ECE4300 Group 2 Report: YOLOv5 Image Detection

Nicholas Garcia, Michael Guillermo, Danny Manzo, Arthur Wang
Electrical and Computer Engineering Department at
Cal Poly Pomona
Pomona, United States
nggarcia1@cpp.edu, mrguillermo@cpp.edu, manzo@cpp.edu, arthurwang@cpp.edu

***Abstract*—** Object detection is a technique in which a software system can recognize and locate an object in a given medium. There are several models that can identify and trace objects, but the You Only Look Once (YOLO) model has proven to be an exemplary solution to object detection. With objectives of speed, precision, and user-friendliness, YOLO can process images in real-time. Experimenting with different computer architectures and custom datasets, group two conducted a comprehensive evaluation of the YOLOv5 model to ascertain its efficiency and performance. In this model's evaluation, we test, analyze, and train datasets on distinct minerals to gain a better understanding of the YOLO object detection algorithm and how its performance and precision are affected by differing CPUs and GPUs. The approach to evaluating the performance of the YOLOv5 model was to train and detect minerals, and our goal was to study the performance of different versions of YOLOv5 across different computer architectures.

Keywords: YOLO, YOLOv5, CPU, GPU, object detection, computer architecture, YAMl, mAP50, mAP50-95, epoch, image, batch, dataset

I. Introduction to the YOLO Model

A. What is YOLO?

The You Only Look Once (YOLO) architecture is one of the most popular models for object detection. Created by Joseph Redmon, the software was named after its ability to pass an input image only once to make predictions about the presence and location of objects in that image. YOLO software utilizes one of the best neural network architectures to produce an algorithm with high accuracy and fast processing speed to allow for real-time identification and positioning of objects in any given image or video. With some form of media input, YOLO implements bounded boxes and determines the size and center position of each box. It also predicts the probability at which the class in question is what the software claims it to be.

B. How it Works

YOLO works by these steps. First, dividing an image into grids makes it easier to detect an object in one of the cells as opposed to throughout the whole image. It then, creates a vector using the following: probability of the objects class, coordinates of the center of the bounding box relative to the cell, the width and height of the bounding box relative to the whole image, and the class that represents the bounding box. As each cell of the grid is given a representative vector, we can use that dataset to create a convolution network. Applying a 24-layer convolution neural network followed by two fully connected layers, is what gives YOLO its quick and precise abilities in object detection. Each convolution layer narrows down the grid dimensions allowing for more accurate detection and location of an object in a cell. Each grid cell is responsible for detecting an object if it falls within its boundaries as well as the confidence at which the object is what the algorithm has predicted it to be.

C. Notable Features Over the Years

YOLO (2015) – can only detect a maximum of two objects of the same class in a grid cell. Learns from coarse object features. Average precision of 63.4%. Utilizes Darknet-19 as backbone.
YOLOv2 (2016) – capable of detecting 9000 categories with an average precision of 78.6%. 19 convolutional layers and 5-max pooling layers.
YOLOv3 (2018) – uses logistic regression to assign object scores to anchor boxes. Achieved a mean average precision of 60.6% at 20 frames per second.
YOLOv4 (2020) – includes mosaic augmentation and smoother convolutional neural networks. Introduction of bag-of-freebies and bag-of-specials to maintain inference time and improve accuracy.
YOLOv5 (2020) – developed using PyTorch instead of Darknet. Average precision of 50.7% with an image size of 640 pixels at 200 frames per second.
YOLOv6 (2022) – adopted an anchor-free detector. Achieved an average precision of 70% at around 50 frames per second on NVIDIA Tesla T4.
YOLOv7 (2022) – boosts three main functions: E-ELAN, model scaling, and bag-of-freebies.
YOLOv8 (2023) – faster Non-maximum Suppression process. Achieved an average precision of 53.9% with an image size of 640 pixels.

D. YOLOv5

The YOLO model has had several iterations since its inception in 2015 but for the sake of our evaluation, we chose to focus on one of the models that were released in June of 2020 by Glenn Jocher from Ultralytics: YOLOv5. Building on the improvements made just two months before on YOLOv4, this newer edition comes in four different versions. The versions range from small to extra-large and each take a varying amount of time to train and implement. The two main processes in YOLOv5 in regard to training is data augmentation and loss calculations. When an input is passed through a loader, it makes three augmentations to the data: scaling, mosaic augmentation, and color space adjustments. A combination of all these components have led to an impressive average precision of 50.7% with an image size of 640 pixels and 55.8% with images of 1536 pixels.

