Landmark Locator: Real-Time NYC Landmark Recognition with YOLO and RT-DETR Models

E6692.2024Spring.JWSS.report.jws2215
Johan Willem Schulz Sweldens jws2215
Columbia University

Abstract

This paper presents a software demo called Landmark Locator aimed at assisting tourists in identifying famous New York City landmarks in real-time without the need for an internet connection. The program utilizes object detection to identify ten major landmarks and runs locally on edge devices. The project incorporates three different models for inference: RT-DETR-l, YOLOv8l, and YOLOv8n, each offering unique trade-offs between accuracy and throughput. The paper outlines the methodology used in training these models, highlighting challenges such as dataset collection, model selection, and evaluation metrics. Additionally, the results of the trained models are discussed, comparing them with metrics from reference papers and evaluating their performance in live testing scenarios across various locations in Manhattan. The study concludes with insights into the discrepancy between observed throughput metrics and those reported in the original paper, suggesting areas for further investigation and refinement of the models.

1. Introduction

New York City is one of the most exciting travel destinations for tourists, whether they are from the United States or from around the world. They can see the many famous attractions, shows, musicals, museums, and landmarks that are scattered around the city. Often they walk around the streets of the city, admiring a tall skyscraper or unique architecture, without knowing what those buildings are. Without a tour guide, it may be difficult to know which famous landmark is in front of them.

This project aims to create a software demo that addresses these concerns, *Landmark Locator*. This program is able to do real time object detection on 10 famous New York City landmarks. It is designed to be run on an edge device: Jetson Nano, Laptop, or in the future a smartphone. It runs locally without the need for internet connection, which is likely very helpful for tourists from other countries. Additionally, the goal of the project is to develop a program that runs quickly, providing live inference while not using a great deal of memory. User intractability and information about the landmarks are seen as key aspects of the program.

To achieve this kind of real time object detection, the program allows the user to decide between one of three implementations to run inference on: RT-DETR-l, YOLOv8l, and YOLOv8n. These models are able to provide object detection to images with very quick throughput times. A fast frame rate is an essential part of the program design requirements. These models are trained beforehand on a virtual machine, and the program *Landmark Locator* does not need to install the dataset to do inference.

RT-DETR-l is a vision detection transformer model. YOLOv8l (large) and YOLOv8n (nano) are two different sizes of the same YOLOv8 real time object detection models. The project provides some quantitative and qualitative results on the live inference implementations of these models.

2. Summary of the Original Papers

2.1 Methodology of the Original Papers

YOLO, You Only Look Once, has been an accurate and quick object detection model since 2015. [4] It can provide high quality predictions while processing images with a high throughput. Recent developments in vision transformer architectures present a new method of object detection utilizing DETRs, Detection Transformers. Generally speaking, detection transformers are more accurate than equivalent YOLO models, however there is a significant trade off in throughput. Inference and training takes significantly longer in these detection transformer models. [3]

However, a 2023 paper from Baidu introduces the RT-DETR, Real Time Detection Transformer. It purportedly is able to benefit from the attention modules of the End-to-End Object Detectors while maintaining a 'real-time' processing speed, a best of both world approach. [3]

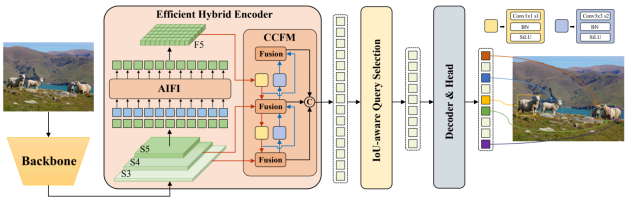

Figure 1: RT-DETR Architecture and Backbone. [3]

2.2 Key Results of the Original Paper

The Baidu paper states that their new detection models are both better than the traditional YOLO and DETR models for all major quantitative evaluation metrics while maintaining an equivalent number of parameters. The following models were trained on a T4 Nvidia GPU and the COCO 80 object dataset. [3]

Model	mAP	Params (M)	FPS	Throughput (seconds)
YOLOv8n	37.3	3.2	-	-
YOLOv8l	52.9	43.7	71	0.0141
RT-DETR-L	53.0	32.0	114	0.0088

Figure 2: Model metrics from Ultralytics and Baidu. The throughput is calculated on T4 Nvidia GPU with an input size of 640x640 pixels. [2] [3] [4] [5]

Model	Backbone	#Epochs	#Params (M)	GFLOPs	$FPS_{bs=1}$	AP^{val}
Real-time Object Detectors						
YOLOv7-L[38]	-	300	36	104	55	51.2
YOLOv7-X[38]	-	300	71	189	45	52.9
YOLOv8-L[14]	-	-	43	165	71	52.9
YOLOv8-X[14]	-	-	68	257	50	53.9
Real-time End-to-end Object Detector (ours)						
RT-DETR-R50	R50	72	42	136	**108**	**53.1**
RT-DETR-R101	R101	72	76	259	**74**	**54.3**
RT-DETR-L	HGNetv2	72	32	110	**114**	**53.0**
RT-DETR-X	HGNetv2	72	67	234	**74**	**54.8**

Figure 3: Baidu's results demonstrating the RT-DETR superior performance against comparable models. The FPS was calculated on a T4 Nvidia GPU with an input size of 640x640 pixels [3].

