

Crater Detection for Autonomous Lunar Approach Using Deep Learning

a project presented to
The Faculty of the Department of Aerospace Engineering
San José State University

in partial fulfillment of the requirements for the degree
Master of Science in Aerospace Engineering

By

Matthew A. Copeland

August 2025

approved by

Dr. Jeanine Hunter
Faculty Advisor



ABSTRACT

Crater Detection for Autonomous Lunar Approach Using Deep Learning

Matthew A. Copeland

In the modern world, AI and machine learning have become an impactful technology that has revolutionized many industries. One such industry is space, where new technology has largely changed the way humans interact and explore other planets. Deep learning, a form of machine learning, is applied to the problem of detecting lunar surface craters for use in an autonomous landing and hazard avoidance system. In recent years, several lunar landers have failed by falling or tipping over largely in part due to the hazardous lunar surface terrain covered in millions of small craters. The objective of this project is to develop a vision-based crater detection framework capable of operating on a lander during descent. A custom dataset was developed using 640x640 patches obtained from images taken by the Lunar Reconnaissance Orbiter's (LRO) onboard Narrow Angle Camera (NAC). Each patch is preprocessed and has data augmentation performed on them to increase diverse lunar landing conditions to better represent real planetary environments. Hand labeling, using the Label Studio software, was done to ensure high label quality and customizability. The pretrained YOLOv11 Nano model from Ultralytics was used to train and evaluate the performance using the custom dataset. There were five versions of the model, each using an increasingly expanding dataset starting with 20 patches and the final dataset having 110 patches. The performance metrics were compared with each model version to show how a refined dataset can impact performance. Inference testing was also done on an unseen image to demonstrate and compare the visual detection performance of each version. A brief comparison was conducted between YOLOv11 Nano and YOLOv11 Small to look at the impact model size can have on detection performance. This project demonstrates a complete use case for deep learning in the context of spacecraft hazard detection and avoidance. This paper shows that lightweight models trained on large, diverse datasets can serve as effective crater detection systems. More work needs to be done in this area but there are strong implications for future autonomous landing and hazard avoidance technologies.

Acknowledgements

I would like to thank my project advisor, Professor Jeanine Hunter, who gave me the courage and curiosity to dive into AI and machine learning. Your guidance and the trust you placed in me to learn throughout this project and see it through to the end is greatly appreciated. I hope I was able to inspire some of your undergrad students to explore AI and learn more about the possibilities of this technology. Attending your classes has always inspired me and continuously given me the drive to learn more about the engineering and the world around me.

I would like to thank my family, friends, and fellow classmates who helped encourage me and pushed me to continue on. I would not be here if it wasn't for your unwavering support which has carried me through every stage of this project. The constant reminder to always pursue my passions, and to continually put in 100% effort in everything I do has continued to help me.

Table of Contents

ABSTRACT.....	iii
Acknowledgements.....	iv
List of Tables.....	vi
List of Figures.....	vii
Chapter 1 - Introduction.....	1
1.1 Motivation.....	1
1.2 Literature Review.....	2
1.3 Project Proposal.....	8
1.4 Methodology.....	8
Chapter 2 - Theoretical Framework.....	9
2.1 Convolutional Neural Network.....	9
2.1.1 Convolutional Layer.....	9
2.1.2 Pooling Layer.....	11
2.1.3 Nonlinear Activation Layer.....	12
2.1.4 Fully Connected Layer.....	13
2.2 Backpropagation.....	15
2.2.1 Loss Function.....	15
2.2.2 Gradient Descent.....	16
2.2.3 Regularization.....	18
2.3 Updating Weights and Biases.....	21
Chapter 3 - System Architecture.....	23
3.1 YOLO Architecture.....	23
3.2 YOLOv11.....	26
3.3 Transfer Learning.....	30
Chapter 4 - Data Preprocessing.....	32
4.1 Data Sources.....	32
4.2 Image Preprocessing.....	34
4.3 Labeling.....	36
4.4 Dataset Creation.....	38
Chapter 5 - Performance Metrics.....	39
5.1 Introduction.....	39
5.2 Confusion Matrix.....	39
5.3 Intersection Over Union (IOU).....	42
5.4 Recall- and Precision-Confidence Curves.....	43
5.5 Average Precision (AP).....	45
5.6 F1 Score.....	46

5.7 Summary.....	47
Chapter 6 - Results.....	48
6.1 Introduction.....	48
6.2 Experimental Setup.....	48
6.2.1 Hardware Configuration.....	48
6.2.2 Software Configuration.....	49
6.2.3 Model Training and Inferencing Configuration.....	50
6.3 Model Version Overview.....	52
6.3.1 Version 1.....	52
6.3.2 Version 2.....	53
6.3.3 Version 3.....	53
6.3.4 Version 4.....	53
6.3.5 Version 5.....	54
6.4 Performance Analysis.....	55
6.4.1 Precision-Confidence.....	56
6.4.2 Recall-Confidence.....	58
6.4.3 F1-Confidence.....	60
6.4.4 Precision-Recall.....	61
6.5 Model Inferencing Testing.....	62
6.6 YOLO Nano vs YOLO Small.....	68
Chapter 7 - Conclusion.....	71
References.....	72
Appendix A - Python Script for Converting IMG to PNG Format: 'IMG2PNG.py'.....	75
Appendix B - Python Script for Creating Patches and Performing Data Augmentation: 'image_augmentation.py'.....	76
Appendix C - Python Main Script for Training and Inference Testing: 'main.py'.....	77
Appendix D - Python YAML Configuration File: 'config.yaml'.....	80
Appendix E - Python YAML File Showing General YOLO11 Object Detection Model Architecture from Ultralytics: 'yolo11.yaml'.....	81

List of Tables

Table 3.1 - YOLO model evolution [11].....	24
Table 3.2 - YOLO models from v1 to v8 [11].....	24
Table 3.3 - Performance metrics between YOLOv8 and v11 [23].....	26
Table 6.1 - All model versions with changes to dataset.....	52

List of Figures

Figure 1.1 - AI overview with branching subfields [7].....	2
Figure 1.2 - Example of a generic CNN architecture [8].....	3
Figure 1.3 - Architectural difference between one-stage and two-stage detectors [9].....	4
Figure 1.4 - Semi-supervised structure [10].....	5
Figure 1.5 - Example of classification errors on overlapping craters [15].....	7
Figure 2.1 - Convolution operation on the local receptive field [16].....	10
Figure 2.2 - Convolution operation with a stride length of (2,3) [16].....	10
Figure 2.3 - Convolution operation with zero-padding [16].....	11
Figure 2.4 - Average pooling operation using a kernel filter [16].....	12
Figure 2.5 - Max pooling operation using a kernel filter [16].....	12
Figure 2.6 - Fully connected layer [16].....	14
Figure 2.7 - Simple visualization of the generic gradient descent algorithm [16].....	17
Figure 2.8 - Gradient descent algorithms representing smaller and larger learning rates [16].....	17
Figure 2.9 - Common issues with training neural networks [19].....	19
Figure 2.10 - Dropout regularization technique [20].....	20
Figure 2.11 - Typical neural network training using backpropagation [19].....	22
Figure 3.1 - High level visual representation of modern object detection architectures [21].....	25
Figure 3.2 - Comparison between older C2F block and the newer C3K2 block [22].....	27
Figure 3.3 - SPFF module and how it interacts with generic input data [22].....	28
Figure 3.4 - New C2PSA block incorporating PSA for attention mechanisms [22].....	29
Figure 3.5 - Overview of the YOLOv11 model architecture [22].....	30
Figure 4.1 - Example .PNG image taken by the LRO NAC [25].....	33
Figure 4.2 - NASA Lunar Orbital Data Explorer [25].....	34
Figure 4.3 - Example 640x640 patch generated from Figure 4.2.....	35
Figure 4.4 - Patch labeled using Label Studio.....	37
Figure 5.1 - Example of a confusion matrix.....	40
Figure 5.2 - Low quality image of misidentification due to surface variation.....	41
Figure 5.3 - Mathematical representation of Intersection over Union (IoU).....	42
Figure 6.1 - Lunar patch used for all basic inference testing.....	51
Figure 6.2 - Model description between Nano and Small.....	54
Figure 6.3 - Precision-confidence curve performance through each model version.....	56
Figure 6.4 - Recall-confidence curve performance through each model version.....	58
Figure 6.5 - F1-confidence curve performance through each model version.....	60
Figure 6.6 - Precision-recall curve performance through each model version.....	61
Figure 6.7 - Model version 1 inference test (confidence minimum of 0.5).....	63
Figure 6.8 - Model version 2 inference test (confidence minimum of 0.5).....	64

Figure 6.9 - Model version 3 inference test (confidence minimum of 0.5).....	65
Figure 6.10 - Model version 4 inference test (confidence minimum of 0.5).....	66
Figure 6.11 - Model version 5 inference test (confidence minimum of 0.5).....	67
Figure 6.12 - Nano model precision-confidence curve.....	68
Figure 6.13 - Small model precision-confidence curve.....	69
Figure 6.14 - Nano model recall-confidence curve.....	69
Figure 6.15 - Small model recall-confidence curve.....	70

Chapter 1 - Introduction

1.1 Motivation

Artificial Intelligence (AI) is a revolutionary technology utilized in many industries such as healthcare, finance, automotive, and manufacturing. One branch of AI is Machine Learning (ML) which allows computers and machines to learn from data, mimicking human learning processes, and carry out tasks autonomously without human input [1]. As these systems are exposed to more data, the performance and accuracy of a particular task are improved [1]. In recent years, ML has become a critical tool in the aerospace industry, particularly for autonomous navigation and hazard detection. This project leverages ML to develop an autonomous system capable of identifying hazards, such as craters and boulders, during lunar landings. Over the past decade, there has been a significant rise in global interest in lunar exploration, reflected by the growing number of lunar landing missions. In 2025 alone, there are 6 commercial lunar missions scheduled to deliver scientific experiments and technology demonstrations through NASA's Commercial Lunar Payload Services (CLPS) program [2]. Landing on the Moon presents several challenges due to the complex terrain, which is dotted with craters and boulders, all of which pose significant hazards to landing spacecraft. These hazards come in varying sizes that range from small, barely visible depressions or rocks to large, mission-ending obstacles. Accurately identifying and avoiding these obstacles is critical to ensuring the safety and success of any lunar landing mission. This task is difficult as some terrain features are undetectable below a certain size to orbiting satellites, meaning onboard sensors are unable to pick up these features when selecting a landing site [3]. A landing site that looks promising from orbit might end up having certain terrain features that are only detectable or visible to a lander that is very close to the surface, this is due to the unpredictable terrain. However, it's not just terrain features that can pose a risk. Another variable that needs to be considered is the dynamic lunar environment. This makes detecting hazards from orbit challenging due to the variability in lighting conditions, background noise, sensor limitations, and the rugged terrain. Additionally, the vast distance between Earth and the Moon introduces communication delays of 5 to 14 seconds mainly due to increased delays in the Deep Space Network (DSN) [4]. This delay makes it virtually impossible for real-time manual control and hazard avoidance during landing as any input from an operator would take a considerable amount of time to register. Any time delay in actions and responses can have a diverging impact on rapidly dynamic situations that would otherwise require near instantaneous monitoring and control [4]. In response to these conditions it's reasonable to say that any lunar landing missions will require some level of autonomy to safely carry out the mission's objectives. Using autonomous systems can potentially enhance the reliability, precision, and safety of future lunar missions, reducing the risks associated with carrying out unmanned landings.

1.2 Literature Review

Computer Vision (CV) is a branch of AI that allows machines to utilize several different methods to process and understand visual information like images and videos [5]. One particularly useful method of CV is object detection, which allows for autonomous systems to locate, identify, and label objects within an image or video. ML and CV are both fields that are closely intertwined together as CV is able to harness the algorithms employed by ML to make decisions or locate specific objects based on the gathered visual information [5]. Combining these two subfields is powerful when employed in scenarios where visual interpretation is critical for locating, identifying, labeling, and making decisions based on specific objects or features. For example, autonomous vehicles utilize CV to capture and analyze images, while ML algorithms identify and label various roadway hazards such as vehicles, pedestrians, etc. [5]. Going further, two ML techniques in particular are predominant to a ML-CV system, these two techniques are supervised learning and Deep Learning (DL). Supervised learning is a technique that makes it possible for a computer to train a ML algorithm using labelled data [5]. DL is a method that uses multilayered Neural Networks (NNs) that mimic the human brain's ability to accurately recognize, classify, and describe objects within a given dataset [6]. These NNs, otherwise known as deep NNs, are a bundle of three or more complex layers of nodes that are interconnected forming the “thinking” process. The first of these layers is the input layer where the DL model receives the visual data for processing. At the end of the model, the output layer gives the final prediction for what it believes to be the correct classification of objects [6]. A breakdown of AI and its subfields is shown in Figure 1.1 [7].

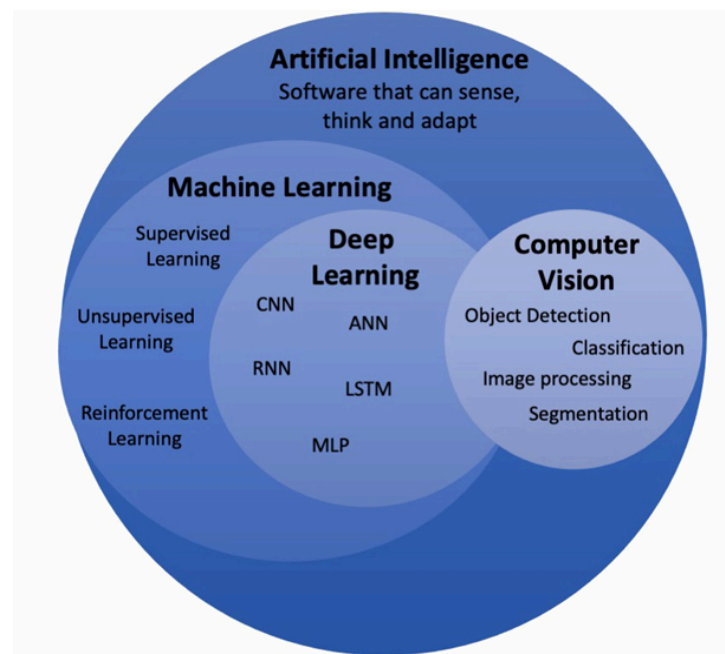


Figure 1.1 - AI overview with branching subfields [7]

The last and perhaps most important part of a visual-based, identifying, and labelling autonomous system, is the Convolutional Neural Network (CNN). CNNs combine all of the previously discussed topics; it's a NN consisting of multiple layers, each serving a specific function in the extraction and interpretation of features from an input image or video. CNNs are a widely used DL algorithm first introduced in the 1960s and showcased promising results in CV [8]. There are three main CNN methods that are widely used in CV applications: image classification, object detection, and segmentation. Four different types of layers make up a CNN: a convolution layer, pooling layer, fully connected layer, and a nonlinearity layer [8]. Convolution layers apply filters to an input and extract the most prominent features by employing a kernel filter. Pooling layers combine convolutional layers, reducing the number of parameters allowing the network to focus on the most important features while improving computational efficiency. Fully connected layers receive input from the final pooling or convolutional output layer and interpret these abstracted features to make classifications or predictions. Nonlinearity layers introduce non-linearity into a CNN model, which is essential for complex learning that would otherwise act as a simple linear function. An example of a generic CNN model is highlighted in Figure 1.2 [8]. There are many different CNN pre-trained algorithms each with their own benefits and drawbacks. The difference in pre-trained algorithms is mainly due to two significant factors: the model structure configuration and the pre-trained layer parameters. CNN models are typically pre-trained on large datasets consisting of hundreds or even thousands of images, training on a large dataset can help fine tune the learning characteristics of a model. Better learning characteristics make a model more suitable for learning a new dataset.

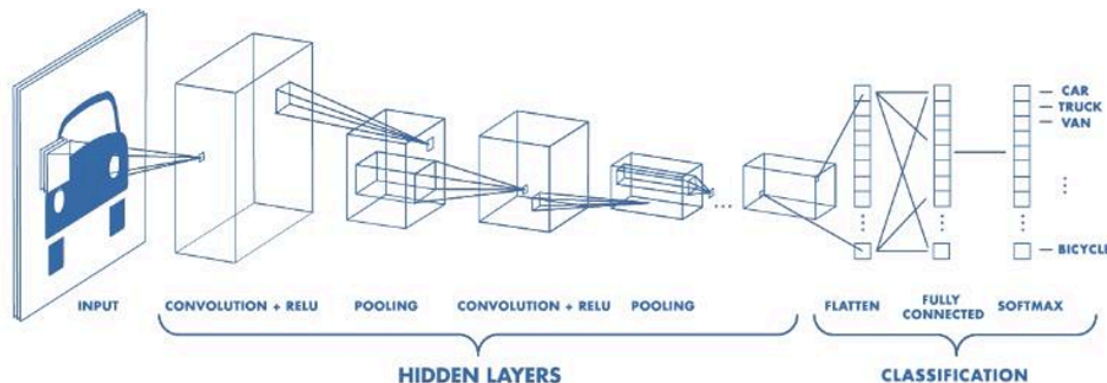


Figure 1.2 - Example of a generic CNN architecture [8]

CNNs have proven to be very useful across a range of industries due to their ability to process, identify, and understand visual information. The deployment of partial or full self driving capabilities in autonomous vehicles is mainly driven by the advancement of CNN object detection algorithms [9]. Object detection has two main objectives: identifying objects within visual information and classifying those objects. After classification, the object is typically labeled with a bounding box to display where exactly they are located in the image. Object

detection algorithms fall into two main categories: one-stage detectors and two-stage detectors. The architecture for these two categories is shown in Figure 1.3 [9]. A comparative study on vehicle detection using CNNs evaluated three primary object detection algorithms: Faster R-CNN, YOLO v3, and Single Shot Detector (SSD). Faster R-CNN is a computationally heavy, two-stage detector algorithm that utilizes a Region Proposal Network (RPN) with a Fast R-CNN detector. The RPN generates proposals for potential objects, these proposals then feed into the Fast R-CNN for object classification. YOLO v3 is a one-stage detector that is a more advanced algorithm than its YOLO predecessors, employing a deeper CNN pretrained on the ImageNet dataset. SSD is a foundational, one-stage detector algorithm that runs derived feature maps from images through additional convolutional layers to make predictions on where identifying bounding boxes may be. To train these algorithms, a dataset consisting of 652 images was put together with 391 images being used for training and 261 images being used for testing. The study found that while all three models achieved over 99% accuracy, the SSD algorithm demonstrated the fastest average detection time at 0.5 seconds and the highest Mean Average Precisions (mAP) of 0.92 [9]. The Faster R-CNN, while accurate, was slower due to its two stage process, averaging a detection time of 5.1 seconds and a mAP score of 0.76 [9]. YOLO v3 landed in the middle and struck a balance between speed and accuracy, achieving a mAP of 0.81 with an average detection time of 1.16 seconds [9]. The results highlight the differences between detection speed and precision amongst three different CNN models all trained on the same dataset. These performance metrics are important when choosing a CNN architecture for detecting hazards. A high accuracy, precision, and fast detection speed is critical when there is limited time to identify and classify these hazards during descent. However, an equally important metric to consider is computational complexity since an autonomous system would likely have limited computing power onboard a lunar lander.