II. Methodology

Our methodology of testing YOLOv5 is by a set of parameters meant to assess its behavior. The first parameter is the type of object that we're testing. We're testing minerals found on Earth, because there's a wide variety of minerals found on Earth such as salt, topaz, and gypsum. For our purposes, we're testing a limited set of minerals including pyrite, quartz, muscovite, malachite, chrysocolla, bornite, and biotite.

The second parameter is our computers. We want to see if certain components of a computer can affect some of the YOLOv5's overall performance variables such as time to complete, its mAP50 (the mean average precision with a interaction over union (IoU) threshold of 0.50; used to measure "easy" detections), and its mAP50-95 (the mean average precision with an interaction between a IoU value of 0.50 and 0.95) [1].

Two other parameters that will be tested are precision and recall. In this case, precision is the proportion of the "positive class predictions that were actually correct" [2]. Recall refers to the "proportion of actual positive class samples that were identified by the model".[2] In both cases, these two parameters will identify areas on which the dataset needs to improve upon.

The last parameter is our CPUs. We believe that each CPU has its own behavior/architecture that'll respond to demanding tasks differently. Therefore, we'll evaluate our CPU's behavior when training with YOLOv5.

III. Implementation

Our first approach was to train and assess YOLOv5s (smaller model) on Google Colab. Through Google Colab and with the support Python, we were able to familiarize ourselves with the software and it gave us a general idea on how to train certain images with YOLOv5. The Google Colab application provides its users with a GPU (graphics processing unit) to implement when testing out and training the YOLOv5 model. In each of our experiences, we were given the Tesla T4 GPU. Since our efforts involving Google Colab were to gain a preliminary understanding of how to quickly incorporate custom trained datasets, it proved to be widely beneficial. With the use of Roboflow, we were able to download and train our datasets without having to add our own labels and images.

A. General Steps to Implement YOLOv5

This leads to the next phase of our project which is to implement YOLOv5 onto our computers. We quickly realize that implementing YOLOv5 requires a different series of steps necessary to install, implement and run the image detection system.
The following section features the steps needed to implement YOLOv5 on our computers:

Step 1: Meet or exceed the requirements needed to install YOLOv5 which includes Python 3.8 or higher and PyTorch 1.8 or higher.

```
git clone https://github.com/ultralytics/yolov5  # clone
cd yolov5
pip install -r requirements.txt  # install
```

Fig 1: The command lines used to install YOLOv5[3]

Step 2: Clone the official YOLOv5 package via GitHub.
Step 3: Go to Roboflow and install and unzip the package that contains our desired image sets.
Step 4: Select and train a YOLOv5 model.

It's worth noting that by default, the YOLOv5 will mainly use the CPU without any changes to the settings. In order to enable YOLOv5 to use our GPU, we follow a recommended procedure that will allow YOLOv5 to use our GPUs. This can be explained with three general steps.

Step 1. Uninstall Torch, TorchAudio, and TorchVision via command line.

Step 2: Go to PyTorch.com and specify what parameters do you need.

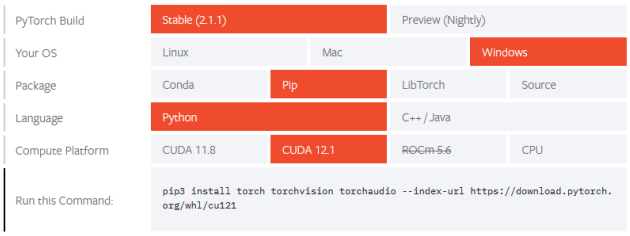

Fig 2: A image showing various options when installing PyTorch and its components.

Step 3: Copy and paste the given command into the operating system's command line.

Also, these three general steps represent our approach to testing YOLOv5, because two of our group's computers have NVIDIA discrete GPUs which includes the CUDA compute platform.

B. How to run YOLOv5 with a Custom Dataset

The next phase of our project is to allow YOLOv5 to run with our custom dataset. Generally, to create a dataset, we need to collect images, create labels for each image, and convert our dataset into a YAML file [3]. This process can be very tedious and time-consuming. Thus, we decided to use a premade dataset to save time and effort on finding the most optimal mineral images. We download the premade dataset onto a zip file and extract it to a file location.