These metrics clearly state that the RE-DETR models have very high throughput and superior average precision. They indicate that this new model allows for greater object detection capabilities and actively increasing the throughput while maintaining a comparable number of parameters.

3. Methodology (of the Students' Project)

3.1. Objectives and Technical Challenges

The project used pretrained weights of the YOLOv8l, YOLOv8n, and RT-DETR that were provided by Ultralytics. These pretrained weights were trained on the 80 classes of COCO. The project implemented transfer learning and fine tuned the final layer of each model for the 10 classes of the custom New York City Landmark dataset. The models that are used in the software Landmark Locator were trained on 100 epochs, however some additional experimentation was done with 50 epochs.

These Ultralytics implementations of the models are convenient to use since they just require the download of the pretrained weights. They also provide a variety of training evaluation metrics and features: early stopping, saving best weights, normalization, and data augmentation. The Ultralytics models also include a data loader which takes in data in correct standard YOLO format. [4][5]

Due to the small amount of data available, there are concerns about overfitting. The entire dataset is only 1.2GB which is significantly smaller than 300k images of COCO. [6] Using pretrained weights and fine tuning, seeks to mitigate overfitting issues stemming from the small amount of data. Training on images of landmarks of New York City does have one caveat where overtraining may not be such a huge issue. In the world, there is only one instance of each landmark. This means that if the model overfits by learning to detect other buildings around the landmark or patterns in the skyline, then the model may actually be improving. This gives the model some wiggle room with regards to training that would not normally be present if doing traditional object detection on multiple world instances like everyday objects, people, or vehicles.

3.2. Methodology Compared to References

Each of the three models is trained for an equal 100 epochs irrespective of the evaluation metrics at those states. However, the models used in the *Landmark Locator* and in metric evaluations are the 'best' weights from the training loops. The models use a batch size of 16 and AdamW as the optimizer just the Baidu paper used for the RT-DETR models. [3] However, I used a learning rate of 0.000714 and a momentum of 0.9 which Ultralytics determined was the optimal hyperparameters for training the models. The input size was 640x640 just like the reference paper.

When evaluating my trained models, I use the same evaluation metrics stated by Baidu while also presenting different metrics. The evaluation metrics recorded in my training methodology are the mean Average Precision 50, precision, recall, box loss, GIoU loss, throughput speed, and FPS. The models were trained on a A100 Nvidia GPU from Google Cloud. However the throughput times and FPS were calculated on the Nvidia Jetson. Later speed metrics that were recorded while performing live testing, were performed on a Macbook Air, taking advantage of its Apple Silicon M2 Chip.

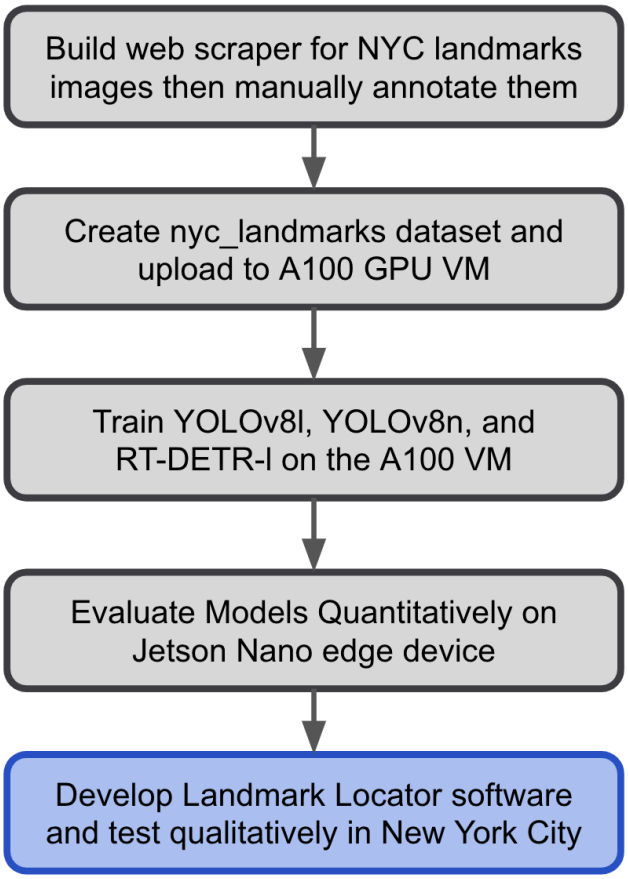

Figure 4: Project Workflow

3.3. Landmark Locator Design Requirements

The product demonstration, Landmark Locator, has to be able to fit a variety of design requirements that make it practical, intuitive, and fun for tourists to use. It is designed to provide real-time object detection for 10 major landmarks in New York City, offering a seamless and intuitive user experience. One of the primary design requirements is to implement object detection instead of image classification, enabling users to accurately identify landmarks from live video streams. These landmarks include iconic structures such as the Empire State Building, Freedom Tower, and Statue of Liberty, among others, ensuring comprehensive coverage of significant architectural landmarks across the city.