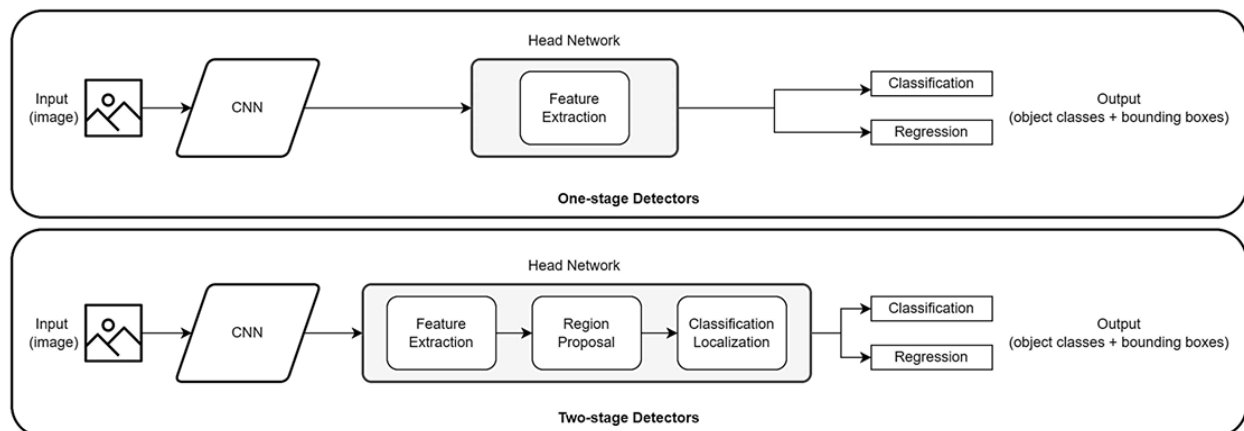


Figure 1.3 - Architectural difference between one-stage and two-stage detectors [9]

The application of CNNs and object detection has already shown to have a significant impact on the development of the aerospace industry. With the increase in orbiting spacecraft and space debris in LEO, detecting hazards is important for the safety of any spacecraft. In one study, a Semi-supervised Object Detection (SSOD) algorithm is used with enhanced feature extraction capabilities, the detection of orbiting spacecraft and space debris was improved over previous models [10]. The methods used boosted detection accuracy while maintaining efficiency by staying within computational limits. SSOD uses a student-teacher approach that is beneficial when training data is limited. The SSOD framework is showcased in Figure 1.4 [10]. The model used the SPARK space target image synthetic data set to train and test on 11 types of space targets and 15,000 RGB images. SPARK takes into account environmental conditions such as lighting, background, target distance, and sensor noise. Synthetic images of space targets were generated using a digital 3D software called Unity3D game engine [10]. In some instances, synthetic images are quicker to generate and provide a large volume of data but for planetary imagery, a large collection of images is readily available. Additionally, using data augmentation can be an even more efficient way to increase the volume of pre-existing data being fed to an object detection model. The core of the model is based on the CNN object detection model YOLOv5 with an additional learning framework called EfficientTeacher (ET) [10]. This model was chosen for its accuracy, high inference speed, and excellent deployment structure along with its ability to optimize and improve the network. In many cases, YOLO models have shown to be exceptional when deployed in tasks that require near real-time object detection with the need for a high degree of accuracy and precision [11]. The latest v12 model of YOLO builds on many areas of its predecessors. It harnesses the capabilities of modern NVIDIA Graphics Processing Units (GPUs) outperforming previous models in speed and efficiency in real-time applications [11].

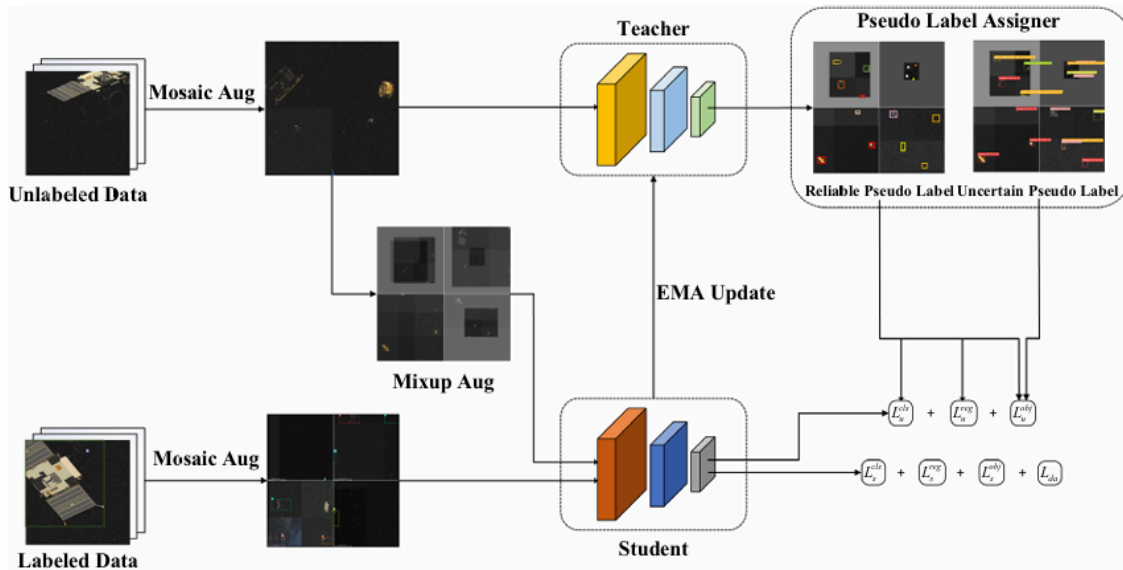


Figure 1.4 - Semi-supervised structure [10]

A similar use case for CNNs utilizes the YOLO architecture to develop an advanced model for smaller crater detection [12]. This model, called YOLO-crater, integrates several innovations to address data issues and improve detection accuracy all while reducing interference from the lunar environment. Digital Orthophoto Maps (DOMs), Digital Elevation Models (DEMs), and slope data from Chang'e-2's (CE-2) onboard sensors were collected and used to construct 23 sample datasets to train the model [12]. Data collection, preprocessing, and labelling are very important steps when training a supervised deep CNN model. There is a large collection of unprocessed data for applications involving crater and boulder detection on the lunar surface. One source of this data is from the Lunar Reconnaissance Orbiters (LROs) Narrow Angle Camera (NAC). Images of the lunar surface from NAC have resolutions as high as 0.5 to 2 meters, making for detailed images of surface hazards [13]. These images often cover large areas so it's necessary to use data augmentation techniques to break down the image into a larger volume of data. Combining NAC images with DEMs obtained from the LRO's Lunar Orbiter Laser Altimeter (LOLA) instrument allows for the development of DOMs [13]. DOMs take out the perspective distortions created by taking detailed aerial images of a spherical surface. Training CNNs on the most accurate data is crucial to developing a mission ready system. Research has gone into creating a dataset that could capture a wide range of crater characteristics to improve detection accuracy [14]. Skipping any step in the data collection and augmentation process can lead to unrealistic classifications which decreases the accuracy and precision of the model.

There are many CNN models to choose from; one such model is the Mask R-CNN architecture, which is used to automate lunar crater detection [14]. The Mask R-CNN model is a hybrid model that uses both object detection and segmentation and is an extension of the Faster R-CNN model architecture. Segmentation works on a pixel by pixel basis as opposed to scanning large areas of the image for distinguishing features like object detection. The combination of these two methods means that the model not only identifies the hazard but it also produces a pixel mask to label the obstacle. The model can be broken down into three steps. First, it uses feature extraction to locate and identify. Second, it displays a bounding box around where it thinks a crater might be located. Finally, the pixel mask is applied to the final classification displaying the objects [14]. Segmentation can be a useful tool if near exact locations of objects are necessary but obtaining this level of accuracy can decrease computational efficiency. This method can be useful in handling scenarios where objects are overlapping which is a prominent feature of lunar hazards. The dataset for training this model was derived using DEMs from several pre-processed catalogs of lunar images. The results showed a high level of accuracy in detecting craters with a mAP score above 90% [14]. No explicit results were showcased about the runtime for detection but these results still show how the Mask R-CNN trained on the dataset was able to detect craters accurately and precisely.

Using CNNs for crater detection can be useful when using relativistic optical navigation for lunar landings [15]. This approach aims to improve the precision and efficiency of optical navigation systems. LRO images from onboard cameras were combined with DEMs to provide comprehensive visual data for training and testing the model. The CNN model was trained using a combination of supervised learning and transfer learning techniques and combined two different architectures. Still leveraging object detection, there was a significant improvement over traditional image matching techniques using the model [15]. Using a similar detection algorithm, this CNN based crater detection algorithm is practical for use in near real time autonomous lunar landings. Figure 1.5 shows how the model performed on a large crater that overlaps with several smaller craters [15]. Overlapping craters present a significant challenge to accurate detection of surface hazards. If these smaller hazards go undetected, an autonomous system might attempt to land in an area that would put it at risk of achieving its mission objectives.

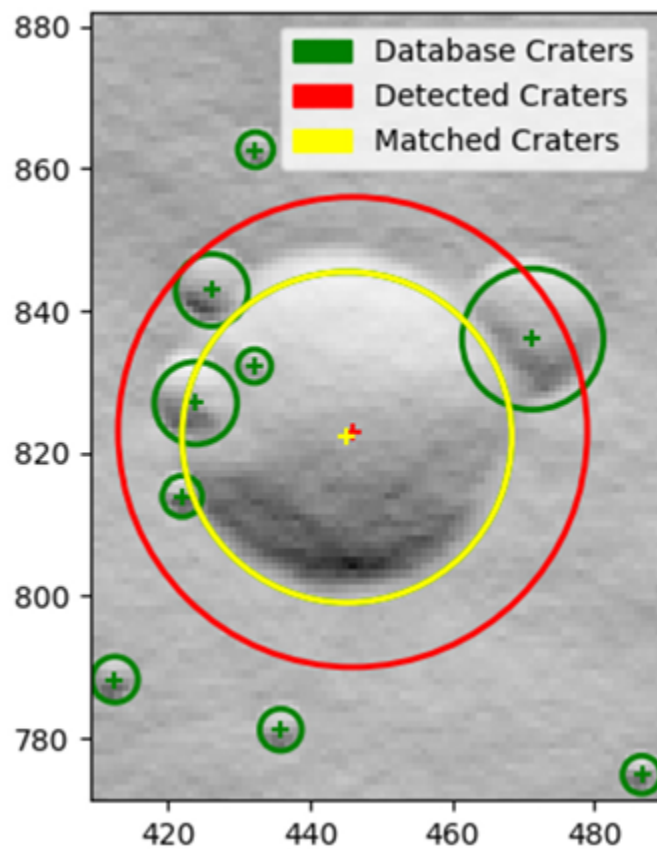


Figure 1.5 - Example of classification errors on overlapping craters [15]

1.3 Project Proposal

The objective of this project is to develop an autonomous system using CNN models to identify and classify lunar surface hazards. This system will extract and analyze features from lunar surface images captured by cameras onboard orbiting spacecraft. Using object detection, the model will be capable of processing visual information and providing near real time labeling of hazards. DL techniques will provide the capability to train the model to recognize a variety of hazard sizes and shapes under realistic lunar environmental conditions. These environmental conditions include the variation in lighting, angle of observation, and sensor limitations that may be prevalent in optical systems. Additionally, the model will need to take into account the presence of overlapping craters, surface irregularities, and terrain elevation. Focus will be placed on improving the precision and accuracy of the model while ensuring a fast detection time to meet real time processing requirements. The successful implementation of this system could reduce landing risks, improve landing site analysis, and increase navigational accuracy.

1.4 Methodology

To achieve the objective of this project, three steps will be taken to produce a high quality model capable of lunar hazard detection. The first step will focus on developing a dataset of lunar images capable of training and testing the model. Surface images and elevation data from the LRO will be sourced from the LROC and LOLA instruments, respectively. Data will need to be highly detailed and provide clear views of hazards to be suitable for use by the model. Data augmentation will be used to improve feature visibility and generalization, reduce overfitting, and increase the volume of data. This training data will need to be labeled to provide the model with what the output should look like to improve its accuracy. The second step will be to design and train the CNN for hazard detection. The goal is to find the best model architecture that will provide enhanced accuracy, precision, and computational efficiency while reducing runtime. Pre-trained models will be utilized to reduce training time and the volume of the dataset required by leveraging transfer learning. The final step will be to evaluate and characterize the performance of the model. Metrics such as *precision*, *recall*, mAP, detection time, and F1 score will be assessed to determine the accuracy and speed of detection. Testing and evaluation of the model will be done on data it has not seen before. This ensures the model will be introduced to a completely new environment to verify the model's generalization capabilities and accuracies.

Chapter 2 - Theoretical Framework

2.1 Convolutional Neural Network

The Convolutional Neural Network (CNN) is a widely used specialized class of deep learning and computer vision model designed to process images or videos. Expanding on the content from Chapter 1, the CNN employs a supervised training approach to learning spatial features making them highly effective in image processing and classification. Unlike traditional neural networks, CNNs are able to extract features automatically by learning to identify edges, shapes, and textures within a given set of data or images. The 4 layers that make up a typical CNN are the convolutional layer, pooling layer, nonlinearity activation layer, and fully connected layer. Each layer within the CNN contributes to refining, extracting, and processing the input data making it possible for autonomous object detection and classification when trained properly.

2.1.1 Convolutional Layer

The core building block of any CNN architecture is the convolutional layer. This layer makes it possible to detect local patterns or features in an image by applying a filter across the entire image and calculating the convolution, sometimes referred to as the cross-correlation. These filters are known as kernels and need to have a filter size that is smaller than the size of the input image but a channel size that is the same [8]. In other words, the kernel is a three dimensional matrix that has a length, width, and depth. The area of pixels that the kernel operates on is known as the local receptive field with the results of the convolution operation being referred to as the hidden layer [16]. When computing the convolution of an image, the filter slides across the image at specific pixel step intervals and estimates the product between the kernel weights and the given set of pixel values [8]. After the convolution product is calculated, there is a bias value offset that is added to make up for any losses accrued during calculations. Computing the convolution generates a feature map, also known as an activation map. This feature map is used as the input for the next layer with the features getting more refined and the continual downsampling of the image. The mathematical representation of the convolution operation with a bias offset for a single channel 2D image is shown in equation (2.1).

$$y[i, j] = b + \sum_x \sum_y^{H W} K_{[x, y]} \cdot I_{[i-x, j-y]} \quad (2.1)$$

Here, y represents the output feature map, K is the kernel, I is the input image, H and W are the height and width of the image respectively, and b is the bias. Each i - j th position is calculated using equation (2.1). Figure 2.1 illustrates how the convolution operation is carried out on the local receptive field using the kernel filter and stored in the feature map [16]. Equation (2.1) can be thought of as the classic mathematical representation of a neural network. In this

instance the output would be an activation level, the kernel and input represent the input and weight from the previous layer, and the bias would be the correction term.

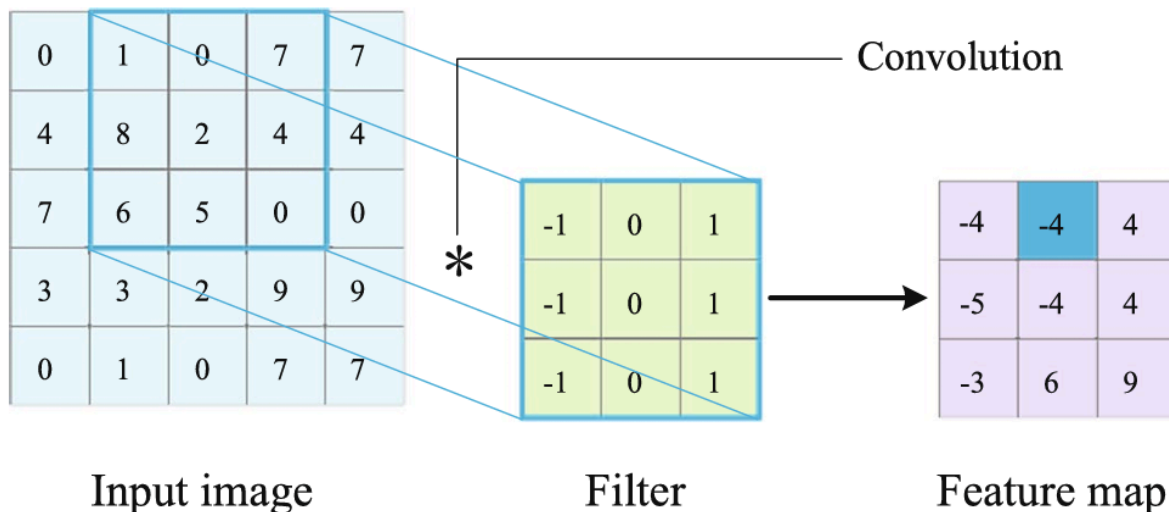


Figure 2.1 - Convolution operation on the local receptive field [16]

When defining a convolutional layer there are two parameters that are important for fine tuning, they are the stride length and padding. The stride length dictates the number of rows and columns the kernel filter moves across the image during the convolution operation. For example, a stride of one means that the filter will move across the image one pixel at a time preserving the image resolution. Increasing the stride length can impact the quality of the output data, size, and feature extraction capabilities while increasing the computational speed [16]. Figure 2.2 shows an example of how a stride length greater than one can dictate the movement of the kernel across the image. These effects need to be considered as a larger stride may be more computationally efficient for object detection but may significantly impact the accuracy and precision of the model. The stride length can be different at each new convolutional layer across the entire CNN model.

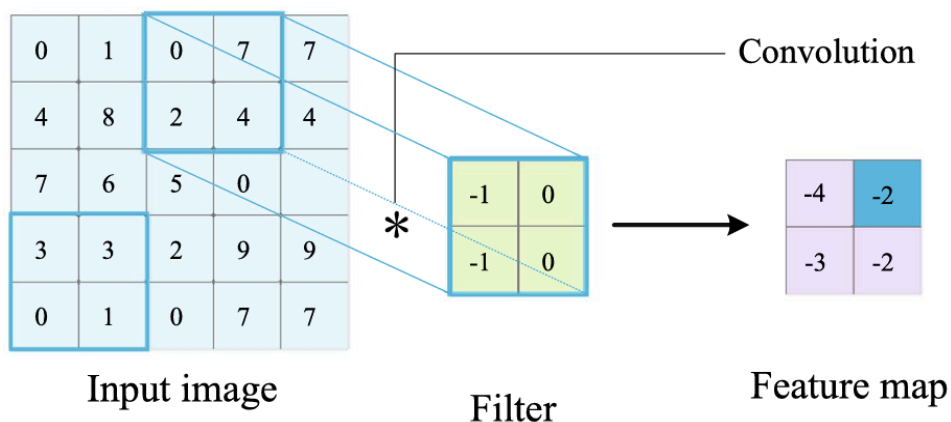


Figure 2.2 - Convolution operation with a stride length of (2,3) [16]

Padding controls how the boundaries of an image are handled during convolution. This process adds artificial pixels to the edges of an input image either by adding a layer of pixels with values of zero or copying the boundary pixels for padding [16]. Figure 2.3 highlights how padding can be applied around the edges of an input image specifically by adding a layer of zero valued pixels. Padding is necessary to ensure that the output of the convolutional layer retains the same dimensions as the input, preventing information loss between layers. This parameter can also be used to make it easier for the CNN model to learn the information surrounding the input image or it can be used in reverse to train the model to focus on objects toward the center of the image.

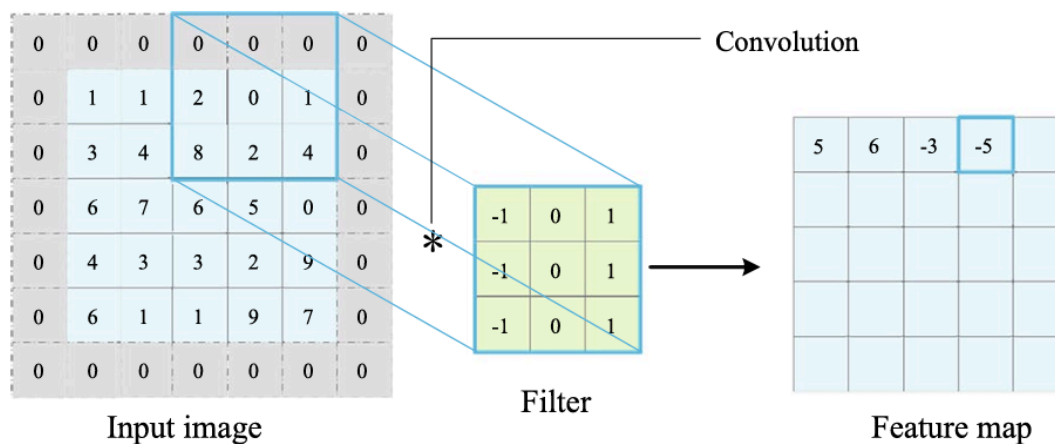


Figure 2.3 - Convolution operation with zero-padding [16]

2.1.2 Pooling Layer

Once a convolutional layer has completed its operations on the image, the feature map generated is input into the pooling layer. The pooling layer is an important operation that increases computational efficiency by downsampling the feature map or essentially reducing the number of parameters, decreasing the feature map dimensions, and minimizing overfitting [16]. The pooling layer utilizes one of two main functions to operate on the feature map, these functions are average pooling and max pooling. Both functions employ a kernel filter to run through the input dataset and carry out the computational operations, similar to the process done during convolution.

The average pooling function takes all the elements within the local receptive field and takes the average [16]. This new value is output to a smaller feature map that will be the input for the next network layer. Figure 2.4 illustrates how the average pooling operation is done on a given randomized dataset using a 2x2 filter with a stride length of two [16].

The max pooling function is the most commonly utilized function and outputs the max value within the local receptive field. This method essentially extracts the most significant features within a given local receptive field and outputs them to the next layer [16]. An example

of this operation is shown in Figure 2.5, where this operation is carried out using a 2x2 filter with a stride length of two [16]. In both pooling operations, the functions downsample the original feature map from the 4x4 input to the final 2x2 feature map.

