CS153 Final Project Report: Comparative Analysis of Object Detection Models on J-EDI Trash Dataset

Stephanie Fulcar
Harvey Mudd College
sfulcar@g.hmc.edu

Julia Hansen
Harvey Mudd College
juhansen@g.hmc.edu

1. Motivation

A popular application of computer vision is in robotics and automation. Robots such as autonomous underwater vehicles (AUVs), drones, and remote sensing robots can be equipped with cameras and sensors which can collect data from the surrounding environment. That data can be inputted into machine learning models to make real time predictions according to the task at hand. Because real time computational efficiency is important while also retaining accuracy, it's important to track the memory performance of various models in order to determine which models are most compatible with battery operated systems while also performing well in object detection. In our project, we seek to test different pre-existing object detection models on different hardwares to evaluate which is best at detecting underwater plastics and most efficient.

2. Background

Common processors used in robots include central processing units (CPUs), graphics processing units (GPUS), tensor processing units (TPUs), digital signal processors (DSPs), and field programmable gate arrays (FPGAs) [5]. These processors are used for a variety of tasks such as fast signal processing and object classification. The application of AI in robotics falls under the category of Edge AI which refers to running AI models on devices which rely on battery power and size constraints [3]. In this project we will focus on Edge AI and aim to explore the performance of models on various processors, primarily GPUs and TPUs.

Previous research has been done to evaluate computer vision models in robotics applications. In this section we will explore the research and datasets available on this topic.

2.1. Dataset and Previous Research

In 2019 Michael Fulton et al published a paper titled "Robotic Detection of Marine Litter Using Deep Visual Detection Models", in the IEEE International Conference on Robotics and Automation [2]. They used the J-EDI dataset of marine debris taken from the deep seas of Japan. The older version of this dataset contains 5,720 images with labels of "bio" or "object" while the newer dataset [4] contains 7,212 images with 16 different labels.

The authors used four different models: YOLO v2, Tiny YOLO, Faster R-CNN with Inception v2, and Single Shot Multibox Detector (SSD) with MobileNet v2. All four of these models were compared for Mean Average Precision (mAP), Intersection over Union (IoU) scores, and processing times. The authors found that the Faster R-CNN and SSD models resulted in a higher mAP score (81.0% and 67.4% respectively), but had a slower processing time. We will focus on YOLO and Faster R-CNN in greater detail in the following subsections due to time constraints.

2.2. Model 1: YOLOv11 Model

The You Only Look Once (YOLO) Model is a object detection model originally developed by Joseph Redmon et al. in 2016 [6]. This model uses a single fully Convolutional Neural Network (CNN) to predict object bounding boxes and class predictions from an input image. Unlike some other object detection models YOLO makes these predictions after only doing a single evaluation of the input image, making it faster and less computationally expensive than existing models at the time and therefore a popular choice for robotics applications. There have been many versions of YOLO created since its introduction in 2016—for this project we evaluated version 11, the lastest model to have been released.

2.3. Model 2: Faster R-CNN

Faster R-CNN [7] is an improvement on the original R-CNN model. R-CNN introduced a Region Proposal Network (RPN), a fully convolutional network that predicts object bounding boxes and objectness scores. Unlike the previously discussed models Faster R-CNN uses two-shot object detection, meaning that predictions are made using two passes of the input image through the network. The first pass is used to determine potential object bounds and the second pass is used to refine and make final predictions. This double pass methods makes this model more accurate

than single pass methods but is also more compuatationally expensive. For this project we evaluated Facebook Detectron2's implementation of the model.

3. Methodology

We replicated parts of the study from the paper, but with newer versions of two of the models and tested them on three different kinds of hardwares: CPU, GPU, and TPU.

The first step was training our models on the J-EDI dataset using the GPU. From there we analyzed test time inference and memory usage and compared results across the two models while also keeping in mind performance metrics such as accuracy. We used Ultralytics to run YOLO v11 and Detectron2 to run Faster R-CNN. Both of these libraries log the memory usage and timing of the models while training. We used the script command in the server terminal before training each model so that these logs would be saved to our server. With both models Roboflow was used to easily format and load in our data into the server.

Once both models were finetuned we compared the logged GPU usage and mean precision accuracy (mAP) for an IOU threshold of 0.5. Then we ran inference on our validation set using first the GPU, then the CPU, and lastly the TPU. We compared the timing and memory performance of both models running on all three hardwares.

4. Metrics

We evaluated both models' performances based on two metrics: compute time and Mean Average Precision (mAP).

4.1. Metric 1: Compute Time

We tested our models on three types of hardware detailed in the Methodology section. By running inference of our models on these three different hardwares we were able to analyze differences in the time it took each test to run across the CPU, GPU, and TPU.

4.2. Metric 2: Mean Average Precision (mAP)

Mean Average Precision (mAP) is a common metric used to evaluate the performance of machine learning models. It is calculated by taking the mean average precision over all classes in the dataset, evaluated at different recall thresholds. Here precision is defined as the ratio of True Positives over the total number of Positive (True and False) made by the model. This is one of the metrics used in the study we are replicating, so we are interested in seeing not only how the models compare to each other, but also how they compare to the original study.