The app prioritizes real-time inference to deliver high throughput, allowing users to quickly and efficiently identify landmarks even in dynamic environments. Users are able to click on the detected image and have the respective wikipedia article about the landmark be opened. The user interface features a red button to exit the video profiling mode, enhancing user control and convenience. Moreover, the app is designed to detect landmarks from both street level and skyline views, catering to diverse user scenarios and providing accurate results regardless of the vantage point. Overall, the Landmark Locator app prioritizes user experience, accuracy, and real-time performance to offer a comprehensive solution for effortlessly identifying and exploring major landmarks in New York City.

4. Implementation

The implementation section is organized into three main sections: Data, Deep Learning Network, and Software Design. The process of collecting and annotating a dataset of ten major New York landmarks, highlighting issues such as dataset size, class imbalance, and outdated or doctored images is covered in the Data section. The Deep Learning Network section covers the architecture and functionalities of two models: RT-DETR and YOLO8. It explains how each model operates and its key features for object detection. Finally, the Software Design section describes the implementation of data scraping, dataset creation, model training, and real-time landmark detection using pre-trained models.

4.1 Data

The dataset is collected by scraping Bing for images of ten major New York Landmarks: Empire State Building, Freedom Tower, 432 Park Avenue, United Nations HQ, Flatiron Building, Brooklyn Bridge (Towers), Chrysler Building, Metlife Building, Statue Of Liberty, and 30 Hudson Yards. The dataset is collected by providing prompts for each of the 10 classes. The prompts include views from the street and views from the city. The images scrapped by the bing prompts are then manually annotated by me using Rectlabel. This process took a very long time. After the creation of the dataset, it was uploaded to Kaggle, so that it could be easily shared. [6]

There were a variety of issues with the dataset. The first is obvious, it is very small with the entire dataset only being around 1.2GB. Not all images are in a resolution that is above 640x640. If they were too small, I did not annotate them and that scrapped image was tossed out. [6]

Figure 5: A validation batch demonstrating the images scrapped and their drawn on bounding boxes. [1] [6]

The second issue is class imbalance. Often when people take pictures in New York City, they tend to try to include tall landmarks in the background, like for example the Empire State Building, World Trade Center, and Chrysler Building. For example, even if a scrapped image is supposed to be of the Brooklyn Bridge, the photographer likely wanted to include the World Trade Center prominently in the background. That kind of implicit bias is replicated in all classes. [6]

Additionally, some of the buildings are rather new: the 30 Hudson Yards and 432 Park Avenue. This leads to problems where often scrapped images of the other classes do not include them since they were not built at the time of the photos taken.

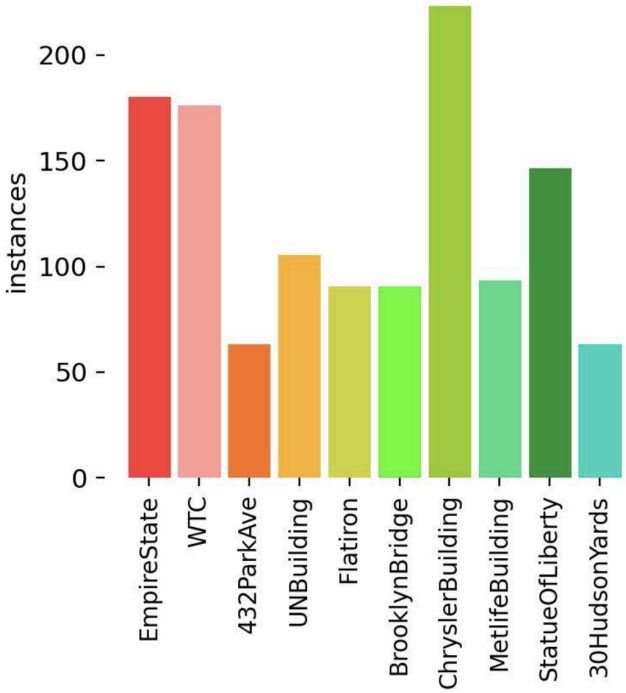

Figure 6: The dataset suffered from class imbalances, from a random set of 500 images, these were object instances in the total 500 image set. [1] [6]

Similarly, a great deal of scrapped images are rather dated. For example some feature the Twin Towers or the new Freedom Tower under construction. These are obvious tells that an image is dated, however dated images of the other classes are likely harder to spot. Additionally, some scrapped images were in black and white. Bing search is not that great.