The next step is to figure out how to allow YOLOv5 to use the downloaded, premade dataset instead of its default. We solve this by editing the dataset's data YAML file to have a given path.

Fig 3: Our premade dataset's default data YAML file. By default, it has no written path.

Fig 4: Our premade dataset's data YAML file written with a path.

Finally, we need to change the command that's used to run YOLOv5. By default, the command used to run YOLOv5 is typed like in Fig 4.[3]

```
python train.py --img 640 --epochs 3 --data coco128.yaml --weights yolov5s.pt #end of command
```

Fig 5: Given command used to train a YOLOv5 model based on its default YAML file.[3]

Since we want to use our premade dataset to train YOLOv5, we edited the data constraint to be located on the same file registry as our premade dataset.

With these changes, we were able to train YOLOv5 with our premade dataset instead of its default dataset.

C. Controls and Testbenches Used

For this project, we will have a few controls that we want to have in an attempt to keep the results as identical as possible.

Control Parameters	Number/Name	Definition
Epoch	10	Total number of training samples in a batch.
Img	640	Image Size
Batch	16	Number of Images per batch.
YOLO Model	YOLOv5s.pt	Size of the YOLOv5 model. (Second smallest)

Fig 6: List of Control Parameters Used in this project.

Group Member	CPU	Memory	GPU1	**GPU2**
Michael	Intel Core i7-1260P	16GB (DDR4)	Intel Iris Xe Graphics	N/A
Nicholas	Intel Core i5-8265U	8GB (DDR4)	Intel UHD Graphics 620	N/A
Danny	Intel Core i7-8750H	32GB (DDR4, 2667Mhz)	Intel UHD Graphics 630	NVIDIA GeForce GTX 1050Ti (4GB)
Arthur	AMD Ryzen 9 5900HS	16GB (DDR4, 3200Mhz)	AMD Ryzen Graphics	NVIDIA GeForce RTX 3060 (6GB)

Fig 7: Our Laptop's Basic Specifications

For this experiment, we'll feature 4 computers. Two of the computers will only train with their CPUs, while the other two will have their GPUs (in this case, **GPU2**) included in the YOLOv5 training.

IV. Results

Using our computer hardware, we attempt to benchmark YOLOv5 in terms of a few factors as mentioned in II: Methodology.

1. mAP50
2. mAP50-95
3. Precision and Recall
4. CPU behavior/performance results (based on average)

For each result, we use Comet or Excel to tabulate our numbers into a graph.

A. mAP50 Results

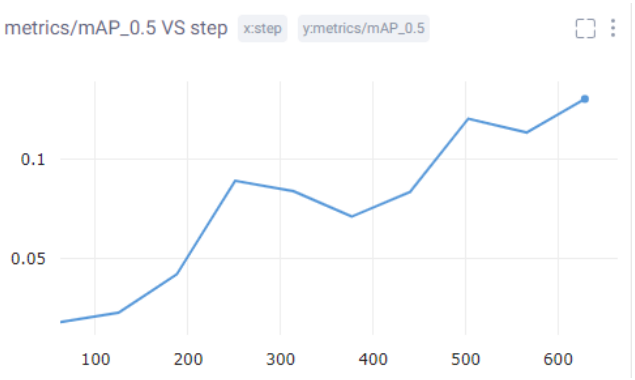

Fig 8: mAP50 graph results with Michael's hardware (CPU ONLY)

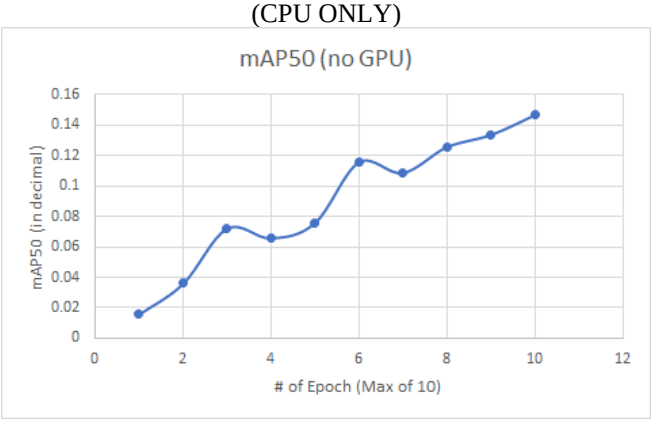

Fig 9: mAP50 graph results with Arthur's hardware (CPU ONLY)