5. Challenges

In this project, we faced several challenges with our dataset, model training, and hardware compatibility which we will describe in the sections below. Because of the issues we faced, we were not able to accomplish as much as we originally intended to in our proposal.

5.1. Dataset Challenges

The J-EDI dataset has some data leakage present in that different frames from the same videos are included in both the training and validation sets. Although this is not as problematic as having the same frames in both sets, we wanted to try cleaning our data so that the model would not see similar images in the validation phase as it saw in the training phase.

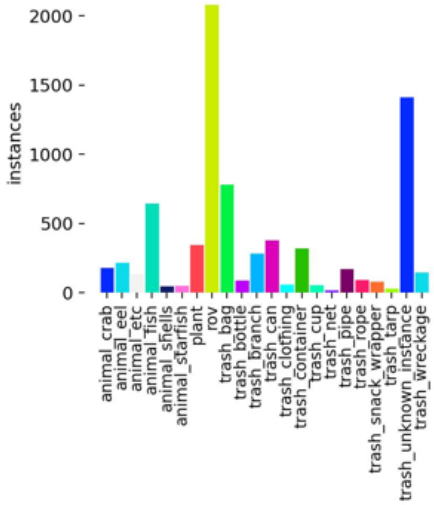

Figure 1. Class Imbalance in Manually Rearranged Training Dataset

To do this we used the COCO dataset API to reformat our data since our dataset is in COCO format. We first manually rearranged the videos to have entirely different videos in each set. Then we used the Json library to extract our desired image IDs and save them to a file according to the new frames present in our training and validation images folders. Lastly we used the COCO API to filter out the annotations that corresponded to the desired image IDs. We then finetuned YOLO v11 with this updated dataset over 25 epochs. However, the mAP was very small (approximately 5%) and even after running for 150 epochs it only increased to 15%. One of the results from the training showed the class distribution and we found it to be imbalanced, with ROVs and unknown trash instances having a much higher representation.

We went back to our original dataset and isolated a whole video from the training set into a separate test set in order to see how well the model could generalize to unseen data since that was our concern with the data leakage. Yet, we found that it could generalize well and was able to detect

most of the objects with relatively high confidence as seen in Figure 1. Because of this result and because our focus is on model comparison and not cleaning the dataset, we decided to revert back to using the original structure of the dataset.

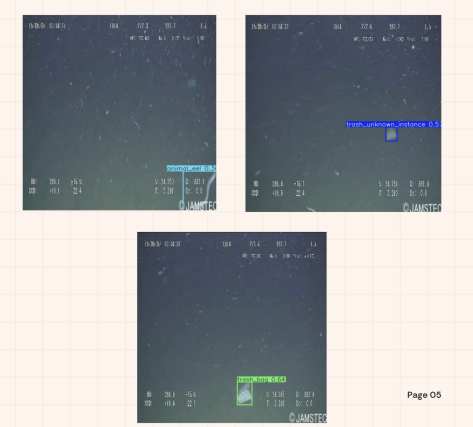

Figure 2. Results of testing YOLO v11 on the original dataset with one video extracted.

5.2. Detectron2 Challenges

For this project, we chose to use Facebook Detectron2's implementation of the Faster R-CNN model because that is model implementation that was originally used in the paper. However we faced some issues while attempting to install the Detectron2 library on the class server. The version of gcc running on the server was not compatible with the Detectron2 installation files which prevented us from directly installing the library and the model. The system administrator (shoutout Tim) helped us work around this and created a virtual environment for us that was able to run Detectron. Ultimately we were able to run the model, but because of the unexpected delay we were unable to test the third model as we intially proposed.

5.3. Hardware Challenges

Initially we intended to run inference for each of our models on a Raspberry Pi to test and compare their performances on a TPU. We were able to gain access to a Raspberry Pi but were unable to properly interface with the hardware, so ultimately we were unable to run our models on it. Instead we pivoted to using the v2-8 TPU runtime type on Google Colab to test the TPU performance of our models, but are aware that these results may be more unreliable since we are not directly connected to the hardware.

6. Results

6.1. YOLOv11 Performance

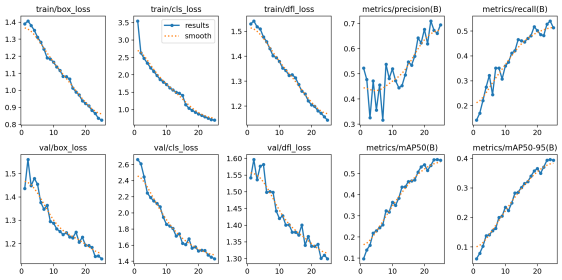

Figure 3. YOLOv11 model training results. The first 3 columns contain the training and validation loss curves for the 3 loss functions that YOLO uses. Precision and recall curves are show in the top row of columns 4 and 5 and the mAP curves are show in the bottom row of columns 4 and 5.