Lastly, one of the most surprising issues was photoshopped/doctored images. This was only spotted with images of the Statue of Liberty, however this could be a problem with the other classes that were overlooked. The Statue of Liberty image below is clearly edited because in reality the statue faces south eastward, with its back to New Jersey. The image below has its back toward Tribeca which is incorrect. Some images had some similar issues, I suspect they may have been artistic renderings using architectural rendering software, however there was no way to be entirely sure. This seemed to be more of an issue with 30 Hudson Yards, which had lots of articles written about it during its construction phase. [6]

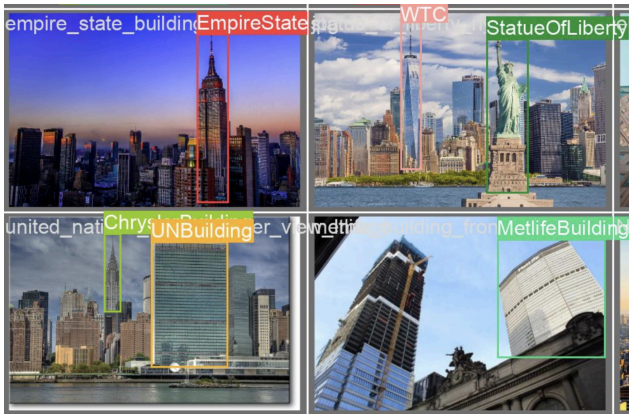

Figure 7: A validation batch demonstrating the images scrapped and their drawn on bounding boxes. Notice the image in the top right is photoshopped [1] [6]

4.2 Deep Learning Network

RT-DETR, or Real-time DETR, integrates a backbone, hybrid encoder, and transformer decoder with auxiliary prediction heads. The model architecture utilizes the output features of the backbone's last three stages to feed into the encoder, which transforms multi-scale features into a sequence of image features through intra-scale interaction and cross-scale fusion. The IoU-aware query selection mechanism is then employed to select a fixed number of image features from the encoder output sequence as initial object queries for the decoder. The decoder, equipped with auxiliary prediction heads, iteratively optimizes object queries to generate bounding boxes and confidence scores. To optimize computational efficiency, RT-DETR introduces an Efficient Hybrid Encoder, consisting of Attention-based Intra-scale Feature Interaction (AIFI) and CNN-based Cross-scale Feature-fusion Module (CCFM). AIFI reduces computational redundancy by performing intra-scale interaction on high-level features, while CCFM fuses adjacent features using fusion blocks composed of convolutional layers. Furthermore, the model employs IoU-aware query selection to ensure high classification scores for features with high IoU scores, enhancing detection performance. [3]

YOLO8 is a real-time object detection model that operates by dividing the input image into a grid and predicting bounding boxes and class probabilities directly from each grid cell. Unlike traditional models that perform classification and localization separately, YOLO8 predicts both simultaneously in a single pass through the network. The model architecture consists of multiple convolutional layers followed by a final detection layer that predicts bounding box coordinates and class probabilities. YOLO8 is known for its speed and efficiency, making it suitable for real-time applications. It utilizes a single neural network to predict multiple bounding boxes and their corresponding class probabilities, enabling rapid inference on various devices. Additionally, YOLO8 employs anchor boxes to improve localization accuracy and utilizes non-maximum suppression to remove redundant detections, resulting in precise and efficient object detection. [4]

4.3 Software Design

4.3.1 Scrapper and Dataset Creator

The first important piece of code written is in the legacy code folder. It is called the **data_training.ipynb.** This workbook had all of the initial efforts on the project that were completed for the project proposal stage. This workbook started by taking a variety of prompts for the landmarks, and passing them through a Bing Images scrapper. This would download roughly 100 images per prompt. This created a folder named after the prompt, populated with the scrapped images. Importantly, this scrapper downloaded the actual full size images, not just the thumbnails. While experimenting with different implementations, I tried to use a Google Images scrapper package, however it only downloads the thumbnails. The thumbnails are low resolution place holders. Once you click on them the real image is brought up. That made the Google scraper useless. This is unfortunate because Google Images provides better quality results compared to Bing Images. I also had tried using the scrapper code in **pretrained_deployment.py** from the lab, however that just downloaded the thumbnails as well. [1]

After running the scrapper and annotating the data, I ran **make_nyc_dataset.py (**which is stored in the utils folder). The make dataset function is fed in parameters about the classes, paths, and validation split to create the dataset in a standard YOLO format. The function is designed to create a dataset for NYC landmarks by organizing downloaded images and their corresponding labels into training and validation sets. Initially, it clears the existing contents of the specified training and validation folders. Then, for each folder within the downloaded images directory, the function processes the text files (.txt) containing labels. It calculates the number of text files to allocate to the validation set based on a specified fraction, and the remaining files are assigned to the training set. [1] The `process_file` function is called to copy each text file and its corresponding image (.jpg) to the appropriate destination folder, creating a new file name that combines the building folder name and the raw file name. Additionally, it checks for the existence of the corresponding .jpg file before copying, skipping the process if the image file is missing. Finally, the `clear_folder` function is called to ensure the destination

folders are empty before populating them with the new dataset.

