# You Only Look Once Machine Learning Solution to Orbital Debris Detection and Classification

New Mexico
Supercomputing Challenge
Final Report
April 2, 2025

*La Cueva High School*

*Team Members:*
**Hadwyn Link**
**Ximena Serna**

*Teacher:*
**Jeremy Jensen**
*Project Mentor:*
**Mario Serna**

# Table of Contents

## Abstract

As the amount of debris in Low Earth Orbit (LEO) increases, satellites are more likely to collide with it. Debris travels at such high speeds that even small debris colliding with a satellite could cause catastrophic damage. As satellites are crucial to many important systems, it is important to protect these satellites. The common methods of preventing collision are by either removing the debris, forcing it to re-enter the atmosphere and burn up, or by maneuvering satellites around debris to avoid it entirely. Additionally, it is important to be able to classify objects in orbit to catalogue the type of debris. Classifying debris allows us to better determine risk and which method of removal to use. However, both classifying and detecting debris are extremely difficult with the current strategies. By using novel machine learning algorithms to more efficiently analyze debris classes and orbits, we can vastly improve the performance of satellite systems and debris removal protocols.

Recent studies pertaining to this field suggest trying You Only Look Once (YOLO), a new type of machine learning. It is extremely fast at detecting objects and works much more efficiently than any previous object detection models, making it more likely that it would work on less powerful space hardware. Our goal was to find how effective YOLO is at classifying debris in LEO.

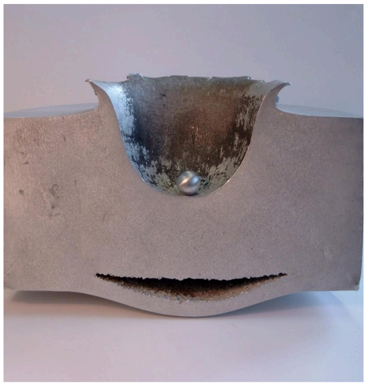
Our solution was to create a simulation which can test YOLO's ability to classify debris. To do so, we generated over 14,000 orbits containing information such as position versus time. With this information, we created a graphical simulation of the debris orbiting Earth. Then, we procedurally took screenshots of the simulation and saved object classes and positions to a file. After slight adjustment, the images were fed into the YOLO program for training, and once it was fully trained we fed the simulation directly into the model to see how it would perform in a real-time environment as opposed to still pictures.

The program finished its training with a very high percentage accuracy of classification. The program had minimal difference between the different types of background in the images. Our results support the conclusion that YOLO can classify debris accurately and can be implemented for the purpose of addressing free floating debris in space. We encourage future researchers to continue this course of study for possible space implantation.

# Introduction

## The Problem

In the past few decades, society has grown to rely on wireless internet, smart phones, and instantaneous communication networks. Many of these commodities, however, rely on satellite technology. With most of the modern world dependent on satellites in some way, it is important to address problems that threaten their existence. By far the largest one is debris; these pieces of residual payloads, broken satellite parts, and asteroids have begun to litter Earth's orbit on an unprecedented scale. As more satellites are put in space, more debris is created: more payloads are left over, and more old satellites are forgotten and broken down. A piece of debris can reach a speed of 18,000 miles per hour, seven times faster than the speed of a bullet. At that velocity, even a collision between a flake of paint and a satellite could prove catastrophic. The image to the left demonstrates the proportion size between a piece of debris in Low Earth Orbit and its collision impact crater. A single collision could easily decommission a satellite, as well as create more debris to worsen the problem. Repairing damaged satellites costs tens of billions to hundreds of millions  of dollars, especially if it requires a human mission. Leaving them vulnerable jeopardizes every function they perform, is a danger to society's way of life, and makes sustainability in space very precarious.



Current methods to detect debris are radar systems which can pinpoint locations of nearby objects as well as contact with a ground-based tracking system with more heavy-duty tools. Neither of these methods can directly determine what exactly the debris is but can collect information such as material. However, in some cases this may not be enough to fully classify the debris, and in the case of ground-based tracking it has a delay between data transfers and can also be very expensive to operate the ground-based tracking systems. Another, newer option is to stream images down from the satellite and perform object detection algorithms on earth. However, streaming such a large amount of data from a satellite would be very expensive and would have a large enough delay for the data to be outdated when it reaches the satellite–remember, debris can move at up to 18,000 miles per hour.

Without information about the debris near a satellite in real time, there is little we can do to reduce collisions. Solutions such as removing the debris, forcing their orbit to burn on reentry, or having satellites maneuver around debris would all require information about the class and location of the debris that would be necessary in order to determine which approach to use. Clearly, another method is necessary to accurately and completely classify and detect debris with attention to both cost and speed.

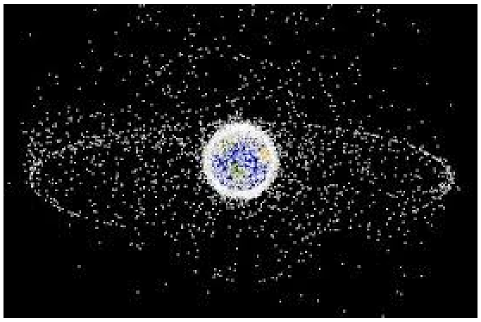
## The Objective

Our proposed solution is to find a way of detecting and classifying pieces of debris with a common type of artificial neural network: YOLO. First introduced by Joseph Redmon et al. in 2015, it is a quick and efficient object detection neural network. It has never been implemented in space before but has been proposed for being more effective than current methods of detection according to studies by Mahendrakar et al. and Ahamed et al. However, no previous studies have suggested any ability of YOLO's classification skills in space.

Classification of debris could help us understand the risk of each piece of debris, provide information on the best way of removal, and gauge whether the debris would burn up in the atmosphere during reentry. Providing this information would be valuable for addressing the debris problem and is an area that has not yet been explored in YOLO's abilities, thus addressing a gap in scientific understanding.