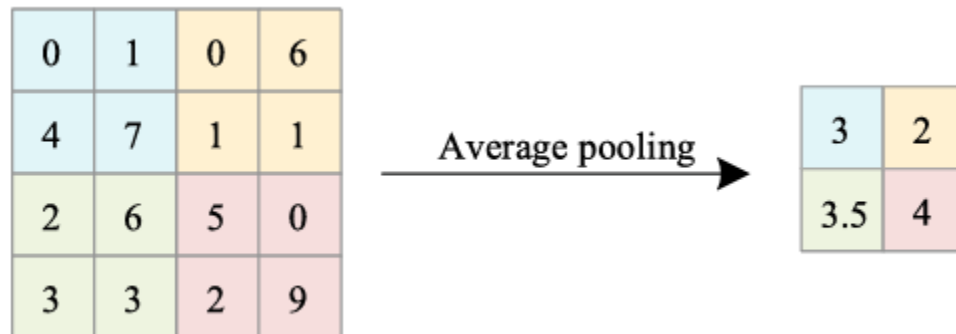


Figure 2.4 - Average pooling operation using a kernel filter [16]

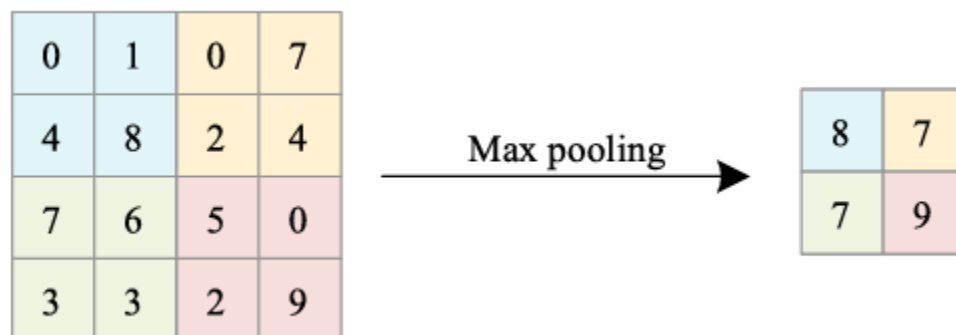


Figure 2.5 - Max pooling operation using a kernel filter [16]

2.1.3 Nonlinear Activation Layer

The nonlinear activation layer plays an important role in any CNN architecture. An activation layer is a type of mathematical function that is typically placed after a convolutional or fully connected layer and receives the output of that layer as an input. The main purpose of the activation function is to approximate any type of nonlinear functions, making it capable of working with a wider range of complex nonlinear models [16]. This is essential, linear models alone are not effective at capturing real world data which is often seen as being highly nonlinear. It is also necessary to ensure that any small changes occurring between layers provides meaningful output to help distinguish between different objects, textures, and patterns. Therefore, at each layer within the network, the activation function determines which features are most important or retained and which are suppressed.

There are several types of activation functions, the most common activation function is the Rectified Linear Unit (ReLU). The ReLU function linearizes the input by assigning any

negative input values to zero while positive input values remain unchanged. The mathematical representation of the ReLU function is showcased in equation (2.2).

$$f(x) = \max(0, x) \quad (2.2)$$

Some additional benefits of ReLU over other activation functions are that it speeds up gradient convergence which decreases the chances of gradient problems that occur frequently [16]. Furthermore, since this is a quite simple linear relation, the computational load is significantly reduced when compared to other activation functions [8]. In deep neural networks, gradient instability is a common issue that leads to gradient vanishing or exploding. When a derivative is greater than 1 this results in an exponential explosion, on the other hand, when the derivative is less than 1 this leads to an exponential decay causing the vanishing gradient problem [16]. ReLU solves this issue since its derivative almost always equals one due to the similarity to the ramp function.

2.1.4 Fully Connected Layer

The fully connected layer is an essential component in a CNN, typically being placed at the end of the network and dealing only with classification. Taking multiple feature map inputs from a convolutional or pooling layer, the fully connected layer takes a long vector of these feature maps and outputs them to the desired output class [16]. It operates like a traditional neural network where each neuron computes a weighted sum of its inputs, applies a bias term, and passes the results through to the next layer of neurons. This process enables the network to learn certain patterns within images and form autonomous complex decision making based on feature extraction. Despite increased computational complexity, these layers are essential in the decision making process and integrating them enables a CNN to learn and recognize complex patterns. Figure 1.2 shows exactly where the fully connected layers are located within a CNN [8]. Looking more closely, Figure 2.6 illustrates how the neurons are connected from one layer to the next until reaching the final bundle of neurons for complex feature classification [16].

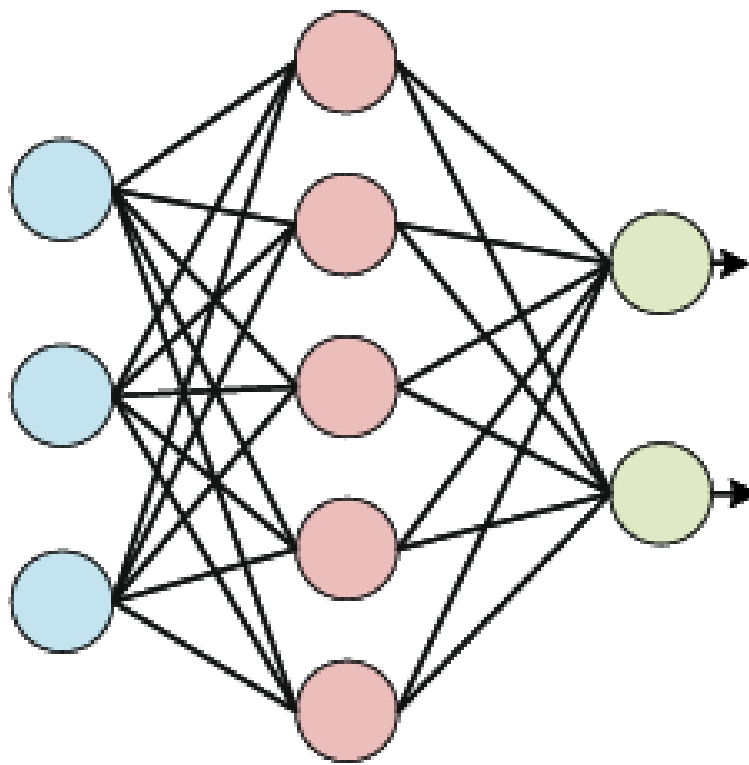


Figure 2.6 - Fully connected layer [16]

2.2 Backpropagation

Backpropagation is the foundational algorithm used to train and optimize CNNs to identify relevant features within images. Utilized during supervised learning, this process continuously updates weights and biases in the neural network that can help minimize loss [16]. Updating each parameter is done by calculating how those parameters contribute to the final prediction and comparing that answer with what is expected. In this case, the algorithm would look at the training data given and make its own predictions then it would find the error between the predicted and true labels to improve the accuracy of the network. Backpropagation uses mathematical concepts such as the loss function, gradient descent, and regularization to train complex CNNs.

2.2.1 Loss Function

The loss function, or sometimes referred to as the cost function, is an important mathematical operation used in backpropagation to compute the error. The error for any neural network is calculated using the distance between the predicted and true label. This essentially allows the model to figure out how inaccurate it is with regard to what the expected outcome is. Without this function, the system would not be able to provide feedback to itself or figure out how it can improve the performance of feature extraction and classification. There are two main types of tasks for machine learning that a loss function can be put under, these two categories are known as regression and classification. Regression tasks are when a network makes continuous predictions based on a set of input values. Classification tasks are when the network assigns a probability, typically between 0 and 1, that certain features fall within a specific category [17]. In this case, classification is what object detection would fall under.

Classification tasks use several different types of loss functions, the most common being the cross-entropy loss function. Cross-Entropy (CE) loss outputs predictions as a probability value between 0 and 1 corresponding to the likelihood that the prediction belongs to a certain class or category [17]. The mathematical representation of CE loss is shown in equation (2.3).

$$L = - \sum_{k=0}^K \hat{y}_k * \log(y_k) \quad (2.3)$$

Here, L represents the computed loss, y represents the prediction made by the network, $\hat{y}$ represents the true label or category of the k -th position, and K represents the number of classes or categories. The logarithmic function is important as it penalizes confident but incorrect predictions while uncertain predictions are less severely penalized. For example, the function will give a score of around 0.1 to 0.2 for smaller differences while larger differences will generally get a score of around 0.9 to 1 [18]. This helps the network learn more cautiously and

avoid overconfidence on predictions that are incorrect. This is just one of many loss functions used for classification tasks with many more used for regression problems.

With such a large number of loss functions, it is important to make sure the chosen loss function is the best for a particular task. There are several important factors to consider when choosing the type of loss function. These factors are: binary vs multiclass classification, sensitivity, and computational efficiency.

Binary classification involves making predictions on data to find the probability that the feature is most likely between two different categories. Multiclass classification is the same except the feature has a probability of being placed into more than two categories [17].

Sensitivity looks at how the loss function deals with predictions that skew the overall statistical distribution of the results. A highly sensitive model might penalize a prediction more based on the fact that the prediction does not directly align with what the true label is. A less sensitive model will not penalize these predictions making the model more susceptible to correctly classifying a wider range of unseen data [17].

Computational efficiency is an important consideration when computational power is not readily available. Certain loss functions may take up more computational power as they require more complex mathematical functions while others will use simple arithmetic to compute the error of a prediction.

To summarize, the loss function plays an important role in the training of a CNN or any generic neural network. With careful consideration of the computational efficiency, sensitivity, and classification type the best loss function can enable a CNN to translate complex features into accurate predictions. Its role in driving the learning process in backpropagation and shaping the decision-making capabilities makes it vital in object detection algorithms.

2.2.2 Gradient Descent

CNN training with backpropagation relies heavily on the chain rule of calculus to compute the gradient of the loss function. The gradient indicates the direction of the maximum rate of change for a function at some point of interest [16]. This is otherwise known as the gradient descent and is at the heart of the optimization process used in backpropagation to fine tune the model's key parameters and minimize the loss function. During training, the gradient descent algorithm is iteratively applied and continuously steps in the opposite direction of the gradient until it reaches convergence [16]. As defined by Zhao et al., the generic mathematical expression for the gradient descent is better represented in equation (2.4).

$$\theta = \theta - \alpha \frac{\partial J(\theta)}{\partial \theta} \quad (2.4)$$

Here, θ represents the parameter to be optimized, α is the learning rate, $J(\theta)$ is the loss function or the function that needs to be minimized [16]. A visual representation of the interaction between equation (2.4) and some generic function is shown in Figure 2.7.

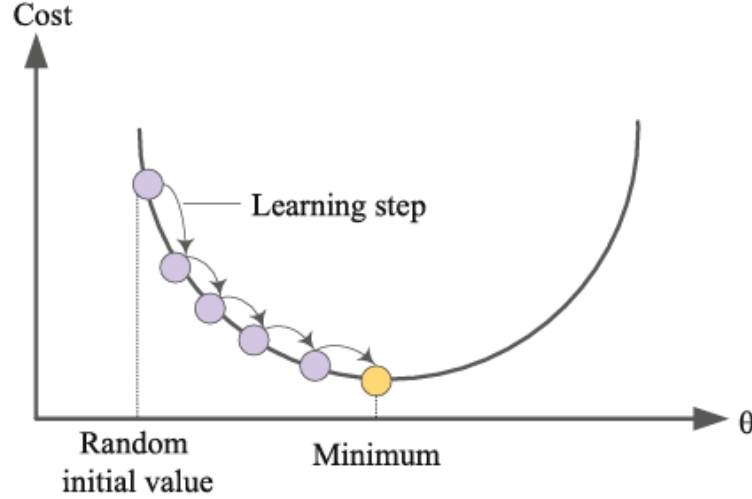


Figure 2.7 - Simple visualization of the generic gradient descent algorithm [16]

The learning parameter is what determines how fast and smoothly the error converges to some minimum of the loss function. This parameter can be thought of as the step size taken during each training cycle. Figure 2.8 highlights how different learning parameters can affect the convergence of the loss function to a function minimum. If the learning rate is too small, the algorithm will converge very slowly thus taking many iterations to reach a convergence that is acceptable as shown in Figure 2.8 (a). For a learning rate that is large, the algorithm might overshoot the function minimum and be stuck in a loop constantly overshooting as shown in Figure 2.8 (b).

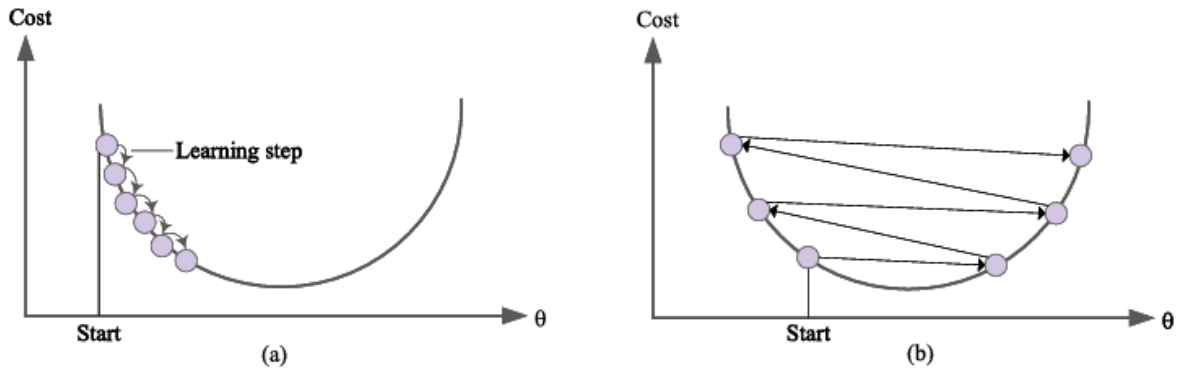


Figure 2.8 - Gradient descent algorithms representing smaller and larger learning rates [16]

There are many gradient descent algorithms, one of the most commonly used variants for training CNNs is the Stochastic Gradient Descent (SGD) algorithm. SGD uses a single data point or mini-batch size at each iteration to estimate the gradient instead of computing the gradient across the entire training dataset. This variant has some benefits such as faster convergence of the error and may help the algorithm get out of a local minimum and into a more optimal position such as the global minimum. The drawbacks of this method are the introduction of noise into the optimization pipeline making the gradient calculations oscillate [18]. The SGD algorithm is similar to equation (2.4) as shown in equation (2.5).

$$\theta = \theta - \alpha \frac{\partial J(x^i, y^i; \theta)}{\partial \theta} \quad (2.5)$$

Here, all the parameters from equation (2.4) are represented the same. The main difference is the addition of the loss function taking into consideration 2 more parameters, x and y , which can be generic parameters that are updated after each iteration.

Gradient descent and the chain rule form the backbone of the backpropagation method for training CNNs as it heavily relies on computing the gradient to update the weights and biases. By applying the chain rule iteratively from the output layer through to the input layer, the error signals can be distributed and make it possible to adjust all weights in proportion to their contribution to the overall error. SGD helps accelerate the optimization process by taking into consideration more parameters and speeding up the convergence of the loss function. As stated previously, SGD is just one of many gradient descent algorithms that are used in training neural networks. Both SGD and the chain rule make it possible to optimize the training of a CNN model to help fine tune the parameters necessary to accurately detect and classify specific objects within data.

2.2.3 Regularization

Regularization is an optimization method that aims to improve the quality of training for the model during backpropagation. The main aim is to prevent overfitting and enhance the generalization capability of the model by adding onto the loss function. Overfitting occurs when a model performs well at feature detection during training but performs poorly at generalizing to unseen data during testing [19]. This issue is prominently found in models with many layers and parameters, or in other words, CNN models. Regularization optimization methods essentially add constraints and modifications to the backpropagation training process by building robustness within the model to learn simpler and prevent memorization of training data. A visual representation of typical issues that arise during model training, including overfitting, is shown in Figure 2.9.

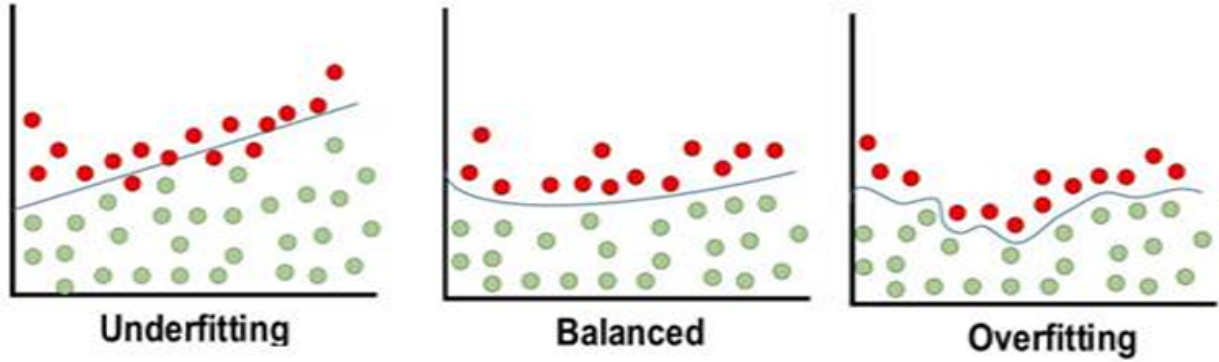


Figure 2.9 - Common issues with training neural networks [19]

There are several regularization methods that can help reduce or solve the overfitting and generalization problems. The most commonly used methods are L2 regularization, dropout, and batch normalization. More often than not, all of these methods are used together and employed in pre-trained models to help improve the learning capabilities of the model to a given dataset.

L2 regularization, sometimes referred to as weight decay, adds a penalty term to the loss function to discourage the model from assigning large-value weights to its parameters [20]. This is often used to help the model generalize better as it requires the model to rely on smaller weights, which are usually more stable than weights with larger values. Shrinking the feature weights to near zero instead of making them equal to zero makes it so that small changes to the input of a neuron do not have large changes to the output of that neuron. Mathematically the loss function now uses the squared sum rather than the absolute sum of the weights when computing the total loss or error of the output [20]. This new mathematical model for the loss function is showcased in equation (2.6).

$$\theta_{total} = \theta_{initial} + \lambda \sum_{k=0}^K w_k^2 \quad (2.6)$$

Here, w^2 is the weight squared, θ is the total and initial loss function respectively, K represents the number of classes, and λ is the regularization parameter. The regularization parameter is a tunable value that determines how much impact the summation has on the total loss function. L2 regularization is a classic method when it comes to optimizing the training process. It makes the model's predictions more stable and reliable by increasing the generalization capabilities making it less susceptible to error.

The next most utilized training optimization method is called dropout. Dropout is the method by which random nodes or neurons are deactivated along with their input and output connections during each training cycle [20]. This method forces the model to learn in different and redundant ways to recognize features since it has to take new paths to achieve accurate results while missing neurons. A particular troubling problem that is addressed with the dropout

method is co-adaptation. Co-adaptation is the tendency for neurons to rely heavily on other neurons, when employing dropout, these neurons that are relied on may be deactivated forming better generalization skills to unseen data. This forces neurons with these reliant connections to adapt and change the way they compute the outputs. Murel and Kavlakoglu provide a good visual representation of the dropout regularization method in Figure 2.10 [20].

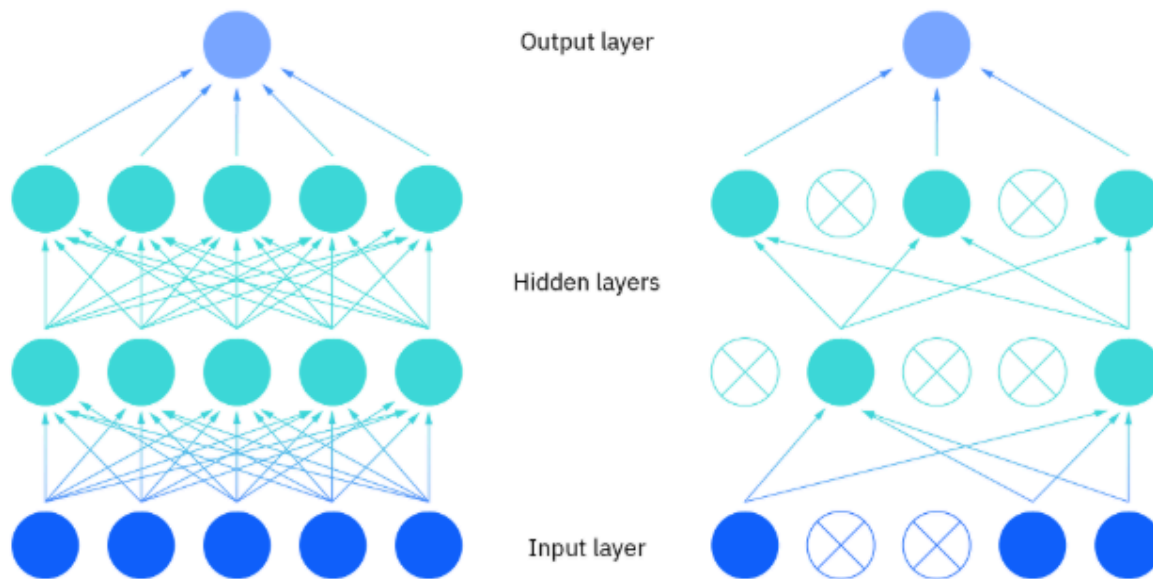


Figure 2.10 - Dropout regularization technique [20]

The last optimization method is batch normalization. Batch normalization works by normalizing the activation of a layer and unifying the distribution of the values found within a feature map by setting them to unit variance and zero mean [16]. Originally developed to address problems in gradient descent algorithms, it also serves as a model regularization technique [16]. The main benefits of this method are improving the gradient vanishing and explosion problem, speeds up the model training process, enhances the model's ability to generalize different types of input data, and helps reduce the chance of overfitting [16, 19]. Similar to dropout, batch normalization acts as a mild form of stochastic regularization by introducing noise into each training cycle making the model more robust and better at generalization.