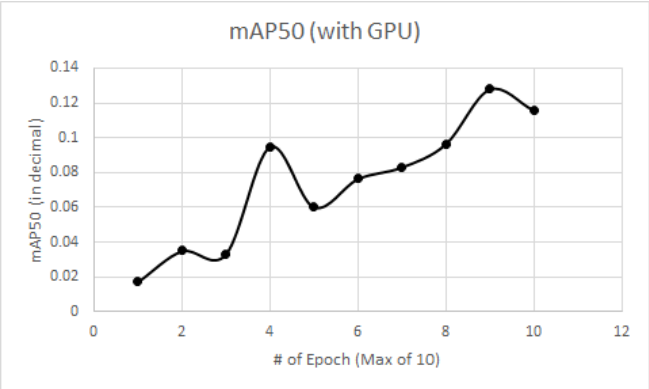

Fig 10: mAP50 graph results with Arthur's hardware. (CPU + GPU2)*

* →IMG=480,Batch=12.

Explained in V: Challenges

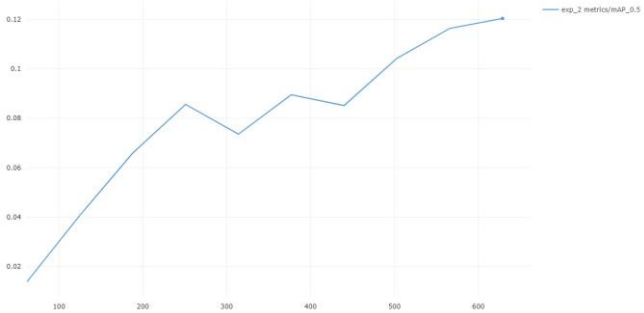

Fig 11: mAP50 graph results with Danny's hardware. (CPU + GPU2)

With a epoch of 10, we can clearly see that for all four graphs, there's a consistent increase overall on the last epoch versus the first epoch.

B. mAP50-95 Results

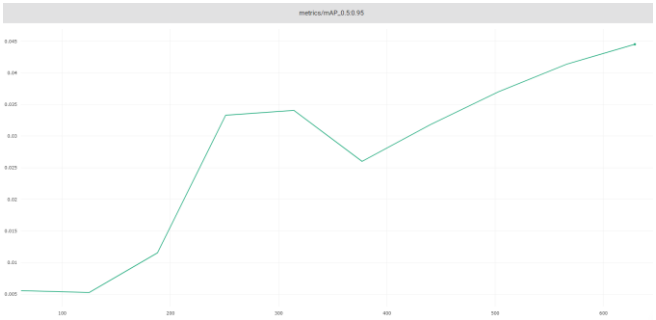

Fig 12: mAP50-95 graph results with Michael's hardware (CPU-ONLY)

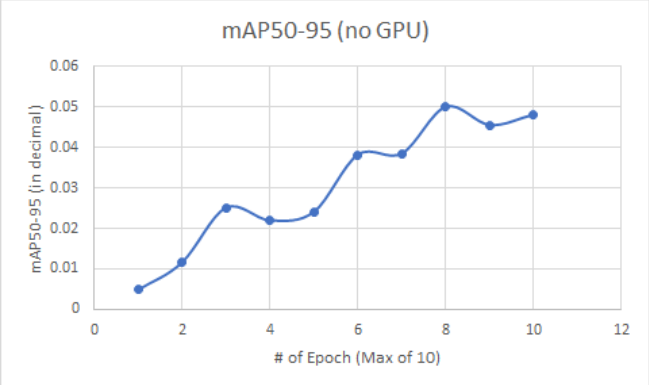

Fig 13: mAP50-95 graph results with Arthur's hardware (CPU-ONLY)