YOLO sets itself apart from many other machine learning algorithms because it is the most likely object detection model to work on current satellite hardware because of its speed and relatively low resource requirements. Realistically, a surveying satellite does not have the time to be able to detect and classify every object it sees. Instead, it has to be able to process the required information before the object disappears. Speedy calculations will lead to a more efficient system of surveying and give the satellite a greater amount of time to act and attempt to remove or avoid the debris. Because LEO has the highest concentration of debris, our simulations have been based on receiving its data from this altitude (2,000km or less). At the conclusion of this study, our results should show how effective YOLO is at detecting and classifying debris in 7 classes: Asteroid, Cube Satellite, Envisat, Voyager, Hubble, the ISS, and the SaturnV5 Rocket. These classes include common objects seen in space along with specific, universally recognised objects that would demonstrate YOLO's ability to classify different types of objects such as Envisat, Voyager etc.

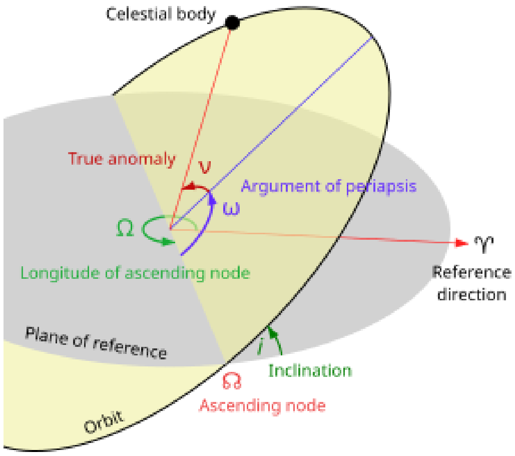
By separating the project into two sections: generating the debris orbits, and creating and training the YOLO program. We generated the debris orbits based on a dataset of real debris orbits, which output individual characteristics of around 14,000 orbits and saved each orbit to a file. These generated files, containing the position versus time of each item of debris, were then converted into a real-time graphical simulation of the debris, as shown in the image to the left. The graphical simulation was adjusted to demonstrate the vantage point of a LEO satellite. Screenshots of the simulation from this vantage point were fed into the YOLO model for training. Afterward, the YOLO was able to receive information live from the generated debris visualization.



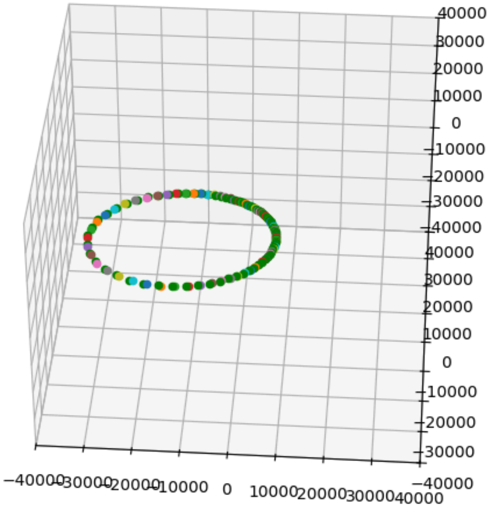
## Solution

### Orbit Generation

The debris orbits were generated from a dataset obtained from Kaggle. The dataset quantified one debris item per line summing to a total of around 14,000 pieces of tracked debris and 14,000 lines of data. To describe each orbit, the dataset gave over 20 Kepler's characteristics along with name, ID, country of origin, and date of creation using a Comma Separated Value(CSV) file. As the information may suggest, the debris information was recorded from real debris in space and does not contain randomly generated numbers.



Kepler characteristics are values that describe an orbit in a way that allows us to derive specific information about its path. For example, a characteristic could include an object's closest and farthest point radius, eccentricity, inclination off the xy plane and ‘w’ rotation away from the zy plane. Using these values, the radius of the debris from earth are given by $R = \frac{A^*(1-e^2)}{1+e^*cos(V)}$ where ‘A’ is the semi major axis, ‘e’ is the eccentricity, and ‘V’ is the true anomaly (angle from $\theta = 0$). A python program input the required values and calculated the radius based on incrementing $\theta$, calculating one full orbit at a time. This provided a basic outline of the orbit’s position over an orbit. It was then rotated off the xy, yz, and zx axis based on the ‘w,’ ‘i,’ and the RA node. The rotation matrix used for adjusting the orbits is given by $R = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix}$. This translates the Kaggle orbits into the 3D path of the orbit.



After the piece of debris’ actual path through space has been found, the time steps need to be calculated. To do this, we used the conservation of momentum, $P = mv$ to find the time between each calculated position. The velocity was calculated by subtracting the potential energy from the total energy and solving for ‘v’. The equation for total energy is given by $E_{total} =- \frac{GMm}{2A}$, the potential energy at the time of measurement is $U =- \frac{GMm}{r}$, and the Kinetic energy is given by $K = \frac{1}{2}mv^2$. Using pythagorean theorem, we calculated the distance between the said measurement and used $\Delta t = \frac{distance}{v}$ to find the time. At the completion of finding each time step, the information was saved to a CSV file for further use. It is saved to the format *time since the start of orbit*, *x*, *y*, *z*. The python program repeated the process for the rest of the debris, reading one more line of the dataset and creating another debris CSV file until every debris file was made.

## Orbit Visualization

The Setup

Each individual orbit is saved to a CSV file, with each line containing information about the x, y, and z coordinates of the debris along with the time. There are 14,372 debris orbits in total, with two extra files for the orbits of the Earth and moon. The Sun orbits around the Earth for the sake of not having to deal with massive floating-point numbers causing precision loss on satellite positions. Instead of a full orbit it changes between 8 approximation positions in a circle around the Earth. The Earth also rotates around its axis on a 24-hour time interval. To keep the sun and earth's rotations synced up with the debris' timescale, they use one of the orbit files to determine the timescale being used, and then take reference to the timestep while using their own logic for positioning. Below is what our simulation currently looks like. The sun is not directly seen by the satellite at its current angle, but it helps simulate the day/night cycle.