At the end of the **data_training.ipynb** workbook, I attempted to train the newly created dataset on YOLOv9c. However these efforts failed on the VM since the models were too new. I decided to downgrade to YOLOv8 for the rest of the project after the proposal stage. That legacy effort is left in. Take note that that workbook is mainly legacy code, with the exception of keeping the end results of the completed nyc_landmarks dataset. [1]

4.3.2 Training the Models

The training code, provided in **training_models.ipynb,** handles training all three models: YOLOv8l, YOLOv8n, and RT-DETR-l. The Ultralytics training code requires an appropriately labeled file called **dataset.yaml** which provides the paths, configuration, and class labels for the model. The models are instantiated with pretrained weights, which can be one of the available options like 'yolov8l.pt', 'yolov8n.pt', or 'rtdetr-l.pt'. The model is then trained using the specified dataset, with parameters including the number of epochs, input image size, and device for computation. Finally, the training results are stored in the `results` variable for further analysis or evaluation. I did experiments training this dataset on my Apple Silicon M2 Chip and a T4 Nvidia GPU VM. The most time efficient device was an A100 GPU VM, which is what was ultimately used for training. The results of the training the models are in the Github in a folder called runs/detect. [1]

4.3.3 Landmark Locator

The main program, **landmark_locator.py** is designed to utilize either a plugin camera or a built in webcam for real-time detection and identification of various landmarks or buildings using pre-trained models. The user can select one of three modes: "detr" for RT-DETR model, "nano" for YOLOv8n model, and "yolo" for YOLOv8l model. The detected buildings are highlighted with bounding boxes on the video feed, and their names along with confidence scores are displayed next to the boxes. Each building is assigned a specific color based on a predefined dictionary, making it easier to distinguish between different landmarks. Additionally, the program allows users to click on the bounding boxes to open a web browser with relevant Wikipedia links for further information about the detected landmarks. Finally, the program calculates and prints the average model inference time and the average processing time per frame, providing insights into the performance of the detection process. To interrupt the execution, users can press the "q" key or click on an exit button displayed on the video feed.

*Figure 8: Screenshot of the program **landmark_locator.py** running. Notice the red exit square on the top left. Clicking inside the bounding box brings up the wikipedia page for respective objects. [1]*

5. Results

5.1 Project Results

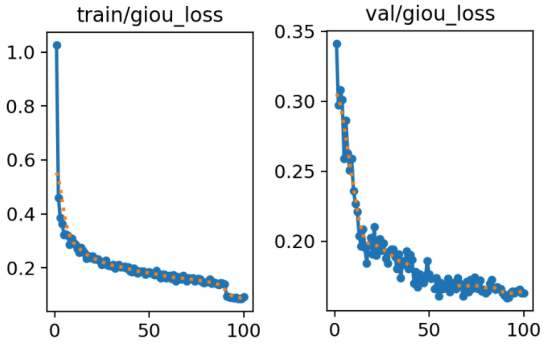

Figure 9: RT-DETR-l Training and Validation GIoU losses over 100 epochs. [1]

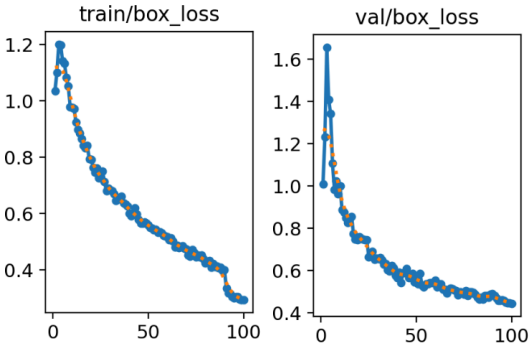

Figure 10: YOLOv8l Training and Validation Box losses over 100 epochs. [1]

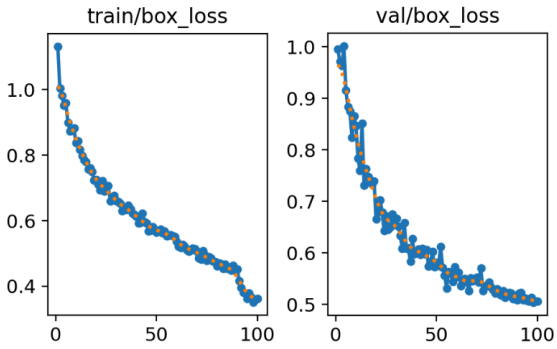

Figure 11: YOLOv8n Training and Validation Box losses over 100 epochs. [1]

All three models trained very well and did not have any major indications of overfitting. After the training, the Ultralytics package was able to provide evaluation metrics. The speed and FPS was calculated by running the models in the **landmark_locatory.py** on the Jetson Nano.

