [item[1] for item in batch]
      batch = {}
      batch['pixel_values'] = encoding['pixel_values']
      batch['pixel_mask'] = encoding['pixel_mask']
      batch['labels'] = labels
      return batch

    train_dataloader = DataLoader(train_dataset, collate_fn=collate_fn, batch_size=14, shuffle=True,
                                  num_workers=32)
    val_dataloader = DataLoader(val_dataset, collate_fn=collate_fn, batch_size=14, num_workers=32)


    # Making DETR PyTorch Lightning Class
    class Detr(pl.LightningModule):
        def __init__(self, lr, lr_backbone, weight_decay):
            super().__init__()

            # Already downloaded pretrained DETR model
            # from Facebook git repository and saved at below location
            self.model = DetrForObjectDetection.from_pretrained(
                "./Full_trainable/five_queries/auxloss/bboxloss_inc",
                revision="no_timm", num_labels=1, ignore_mismatched_sizes=True)
            # Set all parameters as trainable
            for param in self.model.parameters():
                param.requires_grad = True

            self.lr = lr
            self.lr_backbone = lr_backbone
            self.weight_decay = weight_decay

        def forward(self, pixel_values, pixel_mask):
            outputs = self.model(pixel_values=pixel_values, pixel_mask=pixel_mask)
            return outputs

        def common_step(self, batch, batch_idx):
            pixel_values = batch["pixel_values"]
            pixel_mask = batch["pixel_mask"]
            labels = [{k: v.to(self.device) for k, v in t.items()}
                      for t in batch["labels"]]
            outputs = self.model(pixel_values=pixel_values, pixel_mask=pixel_mask, labels=labels)
            loss = outputs.loss
            loss_dict = outputs.loss_dict
            return loss, loss_dict

```

A.3.3 Defining model, loggers, callbacks, and trainer

```
        def training_step(self, batch, batch_idx):
            loss, loss_dict = self.common_step(batch, batch_idx)
            # logs metrics for each training_step,
            # and the average across the epoch
            self.log("training_loss", loss)
            for k, v in loss_dict.items():
                self.log("train_" + k, v.item())
            return loss

        def validation_step(self, batch, batch_idx):
            loss, loss_dict = self.common_step(batch, batch_idx)
            self.log("validation_loss", loss)
            for k, v in loss_dict.items():
                self.log("validation_" + k, v.item())
            return loss

        def configure_optimizers(self):
            param_dicts = [
                {"params": [p for n, p in self.named_parameters()
                            if "backbone" not in n and p.requires_grad]},
                {
                    "params": [p for n, p in self.named_parameters()
                               if "backbone" in n and p.requires_grad],
                    "lr": self.lr_backbone,
                },
            ]
            optimizer = torch.optim.AdamW(param_dicts, lr=self.lr, weight_decay=self.weight_decay)
            return optimizer

        def train_dataloader(self):
            return train_dataloader

        def val_dataloader(self):
            return val_dataloader


    # Make model as object
    model = Detr(lr=1e-4, lr_backbone=1e-5, weight_decay=1e-4)

    # Define loggers and callbacks
    early_stopping_callback = EarlyStopping(monitor='validation_loss', patience=15, mode='min')

    checkpoint_callback = ModelCheckpoint(
        dirpath="./iwb3013h/Traco/org_size/Full_trainable/five_queries/"
                "logstensorboard/DETR_Full_trainable_ogr_size_5_queries/",
        filename="detr-{epoch:02d}-{validation_loss:.4f}",
        monitor='validation_loss', mode='min', save_top_k=1, save_last=True)

    logger = TensorBoardLogger("/home/hpc/iwb3/iwb3013h/Traco/org_size/"
                               "Full_trainable/five_queries/logstensorboard/",
                               name="DETR_Full_trainable_ogr_size_5_queries")