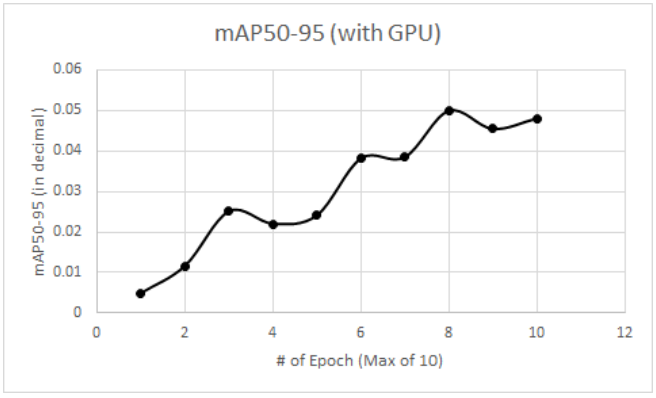

Fig 14: mAP50-95 graph results with Arthur's hardware (CPU + GPU2)*

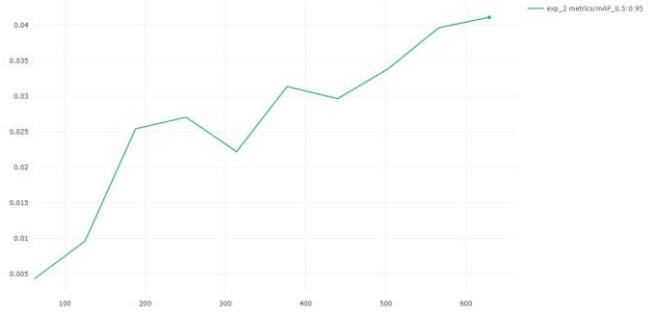

Fig 15: mAP50-95 graph results with Danny's hardware (CPU + GPU2)

C. Precision and Recall Table

P → Precision

R → Recall

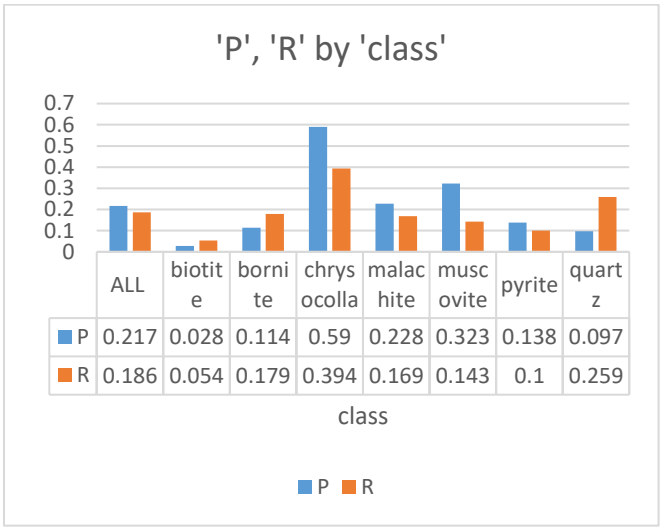

Fig 16: "P" and "R" Table with Michael's hardware. (CPU-ONLY)

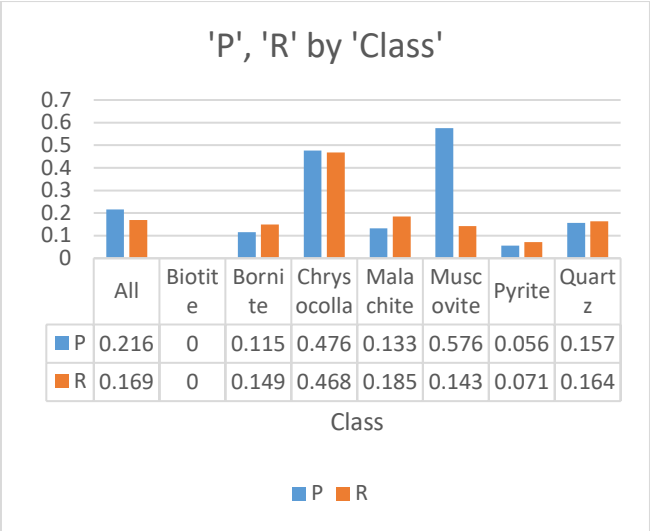

Fig 17: "P" and "R" Table with Arthur's hardware. (CPU-ONLY)

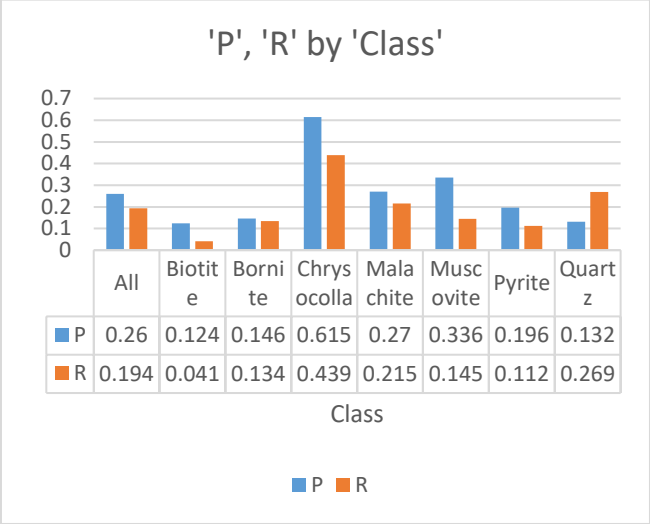

Fig 18: "P" and "R" Table with Arthur's hardware*. (CPU and GPU2)