Metric	RT-DETR-l	YOLOv8-l	YOLOv8-n
mAP50	0.94508	**0.94747**	0.93104
Precision	0.97006	0.94926	**0.97758**
Recall	**0.92792**	0.92154	0.89755
Losses	0.09207 (GIoU Loss)	0.29474 (BoxLoss)	0.36148 (BoxLoss)
Speed	0.3038s	0.1127s	**0.02405s**
FPS	3.2916	8.8731	**41.5800**

Figure 12: Evaluation metrics observed for all three models. The best metrics are in bold. [1]

5.2 Comparison of the Results Between the Original Paper and Students' Project

Quantitatively, the evaluation metrics of the three models did not indicate that one was significantly better than the others. [1] RT-DETR-l had the highest recall score. YOLOv8-l had the highest mAP50. YOLOv8-n had the highest precision metric, additionally it had the highest throughput/FPS as expected. RT-DETR-l had the 2nd highest mAP50 and Precision score which both were not far behind the highest marks. This is different from the Baidu papers results which claimed that the RT-DETR would have the best evaluation performance metrics. [3]

What is most notable, is the poor throughput metrics for RT-DETR-l. While the Baidu paper's metrics were recorded on a Nvidia T4 GPU, the stated improvement was that the RT-DETR-l model would be a 60.56% improvement on the FPS compared to YOLOv8-l. The results observed on the edge device in *Landmark Locator* was that the RT-DETR-l model was the opposite with the model significantly slower than the YOLOv8-l model. In fact the RT-DETR-l model was observed to be 62.90% slower than the YOLOv8-l model. While the evaluation metrics showed that the RT-DETR-l model generally performed very well and occasionally better than the other models, the throughput metrics are very far off the relative stated performance from the Baidu paper. [1][3] This comparison between the results gathered in this project and the original paper introduced a large discrepancy that warrants further exploration.

5.3 Live Testing in New York City

In order to get both quantitative and qualitative evaluations on the three models, I tested the program *Landmark Locator* on an edge device live in New York City. I went to a variety of locations in Manhattan to perform this evaluation testing: Long Island City, Park Avenue, Lexington & 41st, Madison Square Park, Chelsea Park, Pier 15, and Battery Park. The notes, recorded screen captures, and downloaded GIFS for the powerpoint are in the Github repository in a folder called **landmark_videos** [1].

5.3.1 Model Comparisons

Running the program in these locations provided some qualitative information about the effectiveness of the three models. In general, the RT-DETR-l model was the best and most consistent model when it came to correctly and stably identifying the landmarks. The bounding boxes appeared to have a 'stickiness' to them that gave the model a lot of credibility as the best choice for the program. However, the frame rate was significantly slower than the other models. This took away from the practicality of the application. A fully developed implementation would need to continue providing the live feed while applying the bounding boxes as they get detected.

The YOLOv8n was found to be very jittery. The bounding boxes would jump around alot with its shape rapidly changing even if the camera was held still. Sometimes it would fail to detect an object while the other models were able to detect them. It also periodically incorrectly described items/buildings in the frame as one of the 10 classes. The high FPS of around 35 to 45 while testing was greatly appreciated. That feature allowed the model to have a great Augmented Reality experience. The high frame rate was a great benefit to the user experience. Future implementations of the application should take advantage of possible optimizations to detect models to ensure the frame rate is as high as possible. This feature made the model a very attractive choice even if it was consistently the worst option with regards to correctly detecting the landmarks.

The YOLOv8l was in general a middle ground between the two models. It was faster than the RT-DETR model and slower than the YOLOv8n model. However it

was less accurate than the RT-DETR model and more accurate than the YOLOv8n model. [1] The frame rate of around 8 FPS was high enough for the model to feel operational. The 3 FPS of RT-DETR-l model was in fact so slow, the application often had trouble closing, a problem that both YOLO models did not suffer from. The speed of YOLOv8l felt like a safe operational minimum. Anything slower than the 8 FPS took away from the practicality and fun of the application.

5.3.2 Location Results

A video compilation of the screen captures taken during the testing is available on the Youtube page in the references part of this paper. [7]

The first location tested was the park in Long Island City. It was cloudy during the evaluation, which may have had an effect on the performance of the models, since they were all trained on clear days. All models detected the United Nations HQ, the Chrysler Building, and the Empire State Building. However none of them detected 432 Park Avenue or the Metlife Building.