We finetuned YOLO v11 on our dataset over 25 epochs and found that the mAP was 56.4%. Compared to the results presented by the Fulton et al paper, this accuracy is 12.5% higher which is understandable since they used YOLO v2 and we used version 11. The results from the v11 model are shown in Figure 3.

Once the accuracy was determined, the model was tested on the CPU, GPU, and TPU. It was tasked with validating 1,147 images. While training the model used up 0.793 gigabytes per epoch. The compute times are summarized in Table 1.

Hardware Type	CPU	GPU	TPU
Compute Time (s)	833.98	10.26	235.135

Table 1. Compute time comparison across different hardware types.

6.2. Faster R-CNN Performance

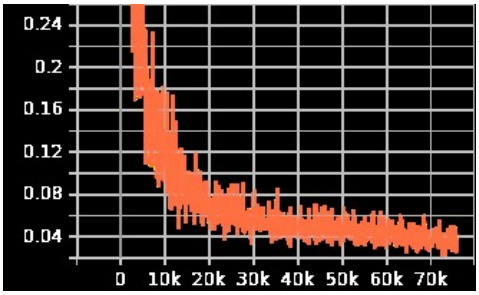

Figure 4. Faster R-CNN loss curve where the x-axis represents number of iterations and the y-axis represents mAP.

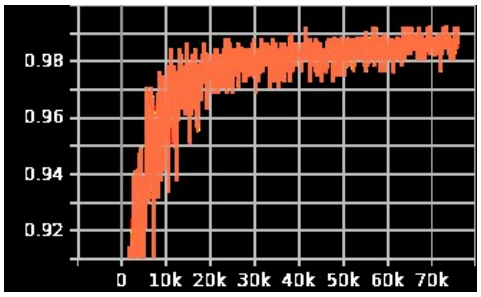

Figure 5. Faster R-CNN mAP curve where the x-axis represents number of iterations and the y-axis represents mAP.

We finetuned Faster R-CNN on our dataset over 74,200 iterations which is the equivalent of 25 epochs and found that the mAP was 98.92%. Compared to the results presented by the Fulton et al paper, this accuracy is 17.92% higher which is again understandable since we used an updated version of Faster R-CNN. The results from this model are shown in Figure 3.

Once the accuracy was determined, the model was tested on the GPU. When we tried to test on the CPU and TPU we encountered an error saying we could not test the model without a GPU in the Detectron2 environment. However, we had previously run a PyTorch implementation of Faster R-CNN and we were able to use that library to test our model on the CPU and TPU. This model had extremely low accuracy, which is why we chose to move forward with the Detectron2 implementation instead, but we believe it can still give us an estimate of how long the model would take to run on the hardware. We ran inference on 1,147 images. While training, the model used up 1.541 gigabytes per epoch. The compute times are summarized in Table 2.

Hardware Type	CPU	GPU	TPU
Compute Time (s)	15,482.24	265.74	6,912.88

Table 2. Faster R-CNN compute time comparison across different hardware types.

7. Conclusion

From our results we found that the YOLO v11 model is faster, but less accurate than the Faster R-CNN model. This is as expected since YOLO models are designed to run more quickly. We saw an enormous difference between the CPU inference times for the YOLO verus Faster R-CNN model. Running on the TPU was faster than running on the CPU, but it was still much slower than running on the GPU which was surprising. However, as stated in section 5.3 we are aware that the TPU offered on Google Colab might not be the most reliable meaning these results might not be very accurate. Even though the inference time on the GPU for the YOLO v11 model is faster than the inference time for the Faster R-CNN model, if accuracy is valued much higher than speed and GPU hardware is available in the robotic system, then deploying the Faster R-CNN model on GPU could be a viable option.

In the future we think it would be interesting to run both models on a Raspberry Pi if a monitor with an attached keyboard and mouse is available as described in section 5.3. This would provide insight into both model's performances on a micro-controller which are often used in robotics.

We would also suggest further investigating the structure of the dataset. One could use data augmentation to increase the size of the under represented classes in the dataset and then run both models on the updated dataset to see if they perform better.

Lastly it would be interesting to compare more models to each other. We would suggest first starting with the SSD model since that is what was used in the paper before testing other models such as State Space Models (SSMs) which are derived from control theory applications and could be more efficient in robotics applications [1].

References

[1] Loick Bourdois. Introduction to state space models (ssm). *Hugging Face*, 2024. 4

[2] Michael Fulton, Jungseok Hong, Md Jahidul Islam, and Junaed Sattar. Robotic detection of marine litter using deep visual detection models. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 5752–5758, 2019. 1

[3] GeeksforGeeks.com. Hardware requirements for artificial intelligence. 2024. 1

[4] Jungseok Hong, Michael S Fulton, and Junaed Sattar. Trashcan 1.0 an instance-segmentation labeled dataset of trash observations, 2020. Retrieved from the Data Repository for the University of Minnesota (DRUM). 1

[5] Jagriti Patidar. Heterogeneous architectures for robots: Combining cpus, gpus, and specialized processors, 2024. 1

[6] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015. 1

[7] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015. 1