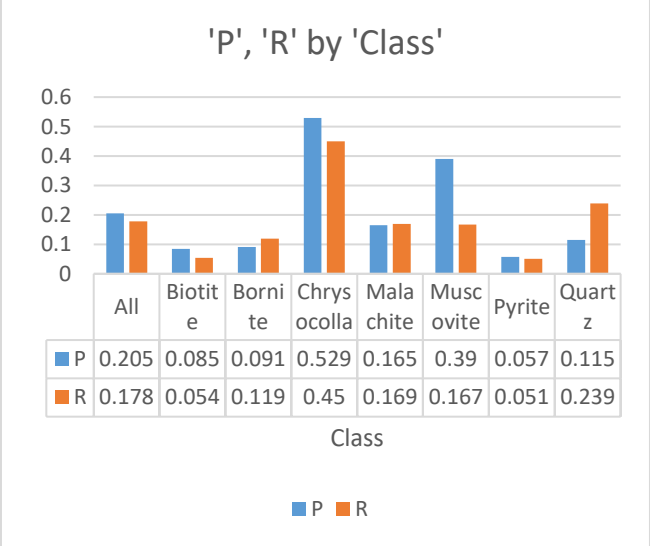

Fig 19: "P" and "R" Table with Danny's hardware. (CPU and GPU2)

Throughout these four graphs, both the precision approximately between 0.178 and 0.26 for datasets. This is a relatively low value which indicates that our datasets don't have enough images, and/or we didn't run enough epochs.

D. CPU Behavioral Charts

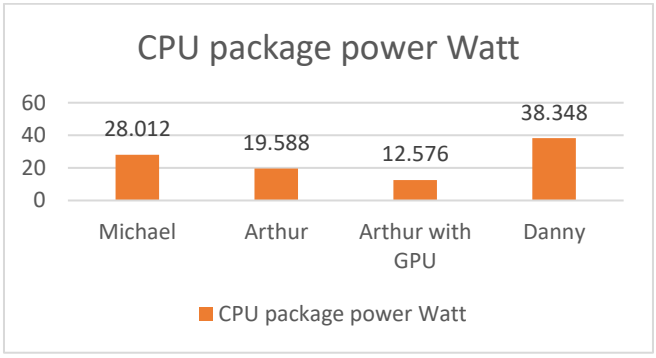

Fig 20: CPU Package Power (in Watts)

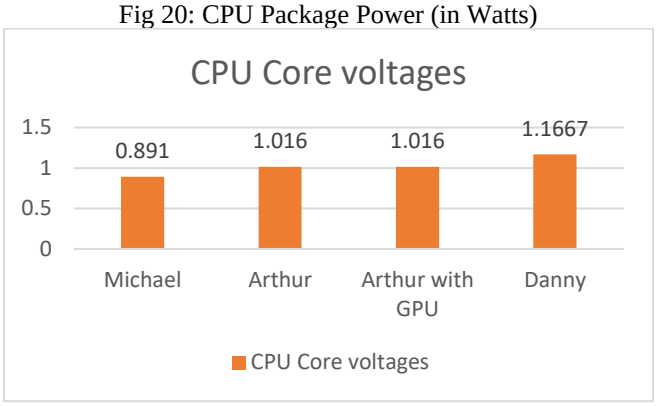

Fig 21: CPU Core Voltages (in Volts)