Optimization strategies like L2 regularization, dropout, and batch normalization play a key role during model training. These methods are often used together to reduce overfitting and improve model performance. L2 helps reduce the amount of influence that large weight values have on the loss function. Dropout encourages the model to become more flexible and build robustness by pushing it to learn alternative paths to make predictions. Batch normalization helps keep the training process stable by normalizing layers to support consistent activation patterns. Together, these techniques form a solid foundation for training models that can generalize well beyond the training data.

2.3 Updating Weights and Biases

One of the most fundamental steps when training a model is to update the network's weights and biases. This is the final and perhaps most important step in the process as all of the results from backpropagation are now utilized to adjust the accuracy of the model. The weights and biases are essential parameters that allow the model to learn how to map the inputs to the outputs with a high degree of accuracy. The ultimate goal for adjusting these parameters is to reduce the error in the network between the predicted label and true label. The generic form of the guiding mathematical expression that represents how the weights are updated is highlighted in equation (2.7). This expression doesn't take into consideration the effects of the regularization techniques but it is easy to see how those methods can be accounted for in equation (2.7).

$$w_{i+1} = w_i - \alpha \frac{\partial L}{\partial w_i} \quad (2.7)$$

Here, w is the new and current weight, L is the loss function, and α is the learning parameter. This expression updates the weight parameter by taking the difference between the current weight and the gradient of the loss function, with respect to the current weight, multiplied by some tunable learning parameter. Similarly, the mathematical expression that represents how the biases are updated is shown in equation (2.8). Again, this expression does not take into account regularization but instead is showcased in the generic form to better understand the foundational mathematics behind training CNNs or any neural network using backpropagation.

$$b_{i+1} = b_i - \alpha \frac{\partial L}{\partial b_i} \quad (2.8)$$

Here, α and L again represent the learning parameter and loss function, respectively, with b representing the new and current bias. This expression functions similarly to equation (2.7) in terms of how it computes the updated parameter.

As training progresses, the updated parameters will converge to some value that will decrease the error or loss function to a local or global minimum. Early in the learning process, the change in the parameters is large as the model makes significant errors in predicting that it has yet to correct. As the predictions get more accurate, this shift in parameters becomes more subtle since the gradient of the loss function will move closer to zero with each training cycle. Taye utilizes Figure 2.11 to visually show where the different elements of the training process connect to the model.

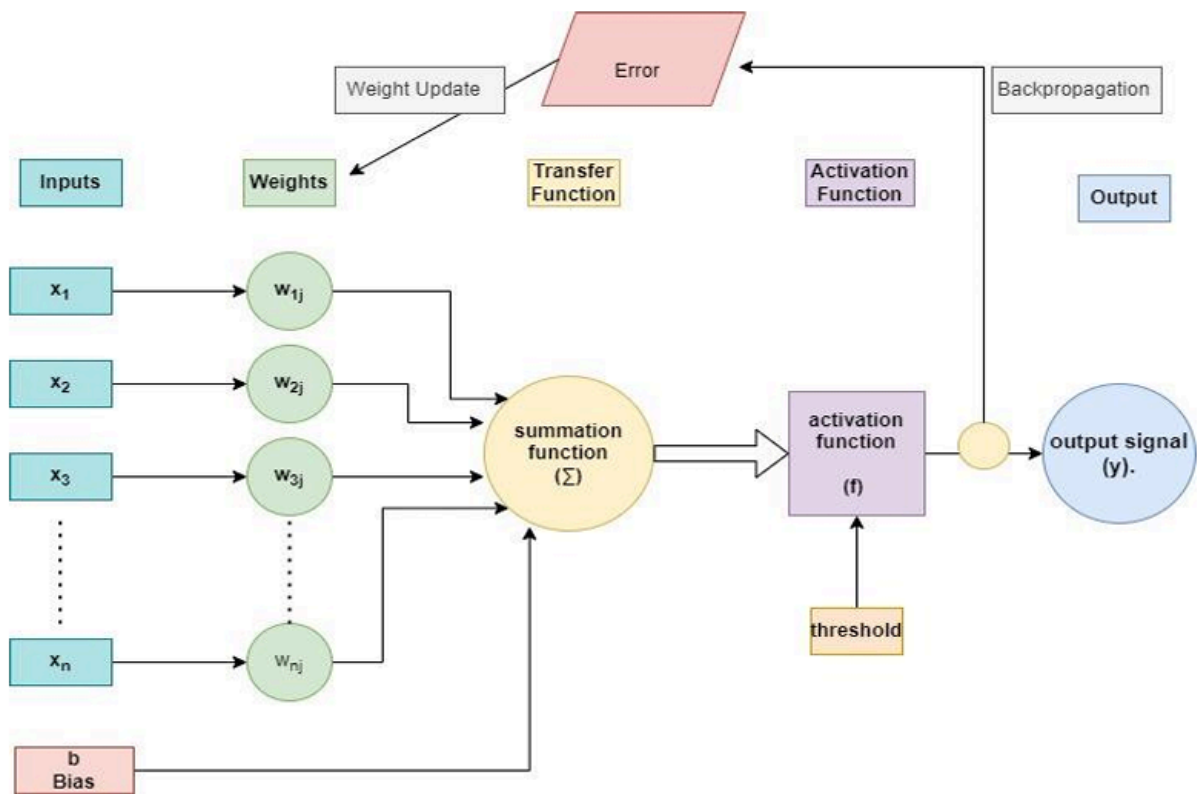


Figure 2.11 - Typical neural network training using backpropagation [19]

Chapter 3 - System Architecture

3.1 YOLO Architecture

You Only Look Once (YOLO) is an object detection model that has become widely adopted due to its high speed, accuracy, and ease of implementation. Unlike traditional two-stage detectors such as R-CNN, which first propose candidate regions before classifying them, YOLO streamlines the process by performing object localization and classification simultaneously within a single network pass. This single-stage approach means that the YOLO model is able to predict bounding boxes and assign class probabilities at the same time instead of doing them consecutively. A visual representation showing the difference in model structure between a single-stage and two-stage detector is presented in Figure 1.3. Thanks to its fast detection capabilities, the YOLO architecture is widely used in various object detection tasks, making it especially suitable for applications that require near real-time performance [11]. This project aims to leverage a pre-trained YOLO model to detect, identify, and label craters on the lunar surface with data from orbiting lunar spacecraft.

The way that YOLO processes images is by breaking up the input data into multiple cells, predicting the bounding boxes, and assigning a class probability to show the likelihood of a feature being within a certain class. For example, when the center of a feature falls within a cell, that cell is responsible for classifying what type of object it is and where it is located within that cell [16]. Even though the YOLO model is fairly new, being just less than a decade since first proposed, there have been 12 main versions of YOLO that have greatly improved on the legacy performance while still maintaining a high detection speed. Early YOLO models had difficulty making accurate predictions on features that were small in the image but were able to perform at high speeds. These issues are not so prevalent with newer models, such as YOLOv11, bringing them up to par, in terms of accuracy, with slower two-stage detector models. Hussain uses Table 3.1 and 3.2 to give a brief visual overview of the YOLO models from v1 to v8 [11]. From Table 3.1, the architectural foundation of each YOLO version is named which highlights how this model evolved over time making large changes. Likewise, Table 3.2 shows the framework that each model uses as recent models shift to utilizing PyTorch more often than Darknet. The mechanisms that change from each version to the next are what impact the performance of the model the most.

Table 3.1 - YOLO model evolution [11]

Variant	Year	Architecture
YOLOv1	2016	GoogLeNet-inspired
YOLOv2	2017	Darknet-19, VGG
YOLOv3	2018	Darknet-53
YOLOv4	2020	CSPDarknet53
YOLOv5	2020	CSPDarknet53
YOLOv6	2021	EfficientRep, CSPDarknet53
YOLOv7	2022	ELAN, CSPDarknet53
YOLOv8	2023	CSPDarknet53

Table 3.2 - YOLO models from v1 to v8 [11]

Variant	Framework	Dataset	Mechanisms
YOLOv1	Darknet	VOC Pascal Dataset	Grid-based approach
YOLOv2	Darknet	VOC Pascal, COCO Dataset	Hierarchical classification
YOLOv3	Darknet	COCO Dataset	Feature Pyramid Network
YOLOv4	Darknet	COCO Dataset	Enhanced quantization, PAN, RepVGG
YOLOv5	PyTorch	COCO Dataset	AutoAnchor, Mosaic, MixUp, EfficientNet
YOLOv6	PyTorch	COCO	RepVGG, PAN, EfficientRep
YOLOv7	Darknet	COCO Dataset	ELAN, Model scaling
YOLOv8	PyTorch	COCO Dataset	C2f module, EfficientRep, CIoU, DFL

The general architecture of YOLO consists of three main components: the backbone, the neck, and the detection head. Each component plays a crucial role in enabling swift object detection capabilities for the YOLO architecture. The backbone component of the YOLO architecture is responsible for extracting features from the input image [21]. There are many backbone architectures utilized in CNN models, some common architectures are DarkNet, EfficientNet, and AlexNet. These classification architectures include many convolutional layers that form the foundation for the feature extraction capability. The neck is an important component that is after the backbone and before the detection head. It's responsible for aggregating and combining the features extracted from the backbone layers to enhance object detection performance [21]. The main benefit is to help detect small, medium, and large objects within the input image which is output as a multi-scale feature map to the detection head. Some examples of neck architectures are a Feature Pyramid Network (FPN), Path Aggregation Network (PANet), and Bi-FPN. The detection head component deals with the decision making at the end of the YOLO model. The main function is to take the feature extraction maps generated by previous layers and perform classification and localization [21]. Essentially, this part of the model assigns a probability that tells which category the detected features are most likely a part of. Along with this, the detection head deals with visualizing the bounding box that shows where these features are located within the input image. Terven provides a simple yet helpful high level visualization of how the backbone, neck, and head make up the entire YOLO model in Figure 3.1 [21].

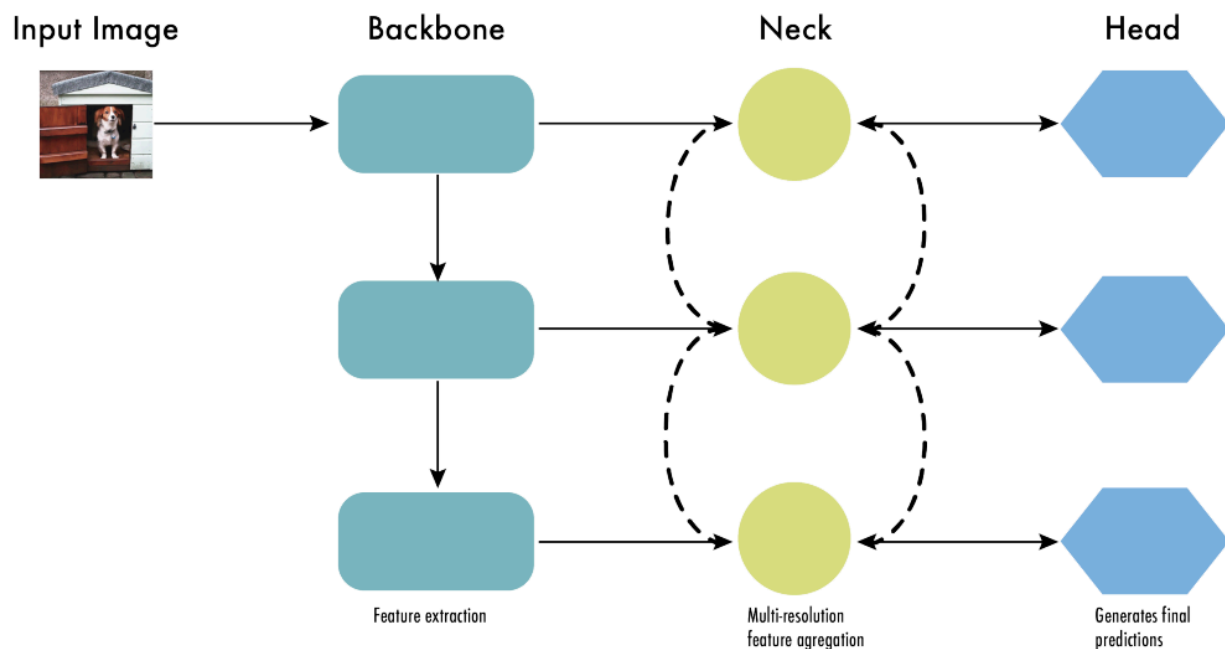


Figure 3.1 - High level visual representation of modern object detection architectures [21]

3.2 YOLOv11

YOLOv11 is a high fidelity model released in September of 2024 by Ultralytics and boasts a performance improvement from previous models in accuracy and speed. This latest model changed and upgraded many aspects of the internals to achieve this performance increase. Table 3.3 shows an increase in the performance metrics from YOLOv8, a widely used and high performing version released in 2023, to the latest v11 architecture [23]. This increase in performance when given the same image and training is highlighted in metrics such as Mean Average Precision (mAP), Speed, and FLOPs. These metrics are discussed in a later chapter but for now a brief description will be given to help understand why these metrics are important. The mAP is a common metric that highlights how well the model balances *precision* and *recall* abilities, across Intersection over Union (IoU) thresholds, with a higher value being associated with better performance [22]. The YOLOv11n model shows a higher mAP value of just over two points when compared to YOLOv8n. This increase in mAP is significant when taking into consideration the decrease in the average time it takes for the model to process the image. All this is done while decreasing the amount of FLOPs, a value representing the computational cost of the model. Another important thing to note is the decrease in the number of parameters that are utilized in the newer model. Less parameters means the model doesn't need as many training cycles to fine tune the model as it can potentially reach convergence of the loss function quicker.

Table 3.3 - Performance metrics between YOLOv8 and v11 [23]

Model	size (pixels)	mAP ^{val} 50-95	Speed CPU ONNX (ms)	Speed T4 TensorRT10 (ms)	params (M)	FLOPs (B)
YOLOv8n	640	37.3	80.4	1.47	3.2	8.7
YOLOv8s	640	44.9	128.4	2.66	11.2	28.6
YOLOv8m	640	50.2	234.7	5.86	25.9	78.9
YOLOv8l	640	52.9	375.2	9.06	43.7	165.2
YOLOv8x	640	53.9	479.1	14.37	68.2	257.8
YOLO11n	640	39.5	56.1	1.5	2.6	6.5
YOLO11s	640	47.0	90.0	2.5	9.4	21.5
YOLO11m	640	51.5	183.2	4.7	20.1	68.0
YOLO11l	640	53.4	238.6	6.2	25.3	86.9
YOLO11x	640	54.7	462.8	11.3	56.9	194.9

Early YOLO models introduced powerful real-time object detection capabilities. YOLOv11 brings several architectural enhancements aimed at further improving these capabilities making the model more accurate and faster. With each iteration of YOLO, the model becomes more streamlined with new and upgraded features addressing the limitations of the previous models. The limitations that continue to plague the YOLO model are around small object detection and generalization across varied datasets. The evolution to YOLOv11 represents a shift toward a more modular, scalable, and efficient architecture more suitable for real-time object detection.

Many improvements were made to the backbone, neck, head, and attention mechanisms in YOLOv11. More specifically, the main architectural improvements are the upgrading and implementation of the C3K2 block, Spatial Pyramid Pooling Fast (SPFF) module, and the Cross Stage Partial with Spatial Attention (C2PSA) block [22].

YOLOv11 introduced a modified transformer-enhanced backbone that integrates local and global feature extraction more effectively. An evolution of the Cross Stage Partial (CSP) bottleneck, the C3K2 block played a crucial role in this modification and was responsible for the feature extraction capabilities present at various stages of the backbone. C3K2 enhances information flow within the network by dividing the feature map and using multiple 3x3 convolutional layers [22]. These smaller kernels are more efficient and less computationally demanding compared to larger ones. With a similar structure to the Cross Stage Partial Focus (C2F) block, the C3K block does not include splitting and instead passes the results from the convolution layer straight through to the bottleneck blocks. The C3K block is the foundation of the C3K2 block as it is incorporated within the architecture focusing on increasing the accuracy without sacrificing the speed. Rao uses Figure 3.2 to clearly show the differences between the blocks used in the backbone of the old and new architectures.

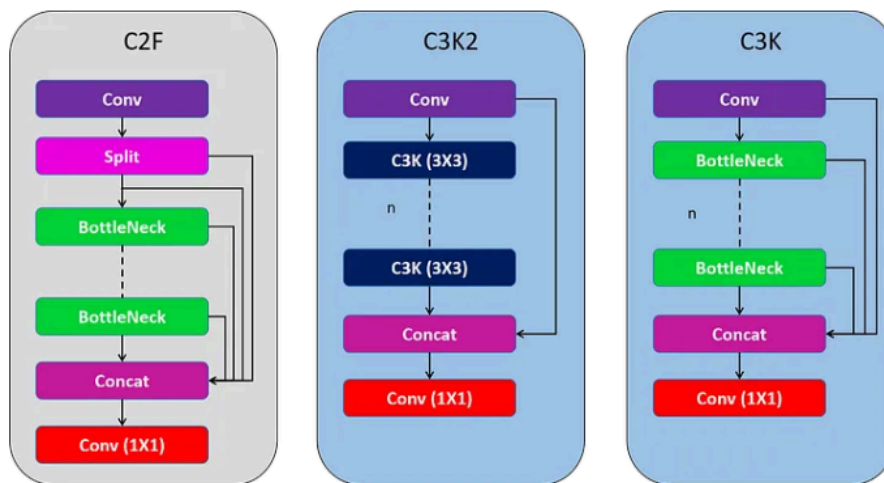


Figure 3.2 - Comparison between older C2F block and the newer C3K2 block [22]

The neck of the module didn't see a drastic change with the architecture only minor changes to the SPFF module and the inclusion of a hybrid Feature Pyramid Network (FPN). This module is important for the aggregating process and improves the network's ability to capture features of varying sizes [22]. The hybrid FPN was introduced to the v11 neck of the model to enhance multi-scale detection performance. It improves variable size detection by better preserving the spatial information of features at different scales. These two improvements do not have a huge impact on the performance but do help continue to address the problems surrounding the detection of objects across multiple scales. Figure 3.3 gives a good visual representation of how the SPFF module is structured and how it manipulates generic data to preserve spatial information.

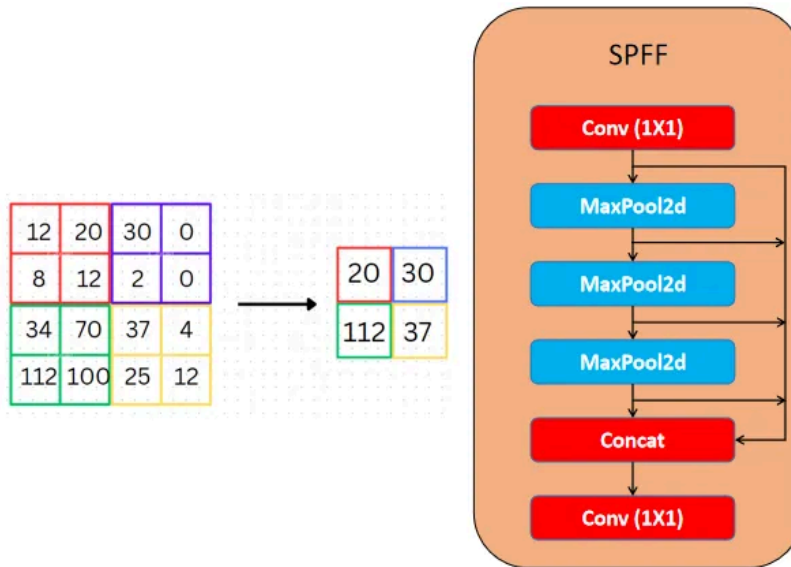


Figure 3.3 - SPFF module and how it interacts with generic input data [22]

Attention mechanisms are an important component of deep neural networks as they provide a model with the ability to focus on important features in an image. With YOLOv11 comes the introduction of the Cross Stage Partial with Spatial Attention (C2PSA) block. This block enhances the overall capabilities of the model by introducing these attention mechanisms. More specifically, the introduction of attention allows for the model to focus on smaller or partially occluded objects by making spatial relevance more of a driving factor when looking at feature maps [22]. The new C2PSA block uses two Partial Spatial Attention (PSA) modules, which are essential for giving the model access to attention capabilities. How the C2PSA incorporates two PSA modules into the architecture can be easily seen in Figure 3.4. Each of these PSA modules are applied to distinct branches of the input feature map which are then subsequently merged as the output. With these inclusions, YOLOv11 achieves improved accuracy over earlier versions, particularly in tasks requiring precise detection capabilities.