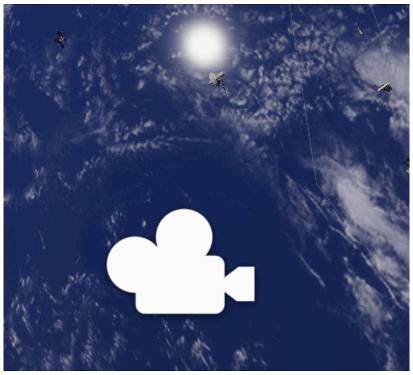
For convenience, the simulation is sped up significantly, and the xyz coordinates of every orbit are decreased by a scale of 256. This scaling down is necessary because with 3D simulations such as Unity, as a number gets larger the amount of floating-point precision a number can hold decreases. By scaling down the number we get more decimal precision, and it is easier to move around the environment during testing. The earth and moon are scaled down in size to match the coordinate shrinking, so the simulation is an overall 1/256 scale model of the Earth-Moon-Sun system.

At runtime, a debris generator object instantiates all 14,372 pieces of debris with one of seven random debris models. Each piece of debris is instantiated at a random angle as well. The camera is attached to a survey satellite in the same orbit zone as the debris, and pointed directly towards the Earth. The debris are scaled up to 100 times their real world sizes to make it easier for the low resolution images we are training the neural network with to actually capture the 3D models. This would not be as much of a problem in a real-world situation because the camera would take higher resolution images that could capture important features from further away.

Optimization

14,372 pieces of debris is a lot to simulate individually. Each piece must update its position in almost every frame and has a fairly dense 3D model attached to it which also uses a significant amount of computing power. On a RTX 3060 graphics card and an intel i9 cpu, the simulation runs at under 10 Frames Per Second(FPS) without optimization. The simulation is intended to run alongside a neural network and must run at or near "real-time" (30-60 FPS) to be viable.

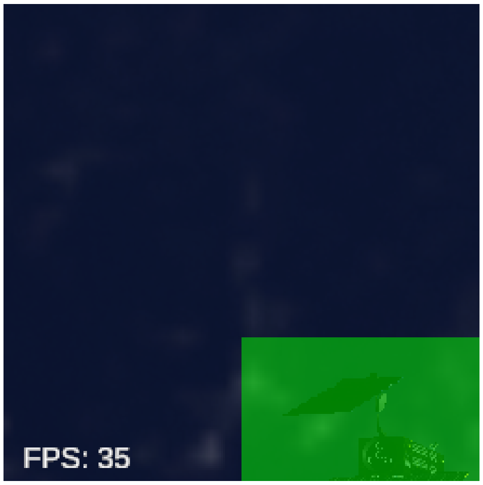
To optimize the simulation without compromising on the amount of debris, everything the camera can't see doesn't need to be fully simulated. We can do this by checking if a debris item is past a certain distance from the camera, and if it is we turn its model off. By running this every time when a piece of debris moves, we create a "mask" around the camera where things are visible, and hide everything else.



This saves some computational power, but not enough to get a "real time" frame rate. The main thing causing FPS to drop is the number of times each debris item changes position. Every piece of debris updated its position every few milliseconds or so. This meant the CPU was always swamped with requests, making each frame take longer to process. To fix this, we again use the principle of not spending resources on debris the camera cannot see. To do this, we again need to know how far the debris is from the camera. First, we have a script on the debris that calculates each time it moves how far away it is from the camera. Then, we calculate an "optimization number"-a number that the debris' delta time and lines skipped will be multiplied by, making its motion choppier (but still following the same path at roughly the same time) and less computationally expensive. After trying a few different methods, we settled on taking 'e' (Euler's number) to the power of the distance minus the radius around the camera that we want the debris moving smoothly in with a minimum value of one and a maximum value of 45, rounded to the nearest integer. Using this function means we get a tight sphere around the camera where everything moves smoothly, but after that point the debris gets optimized quickly (but not so much that the orbit becomes completely unusable). By doing this, the simulation can now run all 14,372 pieces of debris simultaneously with a range of FPS between 30-60. Another thing to note is that the simulation is rendered at a resolution of 256x256, for two reasons: One reason is that it makes the simulation run faster, and the second reason is that it makes the neural network faster to train and run. In a real-world scenario the resolution would need to be much larger than this, so there would be some necessary loss in performance past what we use in this project in order to make the model work better in space.

Dataset Generation

With the simulation complete, the dataset for the object detection model could be generated. This is relatively easy with Unity's built in GUI functions: existing functions can detect which objects are on screen and where they are on screen (this does not use machine learning, as Unity has access to the 3D simulation and can figure all of this out deterministically). We used more GUI functions to draw the rectangles on screen as a sanity check for the boxes before we screenshot each frame of the simulation and write the important information such as class and bounding box of each object to another CSV file (YOLO uses the format center x, center y, width and height for its bounding boxes). Below is an example of one of these boxes and the sanity check:



For the most part this gives relatively good bounding boxes to use in training the neural network. This code runs until each object class has a certain number of screenshots with it in the picture. We generated 100 images per class as this is the recommended minimum for such a model.

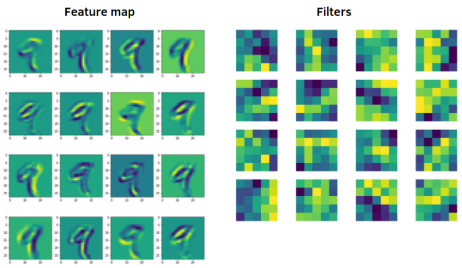
However, the format of the dataset cannot be used with YOLO out of the box: the standard format for YOLO requires a multitude of text files containing object information with one text file per image rather than a CSV file with every image's data in it. We made this transition with a single python script. We also divided each value for the bounding box by the resolution of 256 by 256 since YOLO works off of screen ratio rather than pixels, and changed the definition of the Y value on the image from starting at the bottom, as Unity has it, to starting at the top. After everything was in the proper format, the YOLO object detection model could be trained.