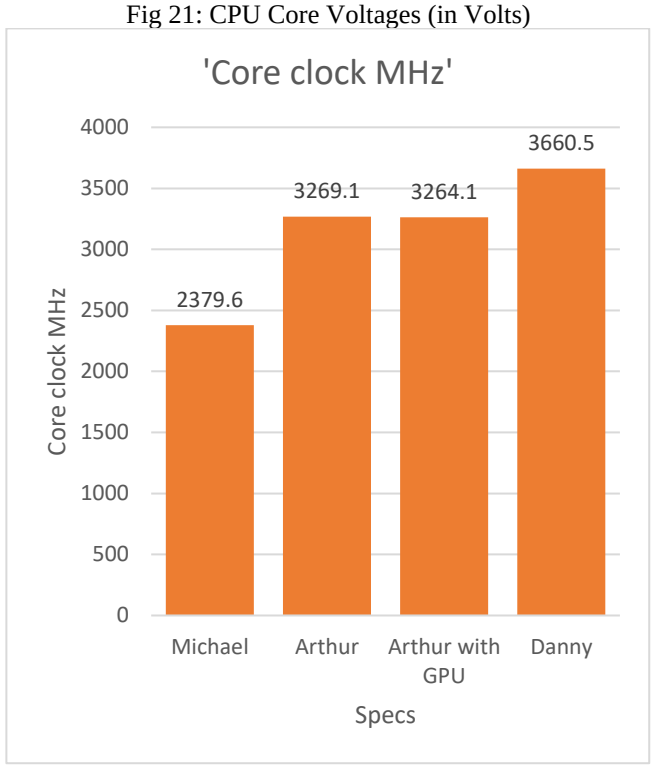

Fig 22: CPU's Core Clock MHz

These results were based on averages during each run of YOLOv5. Interestingly, Arthur's laptop CPU was the only one that was keeping at its base clock of approximately 3.3GHz [6]. Meanwhile, both Michael and Danny's laptops had their CPUs clocked higher than their stated base clocks at 1.6GHz and 2.2GHz [4][5]. This indicates that Michael

and Danny's CPU's ability to briefly clock higher than their base clock speeds were working as advertised. Meanwhile, while Arthur's CPU can also clock higher than its base clock, it wasn't demonstrated on those results [6]. This can mean a multitude of factors such as, but not limited to, his laptop configuration settings, to his CPU's behavior under certain conditions.

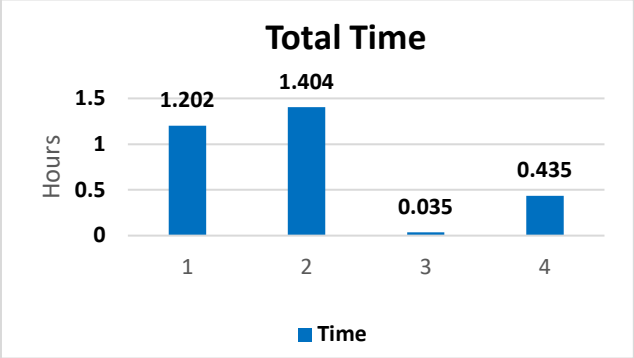

Fig 23: Time to complete YOLOv5 training.

1: Michael's hardware (CPU Only)
2: Arthur's hardware (CPU Only)
3. Arthur's hardware (CPU + GPU2)
4. Danny's hardware (CPU + GPU2)

When using only the CPU to train the YOLOv5 model, we can see that Arthur's CPU was slower to train our dataset by 16.8% (1.404/1.202) compared to Michael's CPU. Based on Fig 22, we can indicate that it's possible that one reason that Arthur's CPU was slower was that it cannot boost its base clock of 3.3Ghz.

Another observation was that adding a discrete GPU will greatly reduce the time to train a YOLOv5 model given our control parameters. For example, when Arthur's laptop included his GPU2 to train YOLOv5, we can see that it was approximately 40 times faster than the CPU alone. We also saw that Danny's hardware also outpaced Michael and Arthur's hardware with its GPU2 by about 2.76 and 3.23 times their values.

V. Challenges

Some challenges that transpired throughout this process mainly originate from trying to implement YOLOv5 on our individual laptops as well as managing CPU temperature to prevent a loss in performance during the testing phase. Installing the dependencies for YOLOv5 proved to cause some issues as some packages would not be properly installed, requiring some troubleshooting to find the missing packages and install them manually. Other issues occurred in the YOLOv5 repository, where errors occurred when downloading the repository directly compared to cloning the GitHub repository.
As for managing CPU performance, managing the temperature of the CPU was necessary to ensure maximum performance when training the dataset. However, with different CPU specs and the cooling systems of each individual's laptops differing, some CPUs ran hotter than others. This resulted in thermal throttling in some cases, causing the CPU performance to degrade to prevent component damage during training. As a result, some trial results would be discarded due to inaccurate results. This issue was rectified by improving air circulation outside of the laptop and clearing the vents to provide maximum air circulation and air cooling.