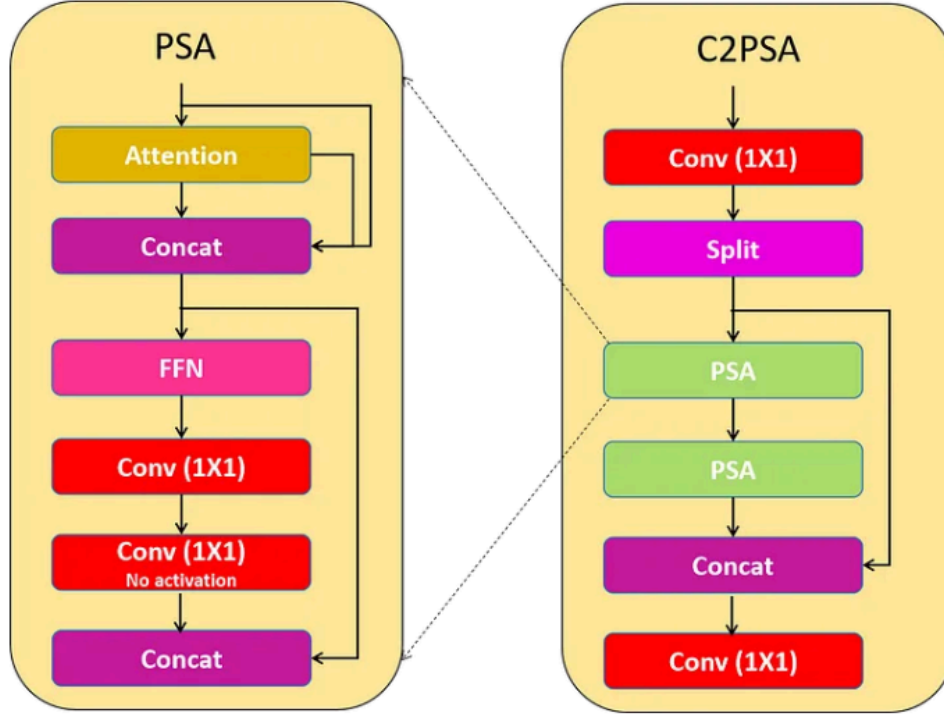


Figure 3.4 - New C2PSA block incorporating PSA for attention mechanisms [22]

The head of the YOLOv11 model uses similar detection and multi-scale prediction abilities to earlier YOLO versions [22]. With the use of multi-scale prediction, the model can detect objects at varying sizes making it more useful for tasks requiring these features. Using the feature maps generated by previous layers, detection boxes and labels can be placed on multiple objects with different sizing. This again addresses the problem of multi-scale detection that was discussed earlier in the chapter.

As discussed in Chapter 2, the activation function is an important layer in a CNN model that allows for nonlinear functions to be accurately modeled. While ReLU is the most common and popular activation function, YOLOv11 instead uses the sigmoid activation function sometimes referred to as the Sigmoid Linear Unit (SiLU).

$$SiLU(x) = x\left(\frac{1}{1+e^{-x}}\right) \quad (3.1)$$

Here, x simply represents the input feature map from the convolution layer. This function enables smoother and more effective gradient flow during backpropagation giving it an edge over traditional ReLU activations. YOLOv11 continues to rely on SGD as the primary gradient descent algorithm for training.

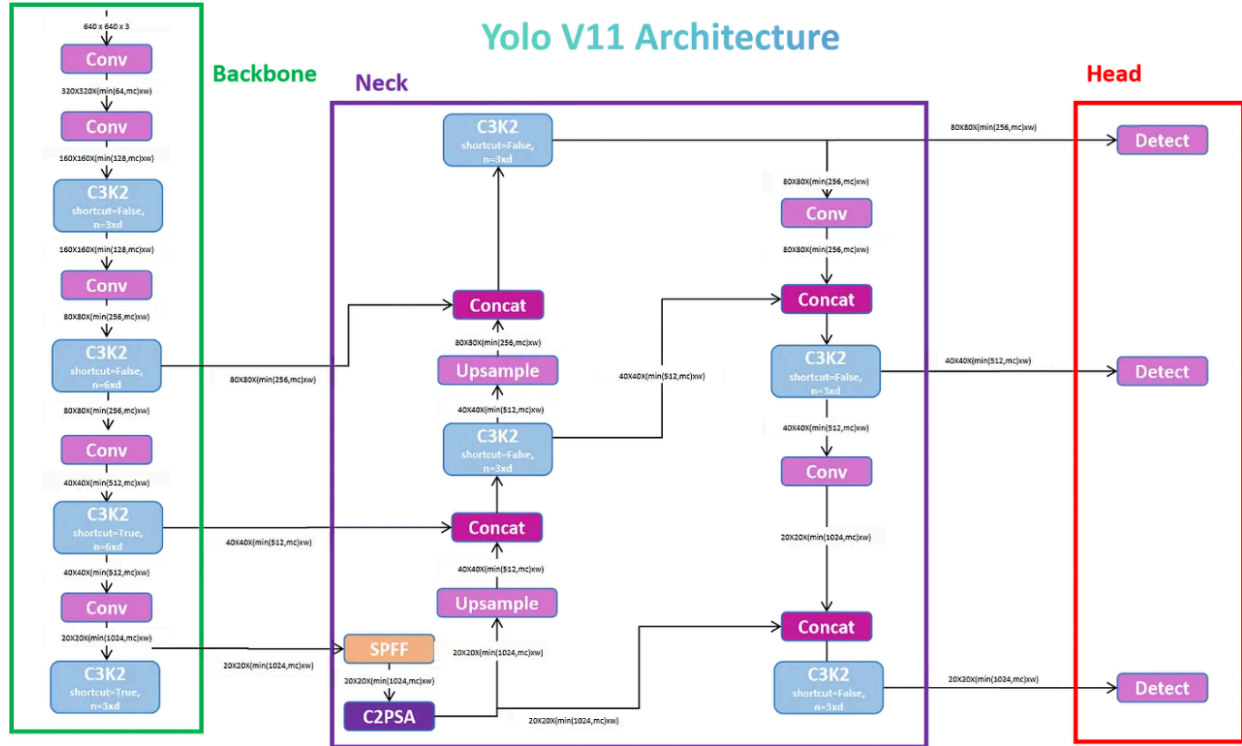


Figure 3.5 - Overview of the YOLOv11 model architecture [22]

The YOLOv11 model provides many benefits over previous versions including enhanced training performance, better prediction accuracy, and faster detection speed. These performance improvements can be linked to the upgraded architecture and addition of new attention features. To better understand the model an overview of the architectural structure for YOLOv11 is shown in Figure 3.5 provided by Rao [22]. The performance of YOLOv11 makes it a viable option to use for this project and develop a crater detection model with near real-time detection capabilities.

3.3 Transfer Learning

Transfer learning is an important tool that allows pre-trained CNN models to utilize feature recognition capabilities, learned from training on other objects, to improve feature recognition on newer objects. This method makes it unnecessary to train a model from scratch which can require a lot of time, data, and computational resources to achieve highly accurate and precise feature recognition. Using pre-trained models in combination with transfer learning can speed up the training process allowing for high fidelity object detection with a comparatively smaller training dataset [15]. This combination means that lower level feature extraction layers don't require as much training or data whereas higher level layers can utilize more computational resources to tailor feature extraction to the specific problem. Overall, this makes for a much more efficient way to improve the performance of the object detection model.

Pre-trained models leverage transfer learning as each model already comes with proficient learning capabilities making them good for use in new applications. Most models typically freeze early layers within their architecture to retain general feature extraction parameters while layers at the end of the network are fine tuned to adapt to the training data for detecting new objects. Using pre-trained models with transfer learning has become a standard practice in object detection applications since it is easier to deploy these models on a variety of platforms.

Chapter 4 - Data Preprocessing

4.1 Data Sources

Data preprocessing is an important aspect of training a high fidelity computer vision model to detect and classify specific features in images. Some would argue that it is just as, if not more, important than developing the architecture of the model itself. Preprocessing is all about developing a high quality dataset that is sufficient enough to train and validate a model to produce positive performance metrics. In other words, the quality and suitability of the dataset play a decisive role in the performance of any supervised computer vision model. Unlike more general tasks where large datasets such as ImageNet, MNIST, or COCO exist, applications such as the one proposed by this project often rely on carefully curated data making data sourcing important. For crater detection, a dataset must capture the variability and subtle changes of the surface that may distinguish craters from surrounding terrain. This dataset must also account for previously mentioned environmental conditions such as lighting, angle of observation, sensor noise, background, and crater distance. Low quality or inconsistent data can lead to a biased model, poor generalization, misclassifications, and inconsistencies in classifying. As previously discussed, sourcing high quality images or data is a critical first step in developing a reliable crater detection model framework.

Several key characteristics that images must have in order to be included in the crater detection dataset have been defined. An important characteristic is high resolution imagery, this is essential for detecting small craters as a crater spanning only a few pixels will be hard to detect. Similarly, images that represent wide coverage across diverse lunar terrain ensures that the trained model can generalize beyond a single environment, this means learning the variability of craters across different types of terrains. Furthermore, images with a large number of craters are important so that the model can have enough features to train and validate on. Similar to having images that represent diverse lunar terrain, images also must vary in their environmental conditions, again, this is so the model can generalize and detect different scenarios.

This project primarily draws on images collected from NASA's Lunar Reconnaissance Orbiter (LRO) which hosts the Narrow Angle Camera (NAC) and Wide Angle Camera (WAC). The NAC has a maximum image scale of 0.5 m/pixel and projected image swath of around 2.5 x 26 km when at a lunar altitude of 50 km [24]. These camera specifications show that images taken by the NAC can provide enough detail and resolution to observe a large number of craters of various sizes. While the LRO does provide the WAC, this imaging device does not take photos at a high enough resolution to observe small craters. The specifications for the WAC are a maximum image scale of 74.9 m/pixel and projected image square swath of around 60 km in the visible color spectrum when at the same lunar altitude [24]. Looking at these camera specifications, it is clear that the most optimal choice to meet the above image characteristics is

the NAC. The NAC can provide the high resolution imagery necessary for detecting small craters, has been utilized across a wide range of diverse lunar terrain, can provide detailed images with a large number of craters, and can showcase various environmental conditions.

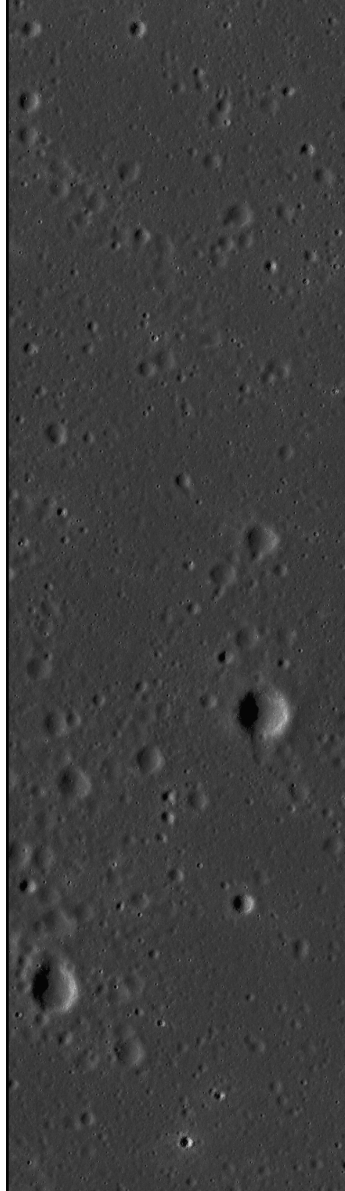


Figure 4.1 - Example .PNG image taken by the LRO NAC [25]

Figure 4.1 is an image that is used in this project to train the model and was taken by the LRO NAC. It is obvious when looking at the image that there is a large number of craters of various sizes with most of the small craters only being seen when zoomed in. Looking at this image, it is 5064 x 17408 pixels and has a bit depth of 8.

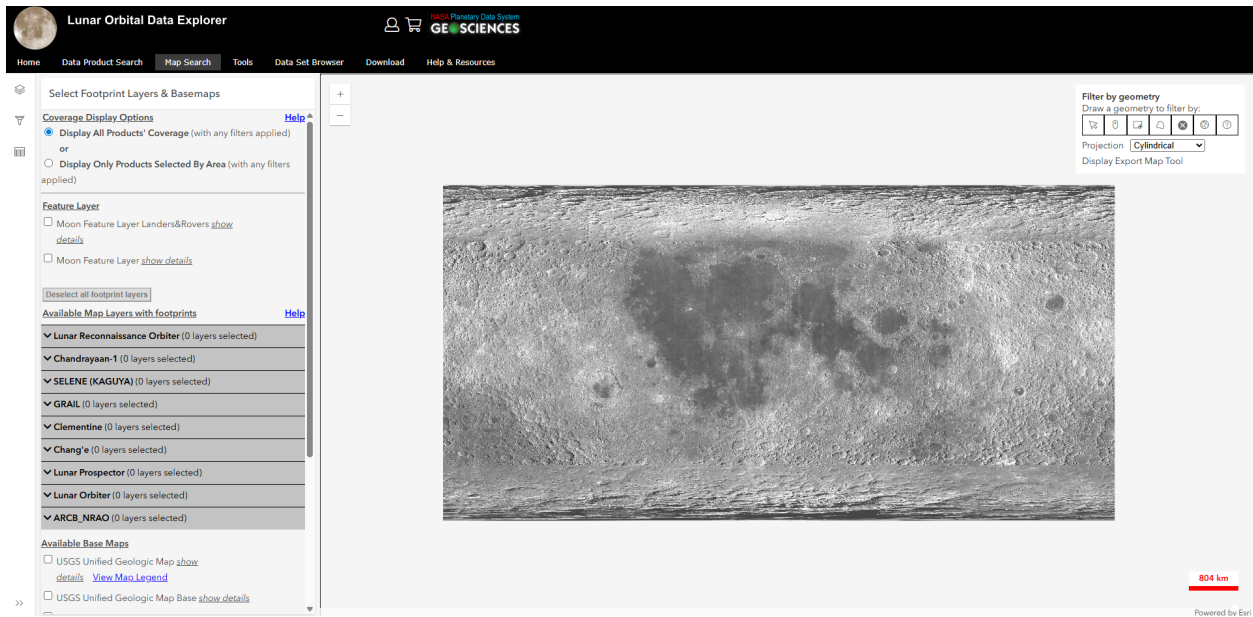


Figure 4.2 - NASA Lunar Orbital Data Explorer [25]

Figure 4.2 shows NASA’s public Lunar Orbital Data Explorer which is a part of the larger NASA Planetary Data System (PDS) and the USGS Astrogeology Science Center. These raw LRO images are originally distributed in .IMG format, this format is unreadable as an image so it has to be converted into a more accessible format during preprocessing. This format preserves image fidelity while allowing easy integration and conversion into a format that is more easily interpreted by modern computers. Although these sources provide comprehensive coverage of the lunar surface, they do have a few limitations. Each individual image comes with fixed conditions, and the presence of image artifacts can introduce several challenges that may affect the quality of training. These issues highlight the need for data preprocessing and image augmentation, both of which are addressed later in this chapter.

4.2 Image Preprocessing

As previously mentioned in Section 4.1, the raw data from the LRO was originally distributed in .IMG format which is a planetary science standard for raster data storage. While this format preserves image quality and metadata, it is not directly usable for computer vision tasks unless it is processed through a data conversion script. To address this, a Python-based preprocessing script was written and used to convert these IMG files into PNG format. The PNG format was chosen over another format like JPG due to its lossless compression, suitability for downstream tasks, and low risk of artifacts occurring during compression. The Python script used for this data conversion process is provided in Appendix A.

The script relies on the Geospatial Data Abstraction Library (GDAL) which is an open-source translator library widely used for working with raster and vector geospatial data. GDAL is a useful library that provides functionality for reading IMG files, accessing pixel values, and converting them into more common formats. The script shown in Appendix A has 4 main functions. The script first opens the IMG dataset and reads its raster band structure, from this structure it computes the maximum and minimum raster values to normalize intensity values. These values are scaled and translated into an 8-bit format essentially mapping the raw pixel values to a 0-255 grayscale range. Finally, this new dataset is exported as PNG, in its lossless format. Converting the raw IMG files into PNG format has not only preserved the image quality but has also enabled the ability to create a diverse dataset. This conversion made it possible to apply methods such as patch generation and general image augmentation which are essential for building a diverse dataset that can capture surface variability of lunar craters and terrain.

After conversion, the images are still too large to be used directly for training as models such as YOLO typically take in images with a resolution of 640x640. These images can have large resolutions, as previously mentioned, Figure 4.1 has a resolution of 5064x17408 pixels. To address this issue, another Python script was used to divide each image into smaller 640x640 pixel patches. Figure 4.3 is a patch that was generated from Figure 4.2 using the Python script that is provided in Appendix B. This script works by moving a 640x640 pixel screen over the image and saving the subsequent area encapsulated in the screen as a separate image.

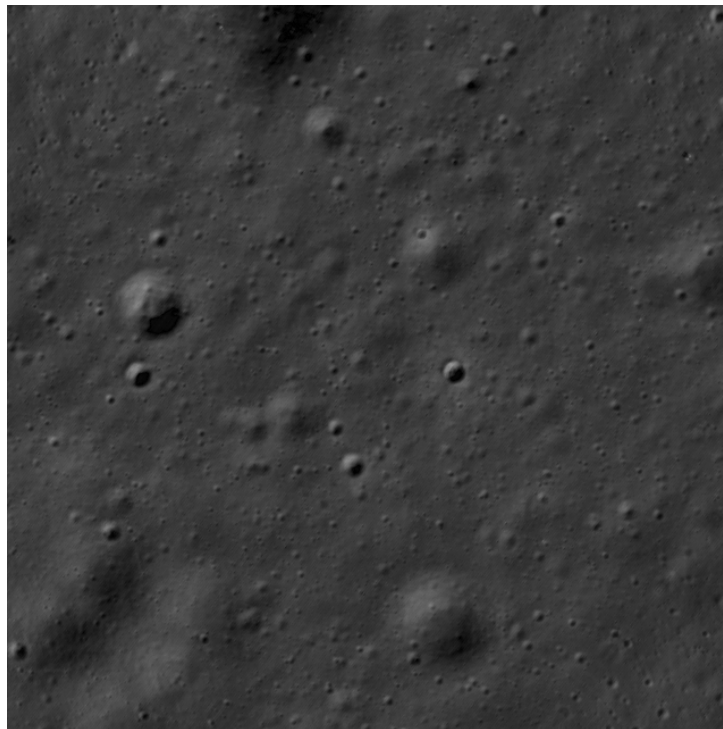


Figure 4.3 - Example 640x640 patch generated from Figure 4.2

After the large images were split into patches, data augmentation was applied to make a dataset that could accurately represent the variability that would be present on the lunar surface. The data augmentation process adjusts the brightness to simulate differences in solar illumination and shadowing and rotates images randomly to account for craters appearing at different orientations. These methods help reduce overfitting by preventing the model from memorizing fixed lighting conditions or orientation patterns. Image augmentation was performed on every patch individually ensuring that variability was consistently represented throughout the training data. This image augmentation Python script is provided in Appendix B.

After the patches have been created and the data augmentation has been completed, each image has to be assessed as to its potential use in the dataset against the key characteristics. Many patches can contain little to no useful information, such as large areas of uniform terrain, excessive shadowing, or regions without a substantial number of craters. Each patch was quickly inspected to identify whether it contained key characteristics that could help benefit the model during training and validation. These characteristics looked for in each image are varying crater sizes, clearly visible illumination conditions, and an abundance of small to large craters. Patches that did not have these key characteristics were discarded, while those that possessed these characteristics were retained and moved forward to the next step in the process.

4.3 Labeling

Once the dataset of augmented and curated lunar surface patches was finalized, the next step was to create the ground truth labels for supervised learning. This process is very important since the performance of the model is highly dependent on the accuracy and consistency of the labels. Labeling was done using an open-source tool called Label Studio which runs directly from the Python IDE environment.

Label Studio provides a simple, browser-based interface that allows images to be uploaded, stored, and annotated with bounding boxes. There are many types of methods for labeling supervised learning data, the most common being bounding boxes which were used to identify individual craters for this project. Each bounding box was drawn to enclose the entire crater as closely as possible to the rim of the crater. This process has to be done carefully to ensure that the labeled features matched what the model was expected to detect.