## Object Detection

The Structure of YOLO

YOLO stands for You Only Look Once, and it is currently the most popular neural network for object detection. There are many different versions of YOLO, ranging from the original YOLO to the most recent YOLOv12 model. During this challenge, we focused on YOLOv4 but eventually switched to YOLOv5 after we ran into a critical bug in our v4 code that required us to switch over to v5 just in case we couldn't solve the bug in time.

YOLO's structure consists of three parts: a backbone, a neck, and a head. The backbone is a convolutional neural network for object classification, and is the part that actually determines what objects are in the scene as well as a great deal of where they are. A convolutional neural network classifies objects using convolutional layers, which extract various "features", such as closed circles or even something as complicated as the contours of a face. Below is an example of what these features and feature extractors look like if visualized.

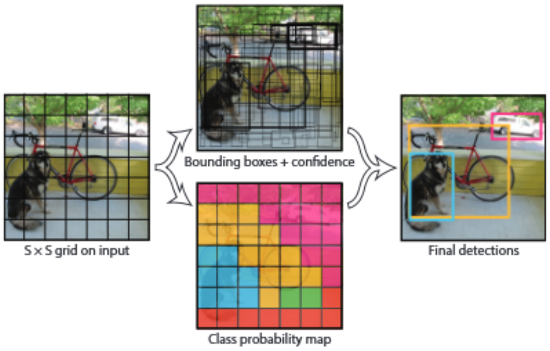


Typically, convolutional neural networks are used to identify a single object in a picture as a standalone neural network, but YOLO modifies the network by essentially splitting the given image into a grid that the backbone classifies individually. This gives the neck and head a starting approximation of where each object is as well as what the objects are. This is the largest part of the neural network, and does most of the heavy lifting.

The neck is relatively small compared to the backbone and head, but serves as an important bridge of information between the two. This section collects the features that were extracted from the image during the backbone section and sends this information to the head. This is commonly done with a Feature Pyramid Network (FPN), which can aggregate features on several layers of the backbone at different scales (convolutional layers shrink the image's dimensions, creating vastly different sizes of features). By feeding accurate data into the head, the head works better and can draw more accurate boxes.

The head is where most of the output of the neural network is put together. The head takes the output from the neck and runs it through a series of "YOLO layers" which generate thousands of boxes. Each box has its own confidence value representing how confident YOLO is that there is an object in the box and class prediction. The class prediction is a list of confidence values where the index of the highest value in the list is the class it believes is in the box. (Object types during computation are represented by numbers.) YOLO outputs a list of three separate tensors with these boxes for low, medium, and high box size based on the different scales the neck inputs to the head. This ensures that regardless of an object's proximity to the screen or relative size, the neural network can still pick up on it.

However, YOLO's output is completely unfiltered and outputs thousands of boxes that need to be sorted through to find the best ones. This is done by the second step of the algorithm, which is separate from the neural network and deterministically culls the low-confidence boxes so that only the most accurate ones remain. This turns thousands of useless boxes into a select few usable boxes that are much closer to the actual object's position and class.

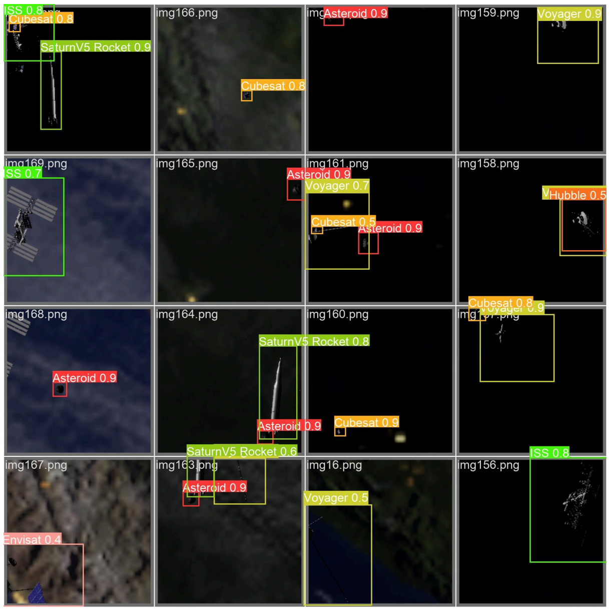


## Attempts at Making a Custom Network

Our original idea was to make a custom implementation of YOLO based on researcher Wong Kin-Yiu's Github repository demonstrating YOLOv4's architecture in Pytorch. We coded this implementation with a dataloader that would have taken our original CSV file dataset format, along with a more specialized structure for our specific needs. The bulk of the code was finished in late February, but when we tried to train the model, the loss of box accuracy plateaued at 0.3 (a very high number) and the output was completely unusable. Unfortunately, attempts to fix this issue didn't look promising, and after a month of bug fixing we decided to switch to a professional implementation of YOLO and continue fixing the problem after we had something working. Having to use a version of YOLO we didn't personally program and customize was disappointing, but through making our own implementation at the start we were able to understand the structure of the tool on a deeper level than if we had started out with a premade model.

## Training YOLOv5 Instead

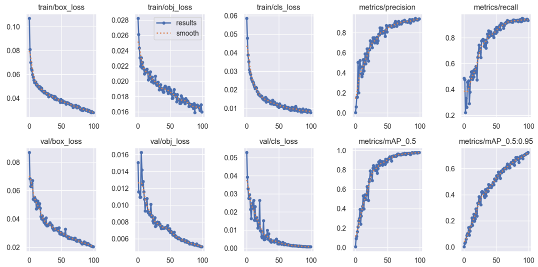
Since we already had a large amount of experience getting our YOLOv4 set up, getting YOLOv5 to train on our dataset was simple. We switched the dataset from the CSV file to a set of txt files so the dataloader could read it, and added a .yaml configuration file that would point towards our dataset and the number and names of our classes. After this, we loaded up a python virtual environment to run 100 epochs of the YOLOv5 nano model, which is the smallest version of YOLO that is typically used. We chose this version because it has the best chance of running on current space technology, while also being accurate enough to pick up on all of the features it needs to pick up.