Lastly, there were some hardware issues that were countered. For example, when Arthur attempted to train YOLOv5 with his GPU2 with the given parameters as seen in *III: Implementation -> C: Controls and Testbenches Used*, he encountered a consistent error message that prevented him from training the YOLOv5 model with the given control parameters.

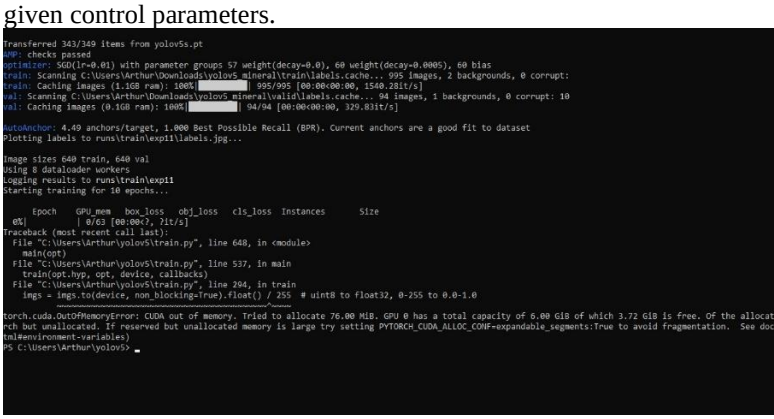

Fig 24: Screenshot of Arthur's computer depicting the error message: torch.cuda.OutOfMemoryError.

Based on the error message, it's likely that either Arthur's GPU2 doesn't have enough video memory or Arthur's hardware doesn't have enough RAM to support the given control parameters. In order to solve this problem, he has reduced the image size (IMG) to 480 and batch size to 12 to keep both his video memory from his GPU2 and his RAM under control. This indicates that there's a bottleneck. This can either mean that the performance of Arthur's GPU2 is held back by an insufficient amount of video memory or his hardware doesn't have enough RAM to support the training, or possibly both.

VI. Moving Forward

In conclusion, the YOLOv5 model has proven to be an efficient and powerful tool for object detection. In our case of detecting and evaluating different minerals on a single custom dataset, it has demonstrated good speed and accuracy throughout different computer architectures. Although, we trained only on 10 epochs and on a limited dataset, the precision was suitable enough for the model to be able to detect and classify objects in a real-time scenario. YOLOv5 being an older model compared to newer models like YOLOv7 or YOLOv8, YOLOv5 is still susceptible. However, to improve precision and achieve optimal results, one must increase the dataset and increase the training parameters such as the epoch number to work through the entire dataset. Our YOLOv5 project stands as a testament to the ongoing improvement of object detection. As the YOLO model continues to grow and improve on new challenges, its use in computer vision will be prosperous in the future.

VII. REFERENCES

[1] "Yolo Performance Metrics," YOLO Performance Metrics - Ultralytics YOLOv8 Docs, https://docs.ultralytics.com/guides/yolo-performance-metrics/#how-to-calculate-metrics-for-yolov8-model (accessed Nov. 28, 2023).

[2] R. Kundu , "Precision vs. Recall: Differences, Use Cases & Evaluation," V7, https://www.v7labs.com/blog/precision-vs-recall-guide (accessed Nov. 28, 2023).

[3] "Train Custom Data," Ultralytics YOLOv8 Docs, https://docs.ultralytics.com/yolov5/tutorials/train_custom_data/#12-create-labels (accessed Nov. 28, 2023).

[4] "Intel® CoreTM I7-1260P processor (18m Cache, up to 4.70 GHz) - product specifications," Intel, https://www.intel.com/content/www/us/en/products/sku/226254/intel-core-i71260p-processor-18m-cache-up-to-4-70-ghz/specifications.html (accessed Nov. 29, 2023).

[5] "Intel® CoreTM I7-8750H processor (9m cache, up to 4.10 GHz) - product specifications," Intel, https://www.intel.com/content/www/us/en/products/sku/134906/intel-core-i78750h-processor-9m-cache-up-to-4-10-ghz/specifications.html (accessed Nov. 30, 2023).

[6] "AMD RyzenTM 9 5900HS Mobile Processor, " AMD, https://www.amd.com/en/products/apu/amd-ryzen-9-5900hs (accessed Nov. 30, 2023).