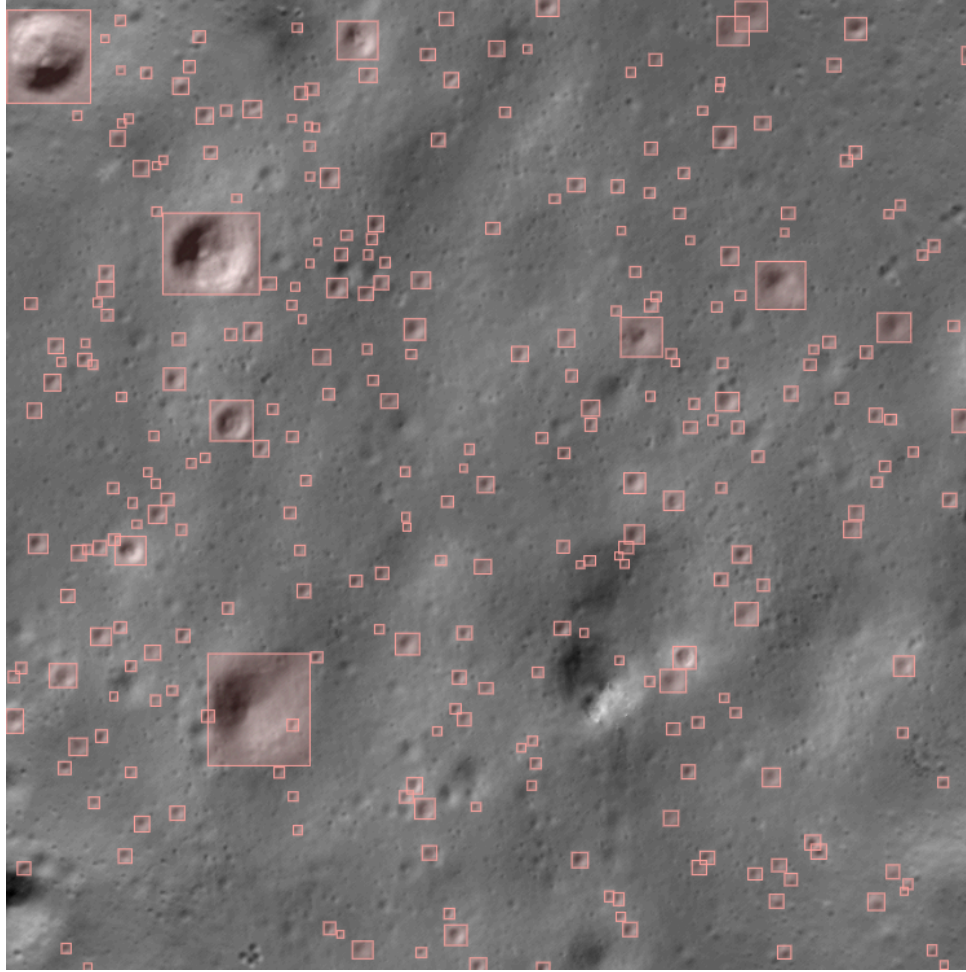


Figure 4.4 - Patch labeled using Label Studio

To ensure consistency and continuity across all the images uploaded to Label Studio, a few guidelines were established. Craters that are on the edge of any patch must have more than 50% of the total crater size visible. Craters that overlap are labeled individually rather than as one large crater; these scenarios require specific bounding boxes to ensure that each crater is labeled correctly. Any error or deviation from these guidelines is not significant alone but at a large scale these repeated discrepancies can have an impact on the performance of the model.

Once all patches are labeled, the dataset of annotations can be exported as a YOLO training compatible set with the images. Exporting with the images makes it easier to train the model and divide up the dataset into training and validation subsets. The annotations are exported as a TXT file that provide the object class, coordinates, height & width of the bounding box. Figure 4.4 shows what the final product of an image that has been labeled using Label Studio would look like.

4.4 Dataset Creation

After image preprocessing, augmentation, and labeling, the final step is to assemble the structured dataset for supervised learning. This involves dividing up the dataset into 2 main categories. The first category is training which is the main process that allows the model to learn new features. The second is validation which is responsible for gauging the performance of the model. More specifically, validation makes predictions on each image in that category and compares those predictions to what it knows is true from the labeled image. For this project, the dataset was split into 80% training and 20 % validation, a common practice in computer vision to provide sufficient data for learning new features while reserving image samples for model evaluation.

It is important to ensure that the splits were randomized so that images were not clustered together in one category and not the other. This could create biases in training and validation, and is why consistency when labeling is so important to creating a high performing model. If images that are used for training are labeled differently than images used for validation, the model will make predictions that are not represented in the validation subset leading to inaccurate performance metrics.

Chapter 5 - Performance Metrics

5.1 Introduction

Evaluating the performance of a model is a crucial step in determining how effective and reliable a model is at performing computer vision tasks. Regardless of how advanced a model is, its performance metrics can be determined by how accurately it performs on unseen data. Performance metrics provide a quantitative way to assess how well a model detects and classifies specific features within an image. Analyzing these metrics can help make decisions about how to better train the model or improve the dataset.

In crater detection, performance evaluation is especially important because of the variability of the environmental conditions of the lunar surface and craters being analyzed. This means that the model must be able to identify craters of different sizes, shapes, illumination, and observation angles or rotations. This variability must be accounted for all while avoiding false positive detections that could be caused by rocks, shadows, or potentially high contrast surface textures. There are several performance metrics that are the go to when it comes to evaluating how accurate and precise a computer vision model can be. These standardized metrics are *precision*, *recall*, average precision (AP), and F1, with these metrics, the overall detection performance can be measured and compared.

Chapter 5 outlines each of these key performance metrics as well as the underlying theory and mathematics behind them such as the confusion matrix, and intersection over union (IOU). Each metric and the underlying theory will be explained more specifically in the context of lunar crater detection. With this knowledge in hand, these metrics are later applied in Chapter 6 to assess the performance of the trained model based on the datasets that are used.

5.2 Confusion Matrix

The confusion matrix is an important tool for evaluating the performance of a model, specifically in supervised learning tasks. It provides a straightforward and structured way of comparing what the model predicts to be a crater to what features are actually craters. This structured tool organizes the model's predictions and ground truth features into categories of correct and incorrect classifications. It becomes easier to understand not only how accurate the model is when making predictions on what it thinks are craters, but also where it tends to make mistakes in misclassifications. Figure 5.1 is an example of what a confusion matrix looks like and the four main categories that it uses to showcase model performance results.

		Actual	
		Positive	Negative
Predicted	Positive	TRUE POSITIVE	FALSE POSITIVE
	Negative	FALSE NEGATIVE	TRUE NEGATIVE

Figure 5.1 - Example of a confusion matrix

In crater detection, the confusion matrix helps identify how often the model can correctly detect a crater and when it incorrectly identifies other features as craters. As seen in Figure 5.1, the matrix is made up of four main categories, true positives (TPs), false positives (FPs), false negatives (FNs), and true negatives (TNs). TPs represent the intersection between an actual crater and the positive prediction that the model makes. TNs are the intersection between features that are not craters and the model correctly predicting that those features are not craters. FPs show how the model predicts the background as being a crater when in reality it is other features. FNs instead show how the model predicts an actual crater as not being a crater or another feature in the image.

These four categories help provide a complete picture of how the model is performing across different lunar environmental conditions and crater types. The confusion matrix provides crucial insight into the prediction capabilities of a model and can show any biases that could occur during model training. If the model produces a lot more false positives than true positives, this could mean that the model is identifying certain features as lunar craters. On the other hand, more false negatives could reveal that a model has difficulty identifying shallow craters or is often confused by similar lighting conditions that may look like craters. A confusion matrix coupled with data analysis could suggest that additional training data or better data augmentation may be necessary to improve accuracy and robustness.

In crater detection, interpreting the confusion matrix can be challenging because the distinction between positives and negatives is not always obvious. The lunar surface, as discussed previously, is a very diverse environment with variations in lighting, surface geography, and crater structure. These variations can lead to cases where the model's prediction might partially overlap a true crater or detect multiple bounding boxes around the same feature. In these scenarios, accuracy becomes just as important as placing the prediction versus actual craters in the right categories. A crater might be labeled correctly as a true positive, but if the bounding box does not align well with the crater's actual position, size, and shape, it may not be reliable enough for autonomous operations. Figure 5.2 shows an example of a misidentification and bounding box prediction error where one true crater is present. The model incorrectly labels two craters being present due to the lighting conditions and possibly the dataset that was used to train the model. To address these issues, one of the most important theoretical computer vision tools is used called Intersection over Union (IoU). IoU provides a quantitative way to measure the overlap between a predicted bounding box and its corresponding ground truth label made as part of the supervised learning process. More about IoU will be discussed later in Section 5.3.



Figure 5.2 - Low quality image of misidentification due to surface variation

In conclusion, the confusion matrix serves as the mathematical foundation for more important performance metrics that will be discussed later on in this chapter. Using the values stored in the confusion matrix, metrics such as *precision* and *recall* can be calculated, giving a more clear visual representation of model performance. A confusion matrix can help improve the performance of a model by giving a complete understanding of its ability to balance accurate detection with misidentification.

5.3 Intersection Over Union (IoU)

Intersection over Union (IoU) is another important theoretical metric used in object detection to help quantify the overall performance of a model. It is primarily used to evaluate how well a predicted bounding box aligns with the ground truth label. Unlike simple classification metrics, IoU focuses on the spatial accuracy of detections, or essentially how closely the model's predicted crater location and size match the true crater labeled during data labeling. IoU provides a single, intuitive value that quantifies this overlap and serves as a core measurement for determining whether a prediction should be considered correct or not.

Mathematically, IoU is defined as the ratio between the area of overlap and the area of union between the predicted and ground truth bounding boxes. The only resulting values that could come from calculating IoU are from zero to one. A value of one indicates a perfect overlap between the predicted and ground truth bounding boxes, while zero means there is no overlap at all. Figure 5.3 provides a good visual representation of how simple the mathematical expression for IoU is.

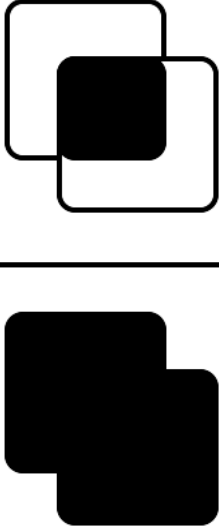
$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$


Figure 5.3 - Mathematical representation of Intersection over Union (IoU)

As mentioned previously, IoU is particularly useful due to the irregular and challenging visual nature of crater boundaries. Variations in crater illumination, shadowing, surface texture, and contrast, can throw off the bounding box made during prediction. These small deviations can significantly impact IoU values when compared to the ground truth labeling. A model that consistently scores high IoU values on validation demonstrates that a model is able to not only identify craters but can also accurately predict their bounding boxes.

When considering IoU alongside the confusion matrix, it creates an essential link between spatial accuracy and quantitatively categorizing these results. The confusion matrix only

tells whether predictions are right or wrong and does not consider the physical spatial accuracy of the predicted bounding boxes. IoU values make it possible to gauge what counts as a TP, TN, FP, FN in the confusion matrix. Without these values, a confusion matrix may place a prediction incorrectly into one of these four categories by over- or under-estimating a model's accuracy. The confusion matrix and IoU together form the foundation of model evaluation by providing an easily understandable representation of detection accuracy and precision. Both metrics make it possible to calculate the *confidence* and *precision* curves which then can be used to further calculate the all important average precision (AP) metric.

While there are many performance metrics, IoU provides a spatial measure of how accurately a model predicts a bounding box relative to the ground truth label. Specifically for crater detection, it plays a vital role in showing how discrepancies in model training or labeling can affect the overall performance. Coupled with other metrics, IoU is a powerful tool that provides the basis for more advanced object detection metrics as training becomes more complex.

5.4 Recall- and Precision-Confidence Curves

In object detection, *confidence* curves are essential for analyzing how the performance of a model can change over the course of varying detection *confidence* thresholds. The *confidence* score represents how likely the model thinks that a specific class, in this case a crater, is within the bounding box it has identified. This is different from IoU because that represents a spatial accuracy metric while *confidence* is the model's internal certainty of an object of interest being present. These curves can help visualize how *precision* and *recall* change as the model makes predictions with higher *confidence* scores. It also makes it possible to compare various versions of the model as the training is iterated with improvements to the dataset. Insights into the model's stability, reliability, and overall behavior under different thresholds has many more benefits to model comprehension.

Precision and *recall* themselves are important to understand as fundamental metrics based off of the categories defined in the confusion matrix. *Precision* is the measure of how many of the positive detections the model makes are actually correct. It is calculated using equation (5.1) which takes the TPs and FPs from the confusion matrix.

$$Precision = \frac{TP}{TP + FP} \quad (5.1)$$

As previously discussed, TP represents the correctly identified craters, and FP represents incorrect detections where the model labels a crater that is not actually there. Having high *precision* means that most detections are accurate and the model makes few false predictions.

Recall measures how well a model captures all of the correct objects within an image. This metric gives an indication of how well the model is at identifying positive detections. Equation (5.2) shows how *recall* is calculated using the TPs and FNs also from the confusion matrix.

$$Recall = \frac{TP}{TP + FN} \quad (5.2)$$

FN represents the number of craters the model failed to detect. High *recall* means that the model successfully identifies most craters, whereas low *recall* means the model is uncertain or less accurate, often misidentifying craters as the background.

A *precision-confidence* curve shows how *precision* changes with increasing *confidence*. At low *confidence* values, typical models accept nearly all detections resulting in a large number of false positives or repeat detections. As the *confidence* values increase, the model becomes more selective and only accepts detections that it believes are more likely craters than not, in other words, highly confident detections. At these higher *confidence* levels, the model has an increase in *precision* but makes fewer total detections as it becomes very particular about selecting craters. FPs can often occur from shadows, rocks, or uneven terrain that can often resemble craters making it more challenging to train a crater detection model to a high level of *precision*. This is why *precision-confidence* curves are so important in the way they relay information about the models detection capabilities to help guide further improvements. In any case, a high *precision* at high *confidence* levels indicates that the model is able to effectively distinguish between true craters and other features or background noise.

A *recall-confidence* curve, similar to a *precision-confidence* curve, shows how *recall* changes with increasing *confidence*. At low *confidence* values, *recall* is typically higher due to the model accepting almost all detections, even detections that the model is not completely certain about. When *confidence* starts to rise, crater detections that the model wasn't so confident about begin to disappear causing *recall* to drop as some true crater detections are filtered out. FNs typically occur due to poor model training and generalization since the model will be uncertain about exactly what is a crater and what is not. These uncertainties lead to the accidental misidentification of craters as background noise or other features.

Together, *recall-* and *precision-confidence* curves can provide a deeper, more visual understanding of a model's detection characteristics across different *confidence* values. A perfect model could maintain both high *precision* and *recall* across the entire range of *confidence* levels. This would demonstrate that the model can make accurate crater detections regardless of what the *confidence* level is. In an ideal world this would be the case but ultimately in practice, models show an inverse relationship between these two metrics. In many cases, improving one curve

may lead to a reduction of performance in the other so a balance is needed to maximize overall detection performance.

Recall- and *precision-confidence* curves serve as an important tool that form the foundation of and work with other evaluation methods. *Confidence* curves show the model's performance continuously across all *confidence* levels while confusion matrices provide a single frame of performance. They are also foundational for computing higher level metrics such as AP and the F1 score, both of which utilize *precision* and *recall* across the *confidence* spectrum to summarize overall crater detection performance.

5.5 Average Precision (AP)

Average precision (AP) is another widely used metric in object detection tasks because it can provide a single value to showcase performance. This single value reflects the accuracy and entirety of the predictions that the model makes on unlabeled images. As was previously stated, AP essentially summarizes the *recall* and *precision* of a model across all *confidence* levels. Previous metrics provide their own insights into model performance but are ultimately the foundation for AP. The main benefit of AP is that it allows for comparing models and how the accuracy of a model may improve across different training dataset iterations.

AP is calculated from the *precision-recall* curve which plots *precision* against *recall* at different *confidence* levels. The *precision-recall* curve is good for seeing the relationship between these two metrics and how they may change with each model version. The area under this curve, or the integration, represents the model's AP. If this curve has a large area, it can mean that the model maintains both high *precision* and *recall*, while a smaller area suggests that there are some trade offs happening in model training between the two. These trade offs could be the result of training on the images that the model is being validated on not accurately representing the dataset.

$$AP = \sum_n (R_n - R_{n-1}) \max_{R \geq R_n} (P(\underline{R})) \quad (5.3)$$

Equation (5.3) represents the mathematical formula for the area under the *precision-recall* curve. This equation uses interpolated *precision* for each *recall* level to get the highest *precision* achieved for any *recall* greater than or equal to that *recall* level. This interpolated *precision* is multiplied by the difference between the current *recall* and previous *recall* value. The summation across all *recall* levels gives the total area under that curve, which is AP.

AP is a particularly reliable way to see the overall quality of the model in crater detection. Some craters may be naturally more difficult to detect than others due to contrast or shadowing, resulting in lower *confidence* levels. A model with high AP is more suited to handle

this while still maintaining accurate detections for more distinct craters. AP depends on IoU since AP is reported at a fixed IoU value, typically 0.5. This means that a predicted crater is correct if its bounding box overlaps the ground truth by at least 50%. Similar to AP, multiple IoU values may be averaged, this is the basis for the metric known as mean average precision (mAP). This mAP can showcase a more balanced view of the model's performance through its accuracy across various object classes.

AP serves as a powerful analytical tool for summarizing a model's detection abilities. It presents both the accuracy of crater detection and reliability of a model to make these accurate detections across the *confidence* spectrum. With AP, model performance can be more easily assessed across different dataset versions and model training allowing for a comprehensive comparison between model iterations. This means that both the accuracy and precision are properly considered when looking to make improvements to performance. In other words, AP is an ideal metric for assessing and improving a crater detection model to achieve the highest detection capabilities possible.

5.6 F1 Score

The F1 score is another metric that provides a single value to quantify how balanced the performance is between *precision* and *recall*. Like other metrics, this value can give a comprehensive indication of a model's accuracy and precision in identifying lunar craters. F1 is similar to AP in the sense that it looks at previous metrics to provide valuable insight into performance. The main difference is that F1 does not look at the area under the *precision-recall* curve but instead looks at the individual values themselves at given *confidence* levels.

$$F1 = 2 \frac{Precision \cdot Recall}{Precision + Recall} \quad (5.4)$$

Equation (5.4) is the mathematical expression for F1 which takes the *precision* and *recall* for a given *confidence* level. In crater detection applications, there are often more true negatives than true positives due to the vastly dominant background. This is important because F1 score is particularly effective in conveying the performance of the model in such cases. It does this by equally weighing the importance of *precision*, which avoids false crater detections, and *recall*, which works to detect all true craters.

High F1 score shows that the model achieves a balance between identifying craters and minimizing incorrect detection, much like other metrics. Often, showing this balance means that the model is able to maintain both strong *precision* and *recall* which is the most optimal outcome. Low F1 score shows that the model is either missing true craters in the form of low *recall* or falsely identifying features such as rocks, shadows, or the background as craters also in the form of low *precision*.

The F1 score can be analyzed as a function of the *confidence* level, which shows the difference between higher or lower *precision* and *recall*. This metric is most effective when looking at changes to model detections or inference across training iterations and datasets. Plotting F1 against *confidence* helps to visually determine an optimal point where the model can achieve the best balance when detecting craters. F1, when combined with many other metrics discussed previously, can help provide a comprehensive picture of model performance and help guide further training for improvements.

5.7 Summary

This chapter outlines the main key performance metrics that are used to evaluate how effective a model is in detecting lunar craters. Foundational metrics such as IoU and confusion matrices make it possible to categorize different bounding box overlapping. *Precision* and *recall* curves form the basis for training iteration showing how these metrics change as *confidence* increases. From these curves, more advanced metrics such as AP and F1 can be determined to provide further insight into performance, model improvement, and training capabilities. Together, all of these metrics create a complete framework for evaluating a crater detection model. These theoretical metrics are applied to real experimental results to assess how an object detection model performs on lunar images.

Chapter 6 - Results

6.1 Introduction

This chapter presents the results from different versions of a YOLO model that are trained for crater detection and classification. The goal is to show how each model version performed using the metrics that were discussed in Chapter 5, and how those results changed as the dataset became more refined over time. The main focus will be showing the actual performance results and showcase examples of predictions made by the models.

Each model version was trained using a more refined and larger dataset while the model architecture and augmentation techniques were unchanged. These changes only in the dataset help showcase how the results are affected and whether these changes make the model better, worse, or did not have much effect at all. The performance is measured using previously mentioned metrics such as *precision*, *recall*, AP, and F1 to give a clear picture of how well the model is detecting craters. Some examples of how the models performed on lunar surface images, both good detections and cases where it made mistakes, are presented to give a better visual understanding of performance.

Going through these results gives a clear understanding of how the model improved from version 1 to version 5. The last part also includes a comparison between the YOLO Nano and YOLO Small models to see how changing the model architecture may affect both accuracy and speed. There are still questions that are left unanswered and analysis still needs to be done to get a full view of the capabilities an object detection model can have for crater detection during lunar operations.

6.2 Experimental Setup

For YOLO model training, there are several experimental conditions that were used to ensure that the results shown later are consistent. All model training and testing were done using the same general environment and conditions. This means any changes in performance came from the refined dataset or model configuration, not differences in the hardware or testing images.