As seen above in one of the output testing images, the model draws boxes over each piece of debris it detects along with the class it believes the object is. After it was trained, we ran the neural network simultaneously with the graphical simulation and visualized the output much like the above image. This worked extremely well, even when the camera was looking at a darker image like when it was nighttime. To improve accuracy at a marginal increase of computational stress, we could increase the resolution and the number of classes. However, we have not given up on fixing YOLOv4. We will continue work on it after the report is finished and try to get it fixed for the expo.

## Results

YOLOv5 ended its training with a box loss of 0.278, an object loss of 0.016, and a class loss of 0.007. Its ending precision was slightly under 94%. If necessary or if we wanted to add more classes, we could train it for longer to get an even more accurate model. The “nano” model structure didn’t even need two computers to run it in real-time as we originally planned, and it could run easily alongside the real-time graphical simulation and a communication script that fed the simulation’s window into the neural network. Here is the YouTube video that demonstrates the simulation and the detection side by side: https://www.youtube.com/watch?v=W5LQczcIe_4. While there is a slight delay in presentation of the detection boxes and the original simulation due to using python as a visualizer of the boxes, the detection algorithm ran at a much higher framerate than the simulation, running at hundreds of FPS and drawing accurate boxes around debris. Additionally the simulation's debris and angle are randomly generated, so there is little overlap between what the neural network explicitly trained on and the material it was tested on here. The model's high performance in this case indicates that the neural network did not overfit while training. The neural network was also able to perform well during the night, demonstrating its ability to accurately discern class and box location based on less clear information. If required we could easily up the image resolution to something more accurate for a satellite, as well as improve the model from YOLOv5 Nano size to small or even medium with a manageable tradeoff of speed in return for more accuracy. Below is the graph of our loss metrics (left) and accuracy (right) during training. For a larger version of this graph along with other important graphs, see the end of this document in the Graphs and Tables section.



## Conclusion

YOLO has proven to be an accurate, performant way of analyzing and classifying space debris. However, it is also rather delicate and difficult to get working, and there are many edge cases and direct attacks that can decrease its accuracy significantly, as is the case with many neural networks. With our own implementation, programming around these issues and getting it running in the first place was complicated and required us to learn a great deal about neural networks, convolutional neural networks, and object detection algorithms in general. This makes it our most significant achievement in the project, even though we were unable to get YOLOV4 working in the final product.

YOLO as a solution to detecting and classifying objects in space has some downsides. Primarily, it can only classify objects it has been explicitly trained on. We used only seven classes in our project, but there are hundreds, if not thousands of types of debris currently in orbit and it would be impossible to catalogue them all and train a network on it. Classification needs more training the more classes you train on, and even though the relatively primitive YOLOv2 could hypothetically guess over 9000 classes relatively accurately (for its time), this creates a larger neural network that requires more memory and power. However, one advantage indicated by research is that a larger neural network would be more "robust" when exposed to radiation than a smaller one, as there would be more redundancies and pathways that the model could still perform relatively well without.

Additionally, YOLO cannot explicitly find the distance to or relative position of the debris in 3D space, just where it is on the screen (Although this can be paired with other algorithms to figure this part out). Finally, we once again run into the issue of performance: current space technology is at least ten years behind modern graphics cards and CPUs, and even in 2017 the most efficient object detection algorithms ran at 6 FPS and were considered extremely performant at the time, as seen in Tsung-Yei Lin's paper on feature pyramid networks. This is due to radiation being hard on modern, fragile graphics cards and frying them easily, along with the amount of power neural networks use. This is a relatively small problem, as even now there is research going into getting neural networks such as this into space and even some commercially available satellite parts that YOLO could reasonably run on without too much trouble.

These weak points are a significant concern, but using other technologies in combination with object detection can cover these weak points and result in a much more accurate debris detection system than we currently have. In particular, a supporting onboard radar system would be able to tell where exactly the debris is (one of the problems with YOLO) as well as provide auxiliary information about objects such as their material or mass. This last point is important because it would let YOLO train on fewer, more general classes of objects (such as satellite, rocket stage, etc.) but still have enough context clues for a deterministic algorithm to piece together the output of both the radar and YOLO for a much more extensive list of detections than either one could give alone.

## **Acknowledgements**

Special thanks to Francisco Viramontes, one of our teammate's internship mentor who has provided helpful insight on both the inner workings of YOLO as well as advice on how to frame our problem to be both accurate to space technology's current ability to handle machine learning algorithms, as well as giving us ideas as to how machine learning could be used effectively in space.

Regina Hunter also helped in proofreading this paper in the final stages, and helped us catch points where we needed to elaborate for clarity of concepts.

We would also like to thank Jeremy Jensen, our team Sponsor for the last three years of competition. Without him we would not have been able to compete, and it would have been much more difficult to communicate with SCC without him.

Finally, Hamilton Link provided us with crucial input on troubleshooting the YOLOv4 network along with Mario Serna who helped with our orbital simulations and calculation.

## Works Cited

Bochkovskiy, Alexey, et al. “YOLOv4: Optimal Speed and Accuracy of Object Detection.” Institute of Information Science, 23 April 2020, YOLOv4: Optimal Speed and Accuracy of Object Detection. Accessed 15 January 2025.

David, Leonard. “Space Junk Removal Is Not Going Smoothly.” 14 April 2021, https://www.scientificamerican.com/article/space-junk-removal-is-not-going-smoothly/. Accessed 20 March 2025.

“Earth Textures | 1.0.0.” Github, 29 May 2024, https://github.com/RSS-Reborn/RSS-Earth/releases/tag/V1.0.0. Accessed 6 February 2025.