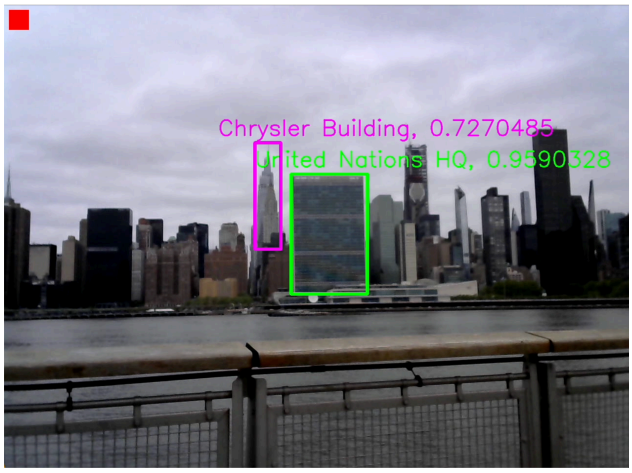

Figure 13: RT-DETR-l detection of the Chrysler Building at 72.7% confidence and United Nations HQ with 95.9% confidence level. 432 Park Avenue failed to detect [1]

The second location tested was Park Avenue facing towards Grand Central Station. All models correctly detected the Metlife Building, however a tree and landpost in the way caused some detection glitches. There also was a reflection of the Metlife Building in the left of the frame off the glass of another building. However, that was ignored by all of the models.

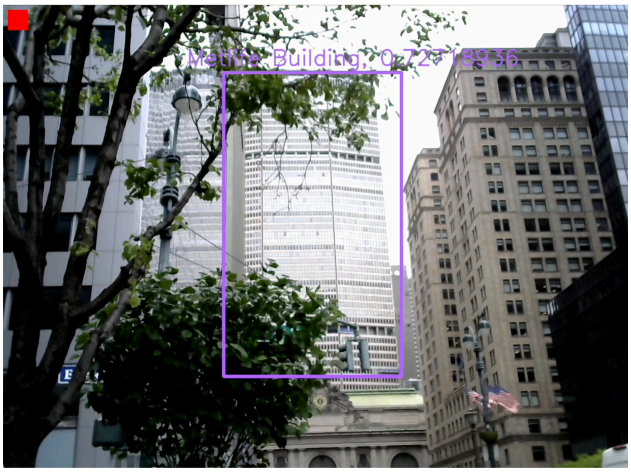

Figure 14: YOLOv9n detection of the Metlife Building with 72.7% Confidence Level. [1]

Afterwards, testing at Lexington & 41st street provided clear detection of the Chrysler building with no issues from any model dispute the fact there was a street sign in the way of the image. Many images in the dataset were from that angle, which likely led to the high confidence level.

Figure 15: RT-DETR-l detection of the Chrysler Building with 92.7% Confidence Level. [1]

Later, the results from testing at Madison Square Park were mixed. The Flatiron Building that day was completely covered in scaffolding with supposed major restoration work being done on the facade. YOLOv8n failed to detect the building at all while both the YOLOv8l and RT-DETR-l models succeeded in detection. The RT-DETR-l detection was superior with the resulting bounding box having no jittery effects. The model seemed unperturbed by the scaffolding.

Figure 16: RT-DETR-l detection of the Flatiron Building with 93.1% confidence level [1]

At Chelsea Park, there was a great view of 30 Hudson Yards. However, this view never appeared in any of the pictures that were scrapped from the internet. All the models failed to detect the building even though it was in clear plain view on a sunny day. Given more data, the models should reasonably be able to detect the building. Sitting at the start of the High Line around 34th street, the two YOLO models were able to detect the building when the confidence requirement for display was dropped to 30%. The RT-DETR Model never detected the building and in fact, incorrectly assumed it was the Empire State Building. Overall, all of the models performed poorly on 30 Hudson Yards. [1]

Figure 17: View of 30 Hudson Yards from Chelsea Park. No models detected the building from this location. [1]

Figure 18: RT-DETR-l incorrectly labeling 30 Hudson Yards at 53.0% confidence level. [1]

The last successful tests were performed at Pier 13 by the East River. All of the models correctly identified the east Brooklyn Bridge Tower. The west tower was out of view due to a parked boat in the way. The YOLOv8n capture was rather jittery again at this location. [1]

A final test was conducted in Battery Park, however none of the models detected the Statue of Liberty. It was likely too far away and too small for these models to detect. Additionally no images in the dataset were from that angle.

5.4 Discussion / Insights Gained

Live testing demonstrated the feasibility of the project. Overall the program worked quite well, especially when using the higher quality built-in webcam on Macbook Air. The program was able to run locally using the M2 Chip and performance was decent. The RT-DETR models were qualitatively the best at detecting objects while the speed YOLOv8n added a great deal of practicality to the augmented reality part of the program. However there were a few insights that were noted from the live testing.