6.2.1 Hardware Configuration

All model training, testing, and other related software tasks were done on a personal desktop computer. This computer is equipped with an NVIDIA GeForce RTX 3060 Ti GPU, an AMD Ryzen 7 5800X CPU, and 32 GB of RAM. The GPU was the main component used for accelerating the training process, allowing the YOLO models to handle large batches of images

and videos to test on. The 3060 Ti provides enough VRAM to train at an image resolution of 640x640 and supports the usage of NVIDIA's parallel processing capabilities through the Compute Unified Device Architecture, or CUDA, platform. CUDA unlocks the GPU's cores to perform mass parallel processing which YOLO can utilize. Computer vision tasks often involves large linear algebraic operations, particularly when performing convolutions.

The Ryzen 5800X is a powerful CPU which made sure that the hardware did not bottleneck the GPU during training or inferencing operations. The CPU's main tasks were to handle data loading, preprocessing, and parallelized image operations such as the patch generation and augmentations. These tasks are not as computationally intense as model training but are important components to the overall model development process. The system RAM was more than enough to store the dataset, load sequences of image patches, and keep multiple training processes running smoothly. Sufficient RAM also helped when converting IMG files, generating augmentations, and running the Label Studio software.

All software tasks were run locally without the use of external cloud computing or external hardware that some large AI tasks may require nowadays. Due to the relatively low number of parameters that these particular YOLO models have, the hardware setup can provide a consistent performance environment suitable for this project. Of course, this is hardware without limitations such as power consumption, mass, and thermal characteristics which would need to be taken into consideration if deployed on a real world space system.

6.2.2 Software Configuration

The software for this project was built around Python 3.12.4 which provided a modern foundation for implementing a crater detection model. Python has a plethora of supported libraries that provide a good framework for model development, dataset preprocessing, and metric analysis.

Libraries used include NumPy, and Matplotlib for data processing and image visualization. Other libraries used are Random and OS to help manage file handling, organizing patch generation into various directories, and randomizing image augmentation. Image augmentation and manipulation were mainly done using the Python Imaging Library (PIL), and OpenCV packages which provided the capability for patch extraction, brightness variation, and rotation transformations. For geospatial data handling, a data type that is represented by the IMG raw lunar files, this project used OSGEO (GDAL), which enabled reading and transforming those large lunar files into a more accessible format.

The main packages used for model development and training were done using PyTorch which allows for the use of CUDA. The Ultralytics package was the main library that made it

possible to implement a pre-trained YOLOv11 model which was built on top of PyTorch. All coding took place in the JetBrains PyCharm IDE, which helped manage the Python environment, debugging, and Git version control.

6.2.3 Model Training and Inferencing Configuration

To be consistent across all model versions, all training parameters and settings were kept the same to ensure no difference in performance. Each model was trained using the Ultralytics YOLOv11 Nano model which has weights from pretraining rather than starting from scratch. Using pretrained weights means the model has a strong foundation and sufficient low level feature extraction abilities. This foundation helps speed up training and improves performance on smaller datasets like the one used for this project. For version 5, the same parameters and settings were used for the YOLOv11 Small model when compared to the Nano model.

The models were trained using an image size of 640x640, which is the same as the resolution of the processed lunar patches discussed in Chapter 4. For each training run, the model was trained for 100 epochs, meaning the entire dataset was passed through the neural network 100 times. The batch size was set at 8 meaning that 8 images were loaded and processed by the neural network in a forward and backward pass before the model updated its weights. As previously discussed, the training was done using the GPU, or device 0. No layers in the model were manually frozen, and the base model was not altered.



Figure 6.1 - Lunar patch used for all basic inference testing

For inference testing, all labels and *confidence* scores on each prediction were shown in the final image. In order to reduce the clutter of classifications, a *confidence* minimum was set at 0.5, this means that any prediction with a *confidence* less than 0.5 will not show in the image. Figure 6.1 is the image that was used on all image inference testing to keep visual performance comparison consistent across different versions. Appendix C is the main Python script that is used when training each model version and running inference tests to verify performance. Appendix D is simply the YAML configuration file that holds the directories and project description.

6.3 Model Version Overview

For this project, five separate models were trained from the base YOLOv11 model. This section provides an overview of the five different model versions only looking at the differences between each and not analyzing performance. Each version has improvements in the dataset rather than in the model architecture to provide a consistent and simplistic analysis of performance. Dataset refinement for this project typically involves increasing the number of training images, improving label quality, and expanding crater diversity. This makes it possible to clearly see how dataset modifications alone influence performance.

Across all five versions, the same YOLOv11 Nano base model was used to train on, except in version 5. For version 5, a comparison between the Nano and Small YOLOv11 models was performed along with improvements to the dataset. The reason for this comparison was to identify if using a larger model with more parameters would provide meaningful benefits to crater detection performance. Each version was trained using the same parameters and evaluated using the same performance metrics discussed in Chapter 5. Each subsection provides a brief summary of each version, the dataset changes introduced, and the motivation behind each update. Table 6.1 shows the progress from version 1 to version 5 of the model with the changes in the dataset and architecture for the last version.

Table 6.1 - All model versions with changes to dataset

Mode Version	Dataset Size	Model Architecture	Dataset Change
v1	20	YOLOv11 Nano	Initial dataset baseline
v2	50	YOLOv11 Nano	Added 30 images
v3	50	YOLOv11 Nano	Updated initial 20 images (v1)
v4	80	YOLOv11 Nano	Added 30 images
v5	110	YOLOv11 Nano & Small	Added 30 images

6.3.1 Version 1

For version 1, a basic baseline dataset was used to train the model to get an initial evaluation of the performance. The dataset for this version consisted of a relatively small number of lunar images with manually created labels. These labels were sufficient for initial testing but contained inconsistencies in the crater bounding box locations and size. With a low number of images used, the variation in the environment was limited in crater size, lighting conditions, and background geography. The goal for version 1 was to get a working model to establish the benchmark against which future models could be compared.

Version 1, despite its limitations given the dataset, gave valuable insight into how well the model would perform. The model would miss small craters, incorrectly identify rocks and shadows as craters, and produce bounding boxes that were not consistent or precise. A lot of these issues were addressed from this initial testing to identify areas where the dataset could be improved. Table 6.1 shows the details of version 1, which started with 20 labeled lunar images to train on using YOLOv11 Nano. The specific performance will be looked into for this version and subsequent versions later on in this chapter.

6.3.2 Version 2

For version 2, the dataset was expanded by adding additional lunar images from the similar lunar region used in version 1. Approximately 30 labeled images were added on top of the initial 20 images from version 1. This increase in sample size introduced more craters, more terrain variations, and lightning conditions. The label quality was also increased to account for the large number of small craters that were typically missed in version 1.

By increasing the dataset, the intention was to give the model more data points to look at to help it better discern from the background and the craters. The original 20 images were not reviewed or altered in any way to provide a consistent increase in performance. Overall, model version 2 had a better generalization of craters and reduced some issues with overfitting seen in version 1. Table 6.1 highlights the total dataset size for version 2 was 50 images while still using YOLOv11 Nano.

6.3.3 Version 3

Model version 3 did not add any new lunar images to the dataset but rather focused more on quality over quantity to see if performance could be increased. Instead, the original 20 images from version 1 were updated by going back and refining the labels. These label refinements included making bounding boxes more consistent and readjusting labels already made to fit around craters better. A big emphasis was put on adding more crater variation by including smaller craters to improve the feature capture capabilities at lower *confidence*. Including smaller craters added more terrain variability making the model more robust and better able to generalize beyond the conditions that were seen in the previous two versions. These changes are briefly highlighted in Table 6.1, as version 3 simply updates the initial version 1 images without changing any other parameters.

6.3.4 Version 4

Version 4 brought additional labeled lunar images to train on. Approximately 30 more images were added to the dataset making for a total image count of 80 images for training. The

labeling quality was kept as consistent as possible to match all the images from previous versions. This is important as changes in the dataset labeling quality can skew the training and thus the performance of the model in ways that could compound as errors pile up. Table 6.1 helps visualize these changes from version to version and shows that version has a larger dataset with similar parameters.

6.3.5 Version 5

Version 5 again brought changes in the form of additional images added to the already 80 image dataset. 30 additional labeled lunar images were included to improve the image diversity across craters, lightning, and the background. While this dataset was not sufficiently large, it still showcases a performance increase to be discussed further later. A larger and larger dataset would be more preferred to help the model truly discern from small and large craters to the background but due to time constraints and manpower, the dataset could not be expanded more.

For version 5, a comparison between the YOLOv11 Nano and YOLOv11 Small was looked at to determine if a slightly larger model would be more sufficient. Essentially, this provided the opportunity to evaluate if more parameters and deeper feature extraction capabilities could significantly improve performance.

```
YOLO11n summary: 181 layers, 2,624,080 parameters, 0 gradients, 6.6 GFLOPs  
YOLO11s summary: 181 layers, 9,458,752 parameters, 0 gradients, 21.7 GFLOPs
```

Figure 6.2 - Model description between Nano and Small

Looking at Figure 6.2, the output on the command line shows the main difference between the two base YOLOv11 model types. Nano boasts significantly less parameters and computing power required by the model versus Small. Computing power, in this particular case, is measured in GFLOPS or Giga Floating-point Operations Per Second. This makes sense as the larger model, Small with over 3.5x the number of parameters as Nano, requires more GFLOPS to process more calculations. While parameters and computer power changes, there is no increase in the number of layers that the model has.

It is important to mention the YOLOv11 model architecture that is kept the same for all versions trained for this project. Looking at Appendix E, which shows the exact YOLOv11 architecture that all model versions have. Chapter 3 talked about the specific layers found within the YOLOv11 model. Looking at the hard coded Python YAML file that actually contains the framework for the model helps provide context and understanding for this project.

6.4 Performance Analysis

This section summarizes the performance results obtained from all five model versions briefly mentioned in section 6.3. These key performance results obtained from all versions are the *precision-confidence*, *recall-confidence*, *F1-confidence*, and *precision-recall* curves. These curves were generated automatically by Ultralytics after model training and validation. Inference testing was also performed on the same data for each model version to show how the model would label a real world lunar image.

Performance analysis is a very important part of evaluating how a model can improve over time with new training data. The results from this project provide a chronological view of how the model evolved with each dataset refinement from version to version. Starting with version 1 that was trained on 20 labeled images to version 5 which saw over a 5x increase in training dataset size to 110 images. In between are three versions that help to create performance checkpoints to evaluate if training is going in the right direction.

6.4.1 Precision-Confidence

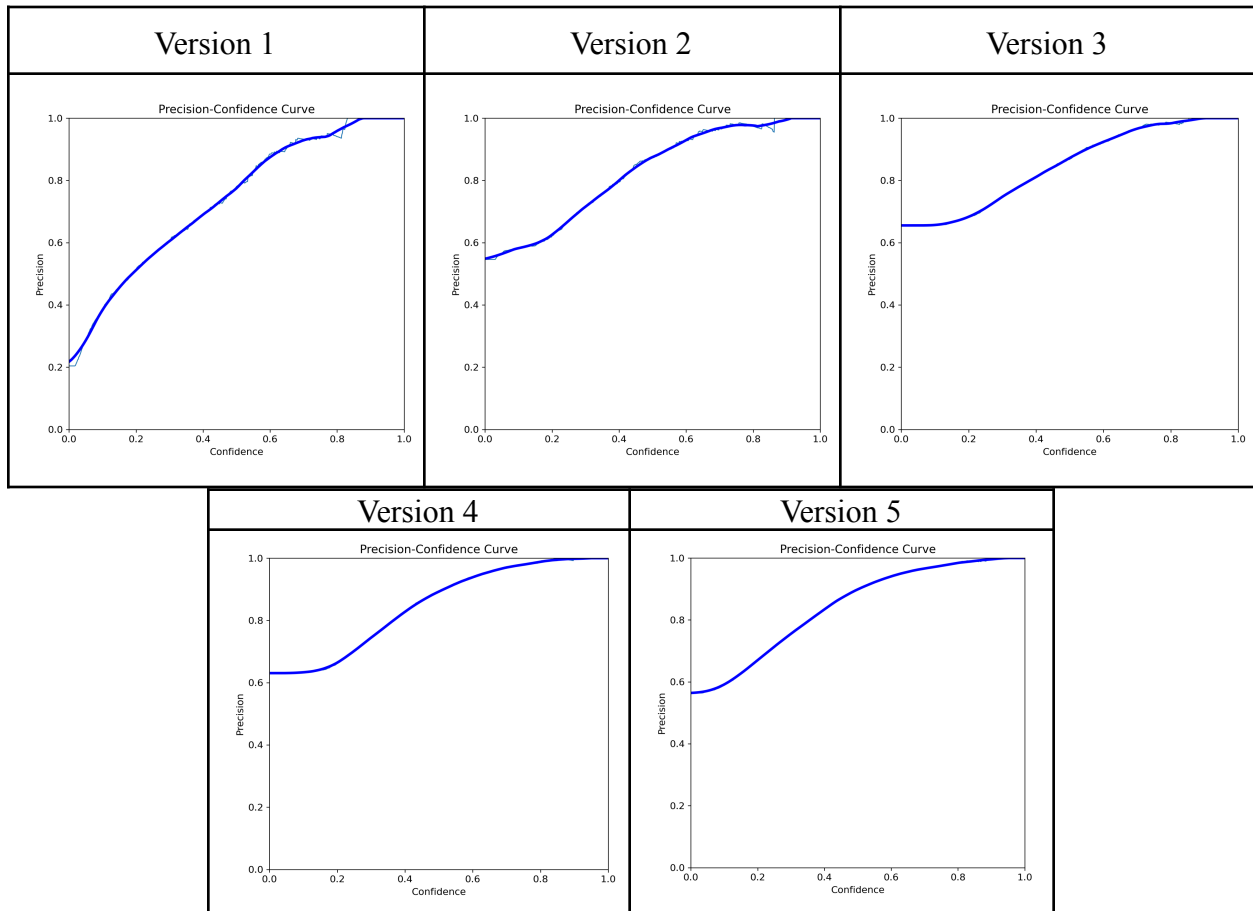


Figure 6.3 - Precision-confidence curve performance through each model version

As discussed in Chapter 5, the *precision-confidence* curve is an important metric that highlights how precise the model is at increasing *confidence* levels. Figure 6.4 shows the *precision-confidence* curves for all 5 model versions as the dataset got more refined. If the model can maintain relatively high *precision* at both low and high *confidence* levels, these curves will showcase that. Just looking at the progress initially, it shows a clear improvement in stability, smoothness, and maximum *precision* as more training data is introduced.

Some important observations are with regard to the *precision* values at low *confidence* levels. In version 1, a steep increase in *precision* indicates that the model is making a lot of mistakes early on when it is not so confident whether something is a crater or not. This behaviour is expected given how small the initial dataset for this version is. This smaller dataset restricts the model's ability to generalize crater features reliably. As the dataset size increases, this is not the case. Version 2, which only added 30 more images, has a very noticeable increase in the initial *precision* at low *confidence*. Essentially going from around 0.2 for version 1 up to almost 0.55 for version 2 which highlights a 2.5x increase over the previous version.

From version 3 to 5, there is further refinement of the dataset leading to a more gradual and smooth increase in *precision* overall. However, there is a slight reduction in *precision* at low *confidence* for version 4 and 5. Like was mentioned previously, this reduction is due to the inverse relationship with *recall*. In the next subsection, there is an increase in the *recall* of the model that can be correlated with this decrease in *precision*.

Looking at the overall shape however, there is a noticeable curve that is taking form with version 4 and 5 almost looking identically smooth and having a similar shape. This curve smoothness in later versions can be attributed to the model learning the patterns of lunar craters that it has been trained on more effectively. The reduction in noise also indicates that the model is producing fewer false positives than it initially did during version 1.

Overall, the progression from version 1 to version 5 shows a clear correlation between the dataset size and label quality with the model stability. Each successive version reduced model noise in the *precision-confidence* curve and improved maximum *precision* at higher *confidence*. These plots validate that increasing the dataset improved the model's *confidence* calibration and reduces false positives.

6.4.2 Recall-Confidence

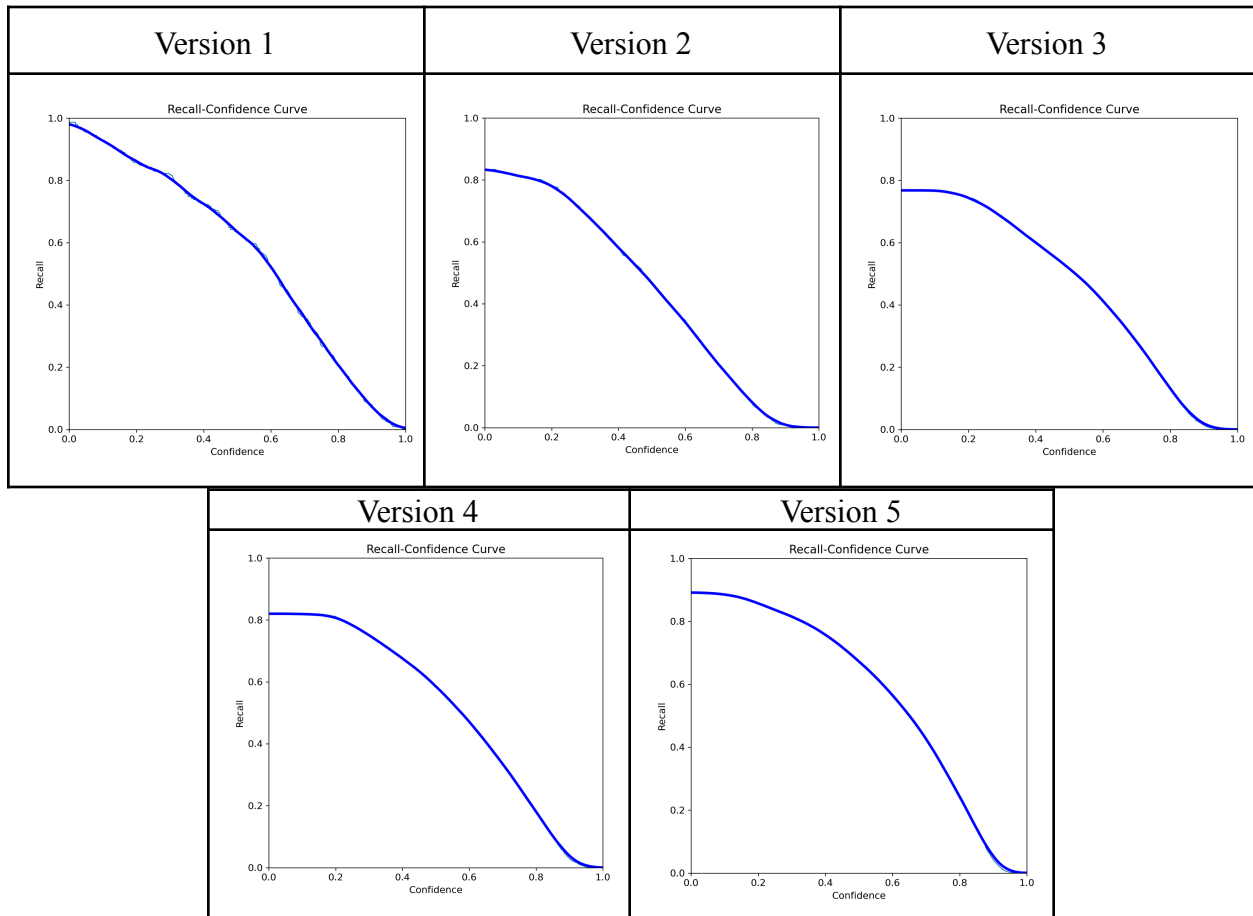


Figure 6.4 - Recall-confidence curve performance through each model version

Recall-confidence is another important metric that can show how effective a model is at identifying all craters correctly. In other words, these curves provide insight into a model's ability to avoid missed detections as the dataset size increases. Figure 6.5 shows how the model progressed in *recall-confidence* performance with each successive version. It is important to remember that *recall* is an inverse relation with *precision*, so oftentimes an improvement in *precision* will lead to a reduction in *recall* or vice versa. Initially, as the dataset is improved there is a decrease in *recall* at lower *confidence* while *recall* at higher *confidence* effectively remains unchanged.

Looking at version 1, the *recall-confidence* curve takes on an almost negative linear pattern as *confidence* increases. Along with this, there is a lot of noise that is created by the uncertainty present when the model is validating its performance. This is most likely due to the limited dataset that was used to train version 1, with no large variability present in lighting, crater shape, and background geography. The model frequently misses smaller craters or craters that

have a low *confidence* score given from the classification. These smaller craters are more confused with the background lunar surface than larger craters.

While this noise is present early on, it is quickly reduced with the introduction of more training data. Version 2 introduces a curve that is more smoother and starts to take on a particular shape with the reduction in *recall* at lower *confidence*. A lower *recall* can be associated with a performance decrease since this means that there are more false negatives being flagged. Versions 3 to 5 show what will be the final shape of the *recall-confidence* curve indicating that the model is stabilizing and reaching some convergence. This stabilization is natural as introducing more training data will only increase performance so much.