“Envisat.” Sketchfab, 2016, https://sketchfab.com/3d-models/envisat-65b0ec49681a44f68dfc8bd4efe95839. Accessed 30 January 2025.

“Hubble space telescope.” Sketchfab, https://sketchfab.com/3d-models/hubble-space-telescope-e22236fab9634c959c0525c7ab9c83d7. Accessed 20 January 2025.

“International Space Station 3D Model.” NASA, 22 April 2019, https://science.nasa.gov/resource/international-space-station-3d-model/. Accessed 17 February 2025.

Keeter, William. “NASA 3D Resources.” NASA, https://nasa3d.arc.nasa.gov/detail/cubesat-1RU%5C. Accessed 20 February 2025.

KHANDEKA, KANDHAL. “SATELLITES AND DEBRIS IN EARTH'S ORBIT.” Kaggle, 2022, https://www.kaggle.com/datasets/kandhalkhandeka/satellites-and-debris-in-earths-orbit. Accessed 4 November 2024.

Lin, Tsung-Yi, et al. “Feature Pyramid Networks for Object Detection.” Cornell University and Cornell Tech, 19 April 2017, https://arxiv.org/pdf/1612.03144. Accessed 1 February 2025.

Mahendrakar, Trupti, et al. “SpaceYOLO: A Human-Inspired Model for Real-time, On-board Spacecraft Feature Detection.” Florida Institute of Technology, https://arxiv.org/pdf/2302.00824. Accessed 29 October 2024.

“Orbit Orientation.” AI Solution, https://ai-solutions.com/_freeflyeruniversityguide/orbit_orientation.htm. Accessed 24 December 2024.

“Planetary Physics: Kepler's Laws of Planetary Motion.” *Orbits and Kepler’s Laws*, NASA, 26 June 2008, https://science.nasa.gov/resource/orbits-and-keplers-laws/. Accessed 30 October 2024.

Redmon, Joseph, et al. “You Only Look Once: Unified, Real-Time Object Detection.” https://arxiv.org/pdf/1506.02640. Accessed January 2025.

“Space Debris.” *HQ Library Navigation*, NASA, https://www.nasa.gov/headquarters/library/find/bibliographies/space-debris/. Accessed 26 October 2024.

"Space Debris Velocities." *RULES OF THUMB AND DATA FOR SPACE DEBRIS STUDIES*, NASA, https://science.nasa.gov/resource/international-space-station-3d-model/. Accessed 25 October 2024.

"Space Rocket Saturn V 3D Model." Free3D, 23 April 2020, https://free3d.com/3d-model/space-rocket-saturn-v-360313.html. Accessed 23 January 2025.

"Space Rocks." CGtrader, https://www.cgtrader.com/free-3d-models/space/other/space-rocks-9351486c-0cd0-46e7-ab36-62c8a621820d. Accessed February 14 2025.

"Voyager." NASA, 31 March 2025, https://nasa3d.arc.nasa.gov/detail/jpl-vtad-voyager. Accessed 11 February 2025.

Yiu, Wong Kin. "PyTorch implementation of YOLOv4." Github, https://github.com/WongKinYiu/PyTorch_YOLOv4. Accessed 27 January 2025.

"YOLO9000: Better, Faster, Stronger." Allen Institute for AI, 25 December 2016, https://arxiv.org/pdf/1612.08242. Accessed 10 January 2025.

"YOLO, Ultralytics." Github, 22 November 2022, https://github.com/ultralytics/yolov5. Accessed 2 January 2025.

## Links To Products

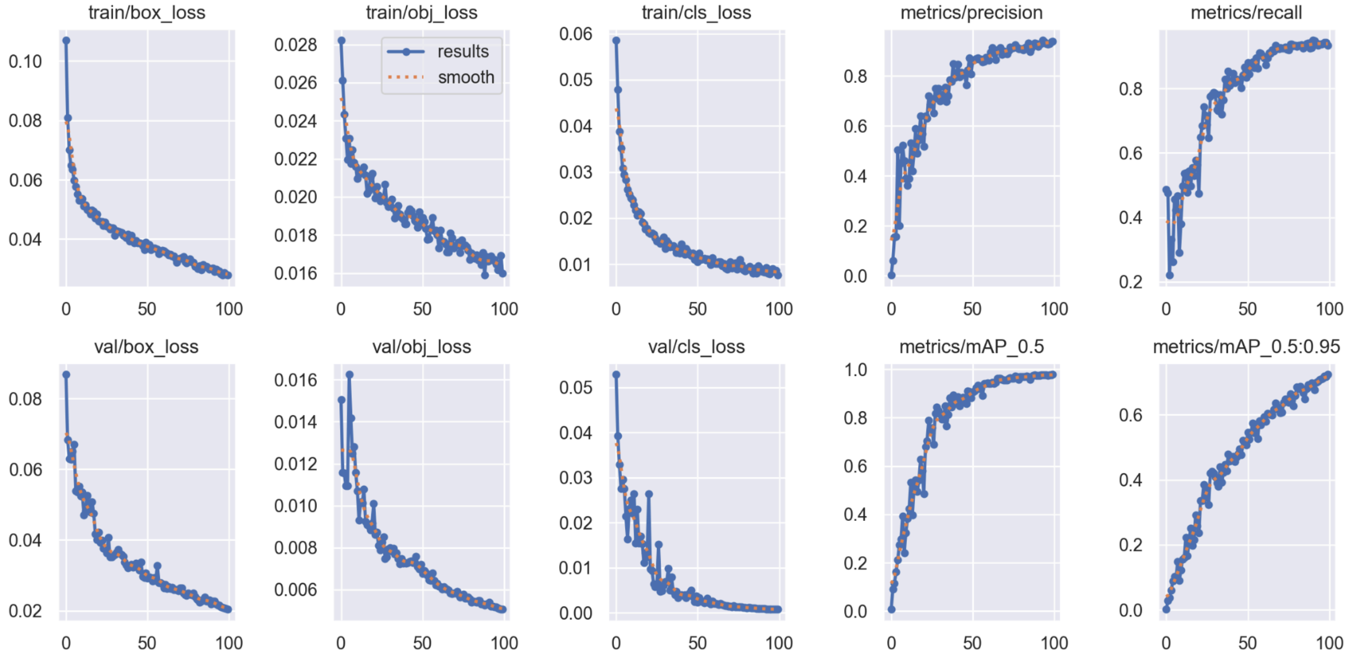
https://github.com/HadwynLink/SCC-2024-2025