    # Make trainer and fit model
    trainer=Trainer(gpus=4, max_epochs=60, gradient_clip_val=0.1,
                    callbacks=[early_stopping_callback,checkpoint_callback],
                    logger=logger, default_root_dir="./Full_trainable/five_queries/logs/")

    trainer.fit(model)
```

A.4 Code for associative algorithm

```python
# Get Final predictions after applying associative alogorithm

# Takes inputs : X as Videos, threshold value, and a list with predicted_num_of hexbugs per video
def get_final_predictions(X, threshold, predicted_hex):
    threshold = threshold
    final_predictions = []
    n_hex = predicted_hex

    # Iterate over each video
    for m, video in enumerate(X):
        tmp = []
        tmp_array = []

        # Iterate over each frame
        for i, img in enumerate(video):

            encoding = processor(images=img, annotations=None, return_tensors="pt")
            pixel_values = encoding["pixel_values"].squeeze()  # remove batch dimension
            device = torch.device("cuda")
            pixel_values = pixel_values.unsqueeze(0).to(device)
            model.to(device)

            with torch.no_grad():
                # forward pass to get class logits and bounding boxes
                outputs = model(pixel_values=pixel_values, pixel_mask=None)

            # postprocess model outputs
            width, height = img.shape[1], img.shape[0]
            postprocessed_outputs = processor.post_process_object_detection(outputs,
                                                                            target_sizes=[(height, width)],
                                                                            threshold=threshold)
            results = postprocessed_outputs[0]

            image_pred_history = []
            for ids, (score, label, (xmin, ymin, xmax, ymax)) in enumerate(
                    zip(results['scores'].tolist(), results['labels'].tolist(),
                        results['boxes'].tolist())):
                pred_n_hexs = len(results['scores'].tolist())
                center = (xmin + 32.5, ymin + 32.5)
                array = [int(i), int(ids), xmin + 32.5, ymin + 32.5]
                farray = np.array(array)
                frame, ids, x, y = farray
                # Check if predictions are less/ missing
                if int(ids + 1) <= n_hex[m]:
                    if int(ids + 1) <= 4:
                        # create a dictionary for each roi in the correct format
                        e = {
                            't': frame,
                            'hexbug': ids,
                            'x': y,
                            'y': x
                        }
                        image_pred_history.append(farray)
                        tmp.append(e)

            tmp_array.append(image_pred_history)

            if (int(ids + 1) < n_hex[m]):
```

```
                difference = n_hex[m] - int(ids + 1)
                for n in range(len(tmp_array) - 1, -1, -1):
                    if len(tmp_array[n]) == n_hex[m]:

                        associative_list = []
                        a_list = tmp_array[n]
                        # Assigns current prediction to previous predictions
                        # Iterate over elements of list with previous predictions
                        for b_element in image_pred_history:
                            min_distance = float('inf')  # Initialize minimum distance as infinity
                            associated_a_element = None  # Initialize associated A element as None

                            # Iterate over elements of list with current predictions
                            for index, a_element in enumerate(a_list):
                                if not index in associative_list:
                                    # Extract frame ID, hexbug ID, x, and y coordinates from list
                                    a_frame_id, a_hexbug_id, a_x, a_y = a_element

                                    # Calculate distance between B element and A element
                                    distance = calculate_distance(a_x, a_y, b_element[2], b_element[3])

                                    # Check if the distance is smaller than the minimum distance
                                    if distance < min_distance:
                                        min_distance = distance
                                        associated_a_element = a_element
                                        associated_a_element_index = index
                            associative_list.append(associated_a_element_index)

                        expected_size = n_hex[m]
                        expected_range = range(0, expected_size)
                        missing_elements = list(set(expected_range) - set(associative_list))

                        # For missed hexbugs, add new instances based on previous frames
                        for missed_number, missed in enumerate(missing_elements):
                            x = tmp_array[n][missed][2]
                            y = tmp_array[n][missed][3]
                            e = {
                                't': int(frame),
                                'hexbug': int(ids + missed_number + 1),
                                'x': y,
                                'y': x}
                            tmp.append(e)
                        break

                    else:
                        pass

        final_predictions.append(tmp)

    return final_predictions

```