The first was that the model suffered a great deal when the camera was tilted at an axis. All of the data it was provided is completely flat, so when the camera is held at even a small angle <10 degrees, the predictions are widely thrown off and the confidence level drops significantly.

The second issue was generally the model failed to predict the buildings when looking at an angle that was not present in the dataset. This was especially present for the Statue of Liberty, 30 Hudson Yards, and 432 Park Avenue.

6. Future Work

This project overall acted as a demonstration of the object detection technologies and possibilities of implementing them into a tourist app run on an edge

device. To properly develop this application, there would need to be a significantly larger dataset. There would need to be around 100 big and small New York City Landmarks to make the application more interesting: Bow Bridge, Times Square, Rockefeller Plaza, New York Stock Exchange, Washington Square Park, Charging Bull, Yankee Stadium and much more. Each item would need a minimum of 500 images per class from a variety of angles, positions, and weather conditions to ensure better training. All the pictures would also need to be of good quality and taken within the last year to prevent training on pictures that are decades old.

Training should also implement a rotation data augmentation to allow for the model to predict the classes even if the user is holding the camera at an angle. Performing this type of data augmentation will also provide more data for the models to train on.

Right now the program will open a link to wikipedia when the box is clicked on. However, it would be more interesting if there was a Small Langue Model running locally that would provide some information about the NYC buildings when prompted.

7. Conclusion

In conclusion, this project aimed to explore the efficacy of modern object detection models, specifically RT-DETR and YOLOv8, in the context of landmark recognition within New York City. The objectives were to evaluate the performance of these models, develop a real-time landmark detection application, and conduct live testing to assess practicality and accuracy.

The methodology involved utilizing pretrained weights for YOLOv8 and RT-DETR provided by Ultralytics, fine-tuning these models on a custom dataset of New York City landmarks, and implementing a real-time detection application called Landmark Locator. Despite challenges such as a small dataset size, class imbalance, and outdated or doctored images, the models were trained successfully, and the application was developed to meet design requirements. [1]

Evaluation of the trained models revealed interesting insights. While RT-DETR showed promising results in terms of accuracy, with superior recall scores, YOLOv8 models demonstrated higher precision and throughput. However, there was a significant discrepancy between the observed throughput metrics and the stated performance in the original RT-DETR paper, highlighting the need for further investigation. [1] [3]

Live testing in various locations across New York City provided qualitative insights into the models' performance in real-world scenarios. RT-DETR exhibited stable and accurate detections, albeit with slower frame rates, while YOLOv8 models showed faster frame rates but occasionally jittery detections.

Lessons learned from the project include the importance of dataset quality, model robustness to camera angles, and the trade-offs between accuracy and speed in real-time applications. Future research should focus on expanding the dataset with more diverse landmarks, implementing data augmentation techniques, and improving model robustness to varying camera angles. Additionally, integrating a local language model for providing landmark information and optimizing the application for better real-time performance would enhance user experience.

Overall, this project serves as a demonstration of the potential of object detection technologies in tourist applications and lays the groundwork for future improvements and research in this domain.

8. Acknowledgement

I am grateful to the TAs in EECS6691 and EECS6692 for their invaluable support throughout the semester. Their guidance and expertise have greatly enriched my learning experience. Thank you for your dedication and assistance.

9. References

[1] Sweldens, "e6692-2024spring-project-jwss-jws2215," GitHub, 2024. [Online]. Available: https://github.com/eecse6692/e6692-2024spring-finalproject-jwss-jws2215/tree/main.

[2] "E6692.2024Spring.jwss.jws2215.presentationFinal," Google Slides, 2024. [Online]. Available: https://docs.google.com/presentation/d/1nf4o6lzg3G4qzuBm4S9ptLFn0J1ISrnKHYaed3WLa_c/edit#slide=id.p1.

[3] Zhao, Y., Lv, W., Xu, S., Wei, J., Wang, G., Dang, Q., Liu, Y., & Chen, J. (2024). DETRs Beat YOLOs on Real-time Object Detection. arXiv preprint arXiv:2304.08069v3 [cs.CV].

[4] Ultralytics. (2024). Ultralytics YOLOv8. [Online]. Available: https://github.com/ultralytics/ultralytics

[5] Ultralytics. (2024). RT-DETR (Realtime Detection Transformer). [Online]. Available: https://github.com/ultralytics/ultralytics

[6] JWS2215. (2024). NYC Landmarks Object Detection [Dataset]. Kaggle. https://www.kaggle.com/jws2215/nyc-landmarks-object-detection

[7] Sweldens, Johan Willem Schulz. "YouTube Channel: JohanWillemSchulzSweldens." YouTube, www.youtube.com/@JohanWillemSchulzSweldens.

10. Appendix

10.1 Individual Student Contributions in Fractions

	jws2215
Last Name	Sweldens
Fraction of (useful) total contribution	1/1
What I did	Entirety

Figure 19: Teammate contribution table