(results table can be found in Yolov5/Runs/results.csv)

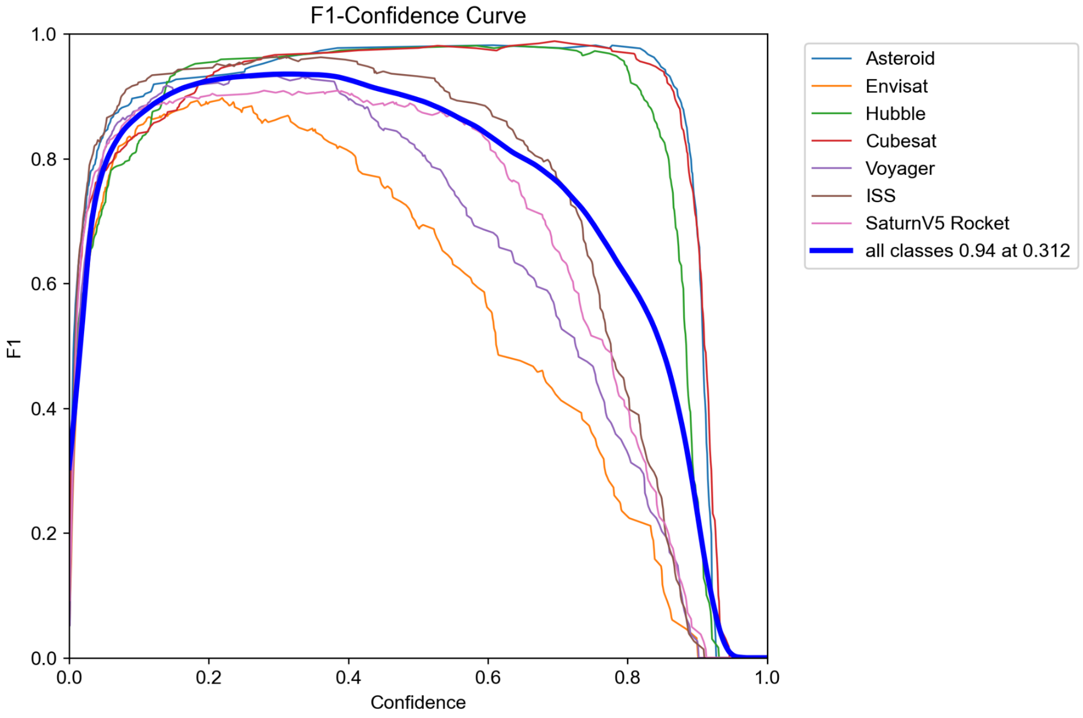
https://www.youtube.com/watch?v=W5LQczcIe_4

# Graphs and Tables

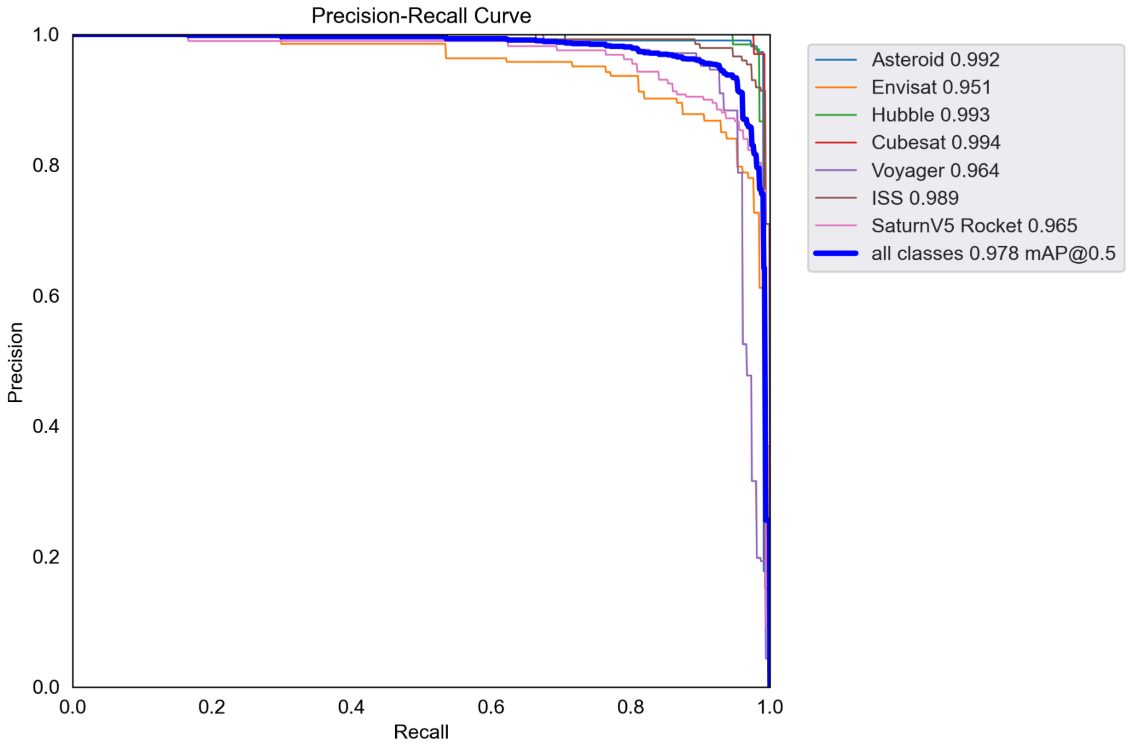
## Graph of loss and precision during training



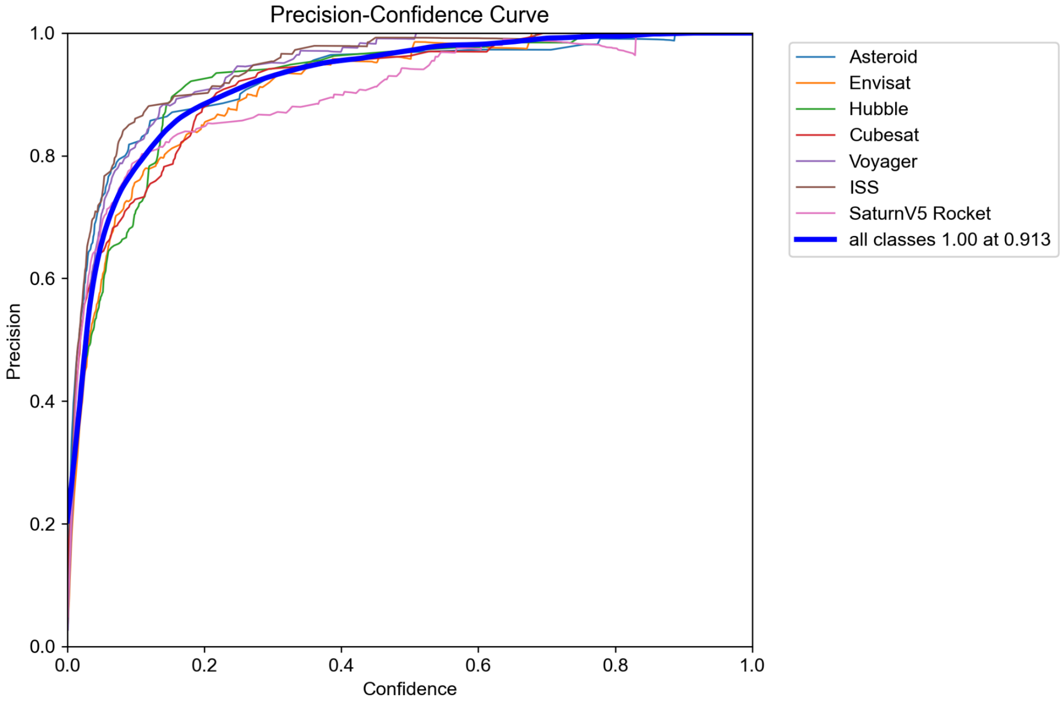
## Confidence Curve



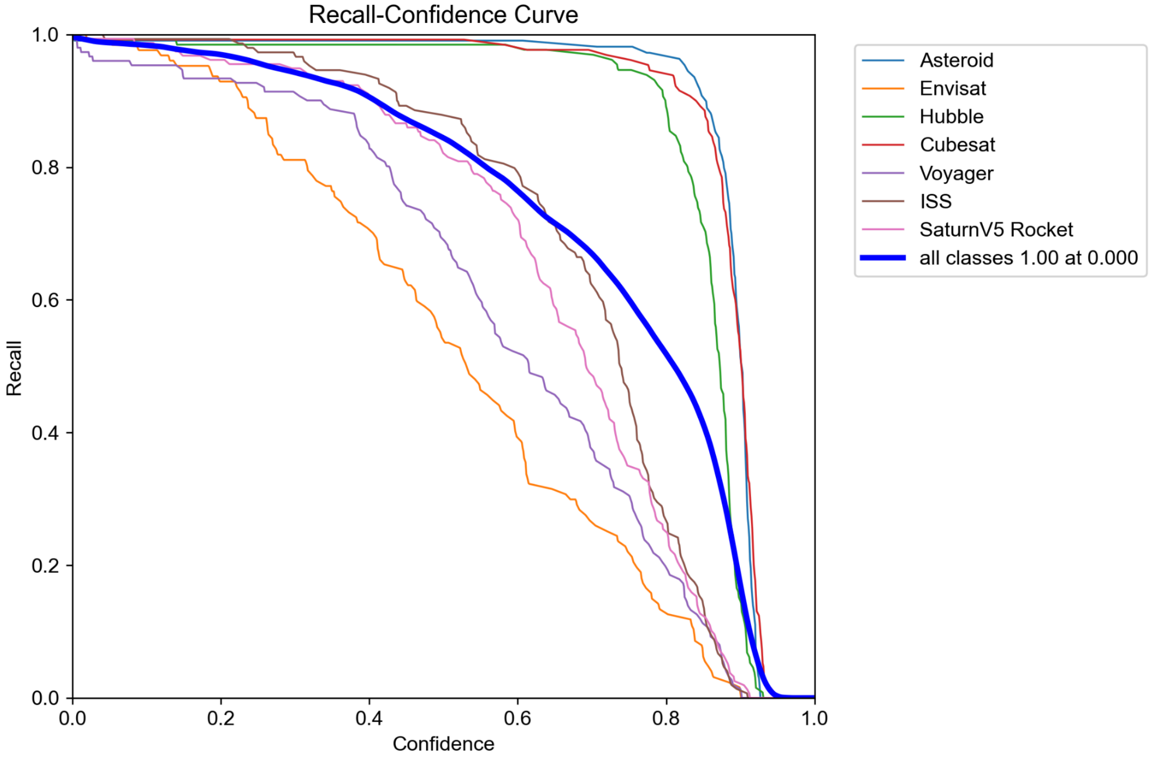
## Precision-Recall Curve



## Precision-Confidence Curve



## Recall-Confidence Curve


Recall-Confidence Curve
Recall
Confidence
Asteroid
Envisat
Hubble
Cubesat
Voyager
ISS
SaturnV5 Rocket
all classes 1.00 at 0.000