One important thing to note is that the *recall* is high at lower *confidence* but almost zero at higher *confidence*. While this may seem incorrect, what it really means is that the model is being more picky and filtering out what it thinks are bad detections in the form of labeling them as false negatives. This is the opposite for low *confidence* where the model accepts more detections as being true leading to a high *recall* or low false negatives.

Overall, the *recall-confidence* performance over the course of the five model versions shows an increase in stability, reduction in noise, and a more gradual decline as the dataset increases. While there is an increase in false negatives, or decrease in *recall*, the model sees an increase in *precision* noted from the previous subsection. This trend complements the improvements observed in the *precision-confidence* curves and supports the conclusion that the model's performance improves consistently with additional training data.

6.4.3 F1-Confidence

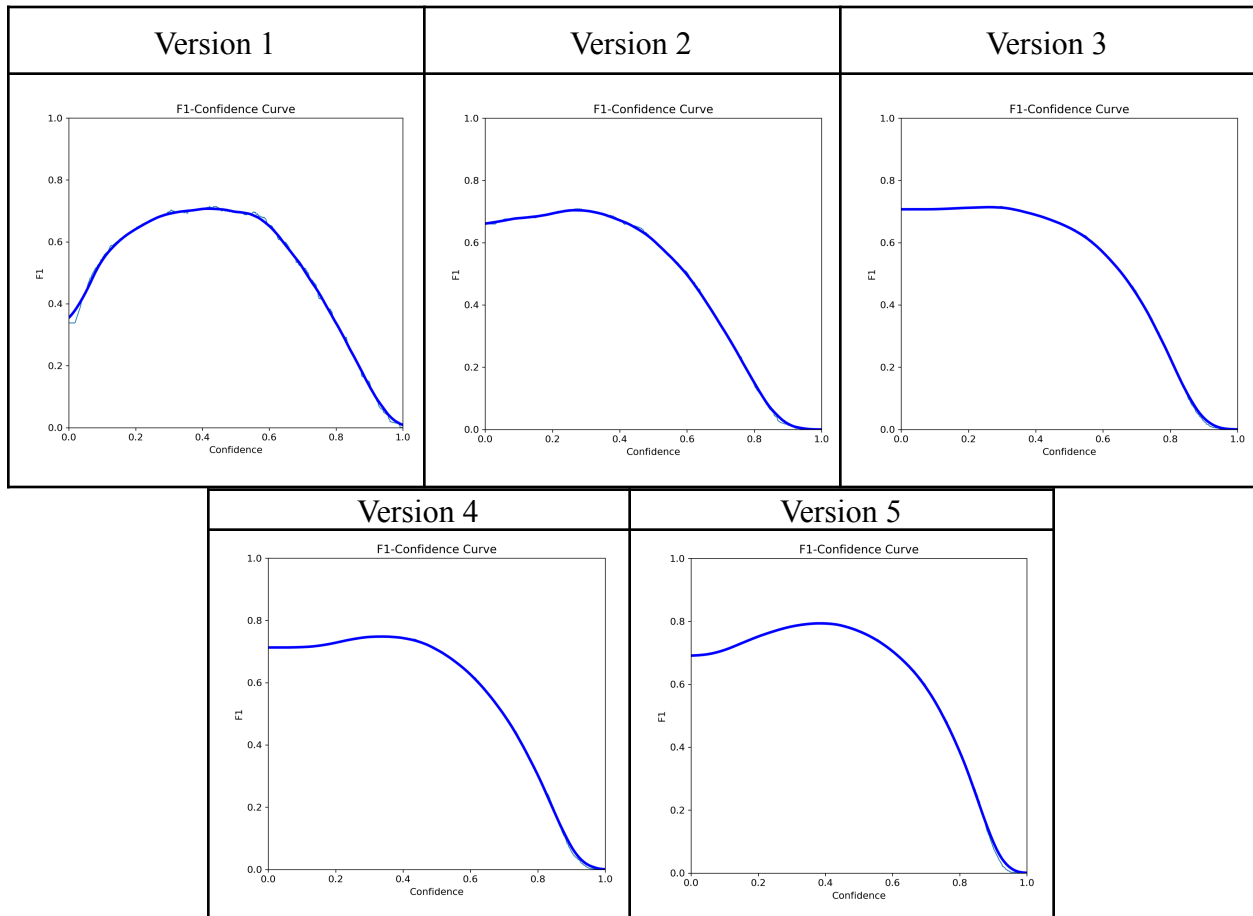


Figure 6.5 - F1-confidence curve performance through each model version

Looking at F1-confidence, discussed previously, which is just the relationship between *precision* and *recall* validation measurements. This is another important metric that can show the overall performance increase of the model as the dataset changes. Figure 6.6 shows this performance increase as the dataset was more refined from version to version. Like previous plots, the F1-confidence will eventually converge to some specific curve indicating the performance ceiling using the current labeling and image techniques. The initial evaluation of these plots show that performance is stagnant at high *confidence* but sees improvements at low *confidence*.

Version 1 has a notable low F1 score at low *confidence*, this is associated with the lower *precision* score at the same *confidence* levels. Again there is a lot of noise initially as the model is very uncertain about its predictions when compared with the validating labels. From version 1 to 2, the score considerably increases from around 0.35 to 0.65 due to the large *precision* increase with the addition of more training data. At higher *confidence*, the model has a drastic reduction in F1 indicating that it is overly conservative and misses many true craters when at

higher *confidence* detection levels. Versions 3 to 5 have very minor increases in performance other than with the stability of the model and noise reduction. These improvements again reflect the model calibration, better generalization, and more reliable detection of craters across all *confidence* levels, sizes, and lighting conditions.

Overall, the evolution of the F1-*confidence* curve from version 1 to version 5 continues to demonstrate that the model will have an increase in performance with a larger dataset. It is important to point out that the dataset also received improvements in the form of labeling quality so this performance increase can also be associated with that. The ability for the model to make strong predictions by balancing *precision* and *recall* is important to consider if further improvements to model performance are to be made.

6.4.4 Precision-Recall

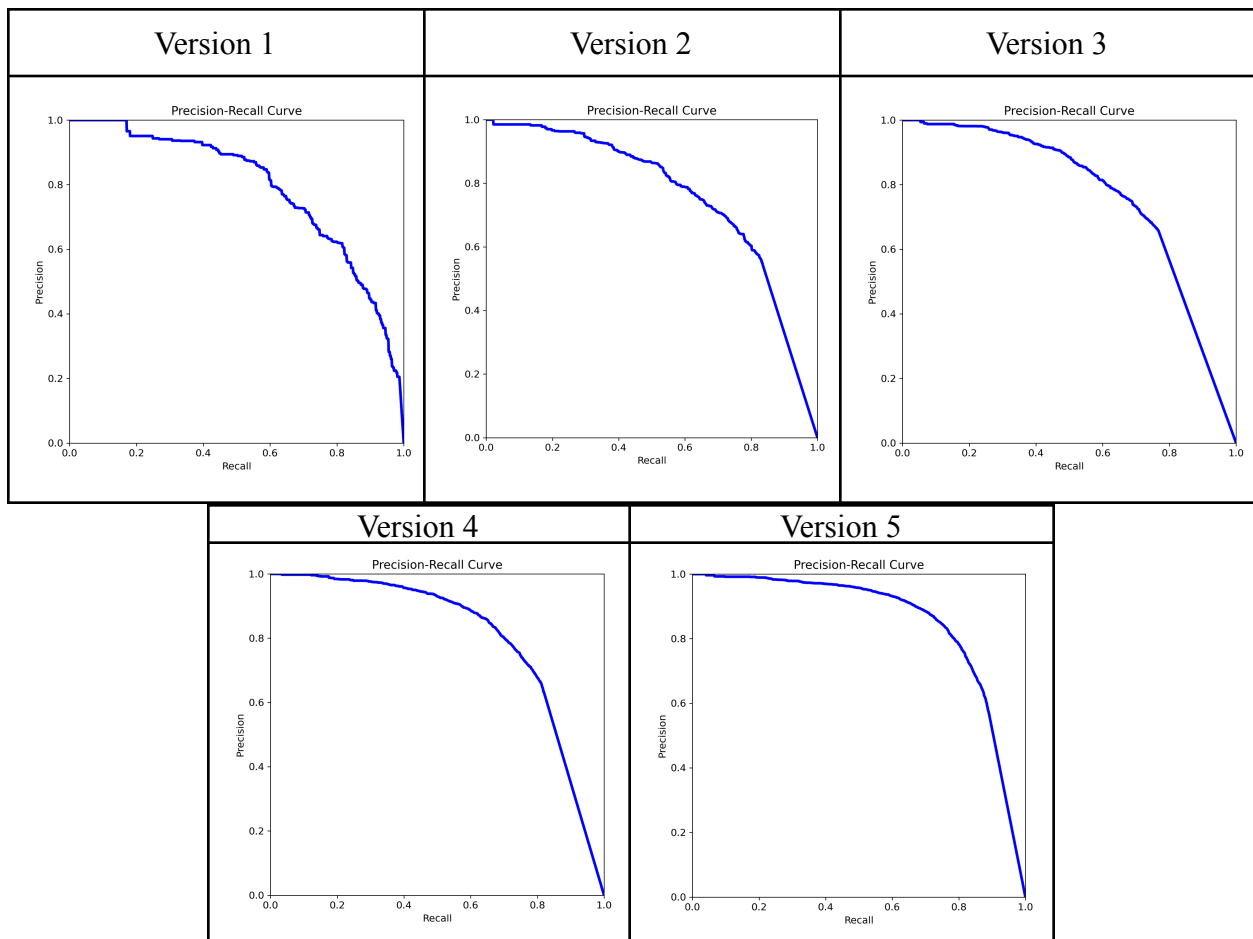


Figure 6.6 - *Precision-recall* curve performance through each model version

The *precision-recall* curve was not explicitly discussed in Chapter 5 but is a good metric for looking at the exact relationship between *precision* and *recall*. Instead of comparing

confidence with either metric, these values are stacked against each other to show their true relationship. Figure 6.7 highlights the curves generated by Ultralytics during each training iteration. What is initially noticed is that consistent curve that is present in all versions of the model indicating the trade off between the two metrics.

The noise from both *precision* and *recall* makes it more prevalent in these curves as can be seen in version 1. Performance at low *confidence* is strong with lower *recall* values maintaining relatively high *precision* near 1.0. However, as the recall begins to increase, the *precision* drops off dramatically indicating the model begins to produce a lot of false positives when attempting to capture more true positive detections.

From version 2 to 5, there is essentially the same trend albeit with a sharper more profound drop as the *recall* is increased. The noise is drastically reduced but is still present even slightly in version 5 of the model but there is still an increase in stability. The most balance between *precision* and *recall* is achieved with version 5 with *precision* remaining high across a lot of the *recall* spectrum. This means a reduction in false negatives while maintaining strong control over false positives.

In conclusion, the model has significant improvements in performance when given high quality labels and more data. This increase in performance is important in applications such as crater detection, where missing a crater could have a significant impact on mission success or performance. While version 5 is not up to the standard of what would be expected of human piloted vehicles, it is still a drastic improvement from version 1. These improvements show a trajectory of increasingly reliable detections that if given more time and resources could potentially meet real world reliability standards. It is important to note that this is simply improving the dataset, further customizations could be made to the model architecture that could lead to further improvements in *precision*, accuracy, and *recall*.

6.5 Model Inferencing Testing

Quantitative performance metrics like the ones described in the previous section are important for assessing a model. However, these metrics do not show how the model would perform on a specific test scenario or lunar image that it has not seen before. This section looks at this scenario by showing each model version the same lunar image to test on. Using the same image to test on is important because it means the results can be directly compared with one another to see if the model improves its detection capabilities. Furthermore, this enables a direct visual assessment of crater accuracy, false positives, and the model's ability to generalize on unseen surface conditions. The image chosen for this assessment was chosen at random to ensure that no particular human bias could influence the results. This patch has had data augmentation performed on it like every other image that the model has been trained on, with a random

rotation and brightness adjustment. As was previously mentioned, the lunar patch used for testing is Figure 6.1 and a *confidence* minimum of 0.5 was chosen to avoid cluttered labeling.

For each model version, the results from the inference test were recorded and compared side by side. This allows for a clear visual comparison between each model and how the dataset refinements changed the outcome.

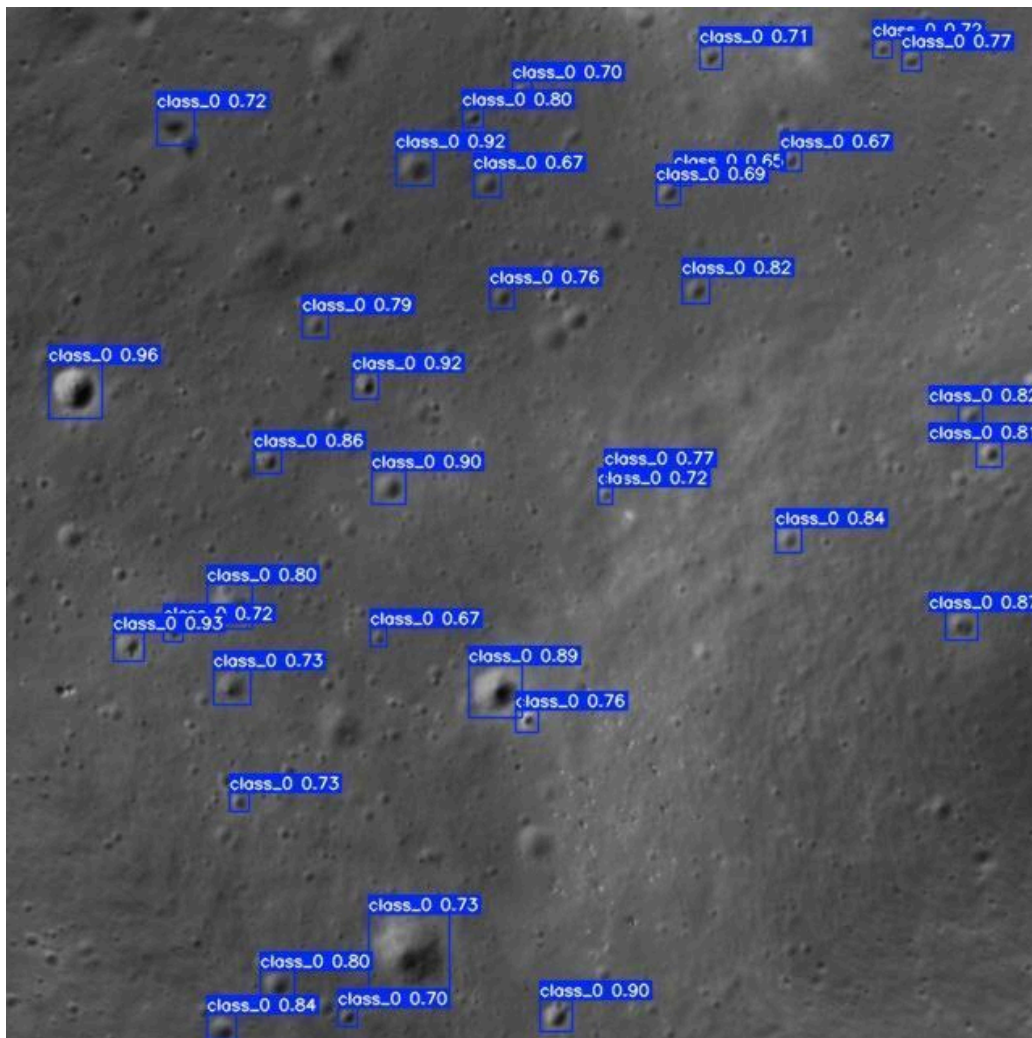


Figure 6.7 - Model version 1 inference test (*confidence* minimum of 0.5)

Looking at Figure 6.7, it is clear to see that the first version of the model did not perform very well. There was still a lot of ambiguity when it came to what craters it would detect and classify correctly. Detection is very random and there are a lot of missed craters and occasional false positives. The model demonstrates limited ability to differentiate small crater boundaries from local shadowed regions that may appear like a crater. The bounding boxes the model has put around identified craters are also not consistent. Some of these boxes are not exact meaning it is either not precisely located at the crater or has spacing that should not be present. These

observations played an important role when it comes to improving the dataset to improve overall model performance. Some examples of improvements from this particular outcome is the need for more examples of craters with better bounding boxes. Overall, the results for model version 1 were promising as they provided an early glimpse into what was possible given the small size of the dataset but more improvements need to be made.

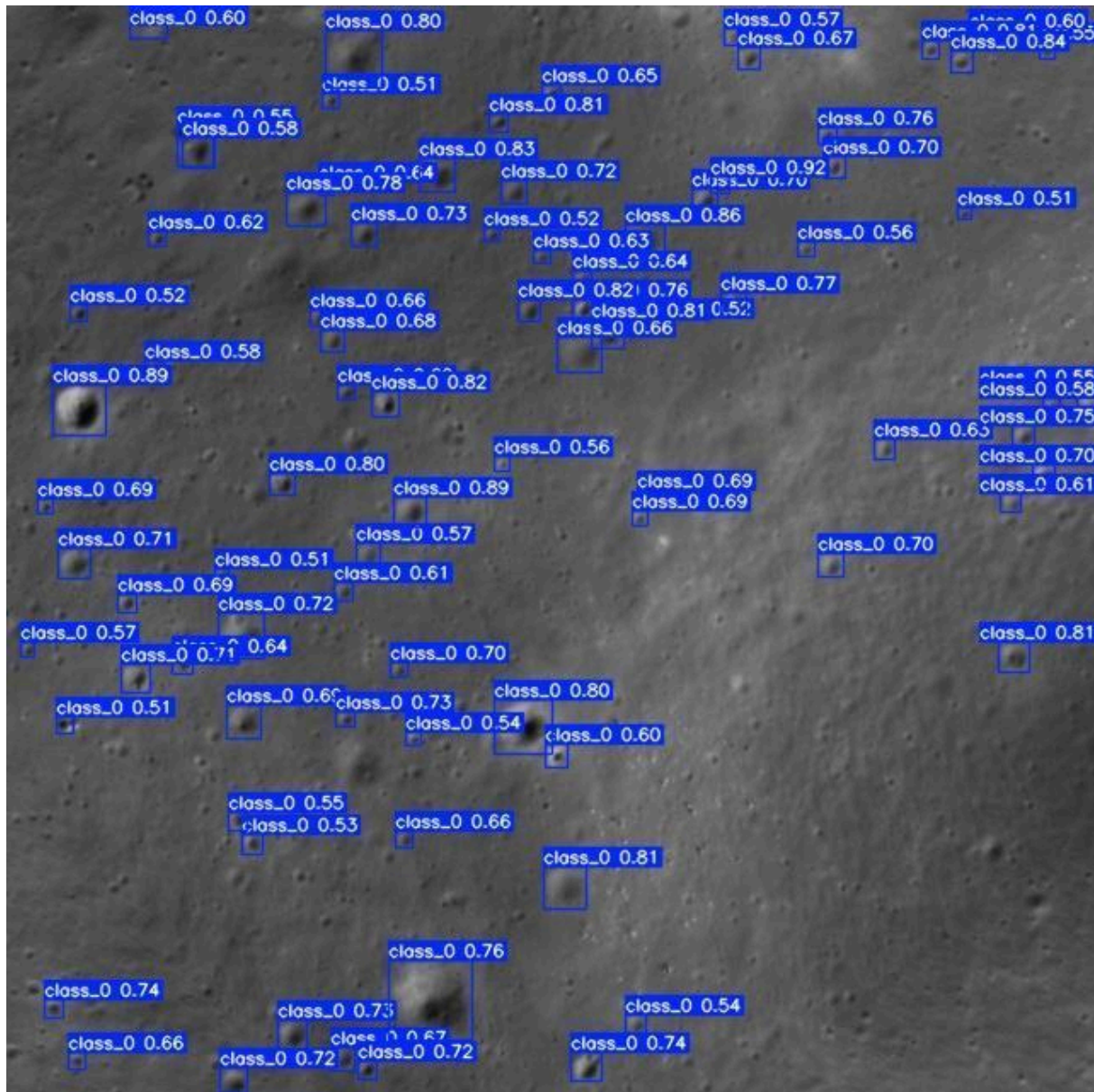


Figure 6.8 - Model version 2 inference test (*confidence* minimum of 0.5)

Looking at Figure 6.8, model version 2 makes a lot of progress for only adding 30 labeled images to the dataset. Increasing the dataset improved detection density, or in other words, there are a lot more craters detected than before. Some smaller craters begin to be identified but there are still several boundary box issues and obvious craters that should've been detected. Again, the

results helped influence the direction of training and what needed to be done. In this case, a refinement of the initial 20 images from model version 1 needed to be done to improve bounding box quality while increasing the number of craters to train on.

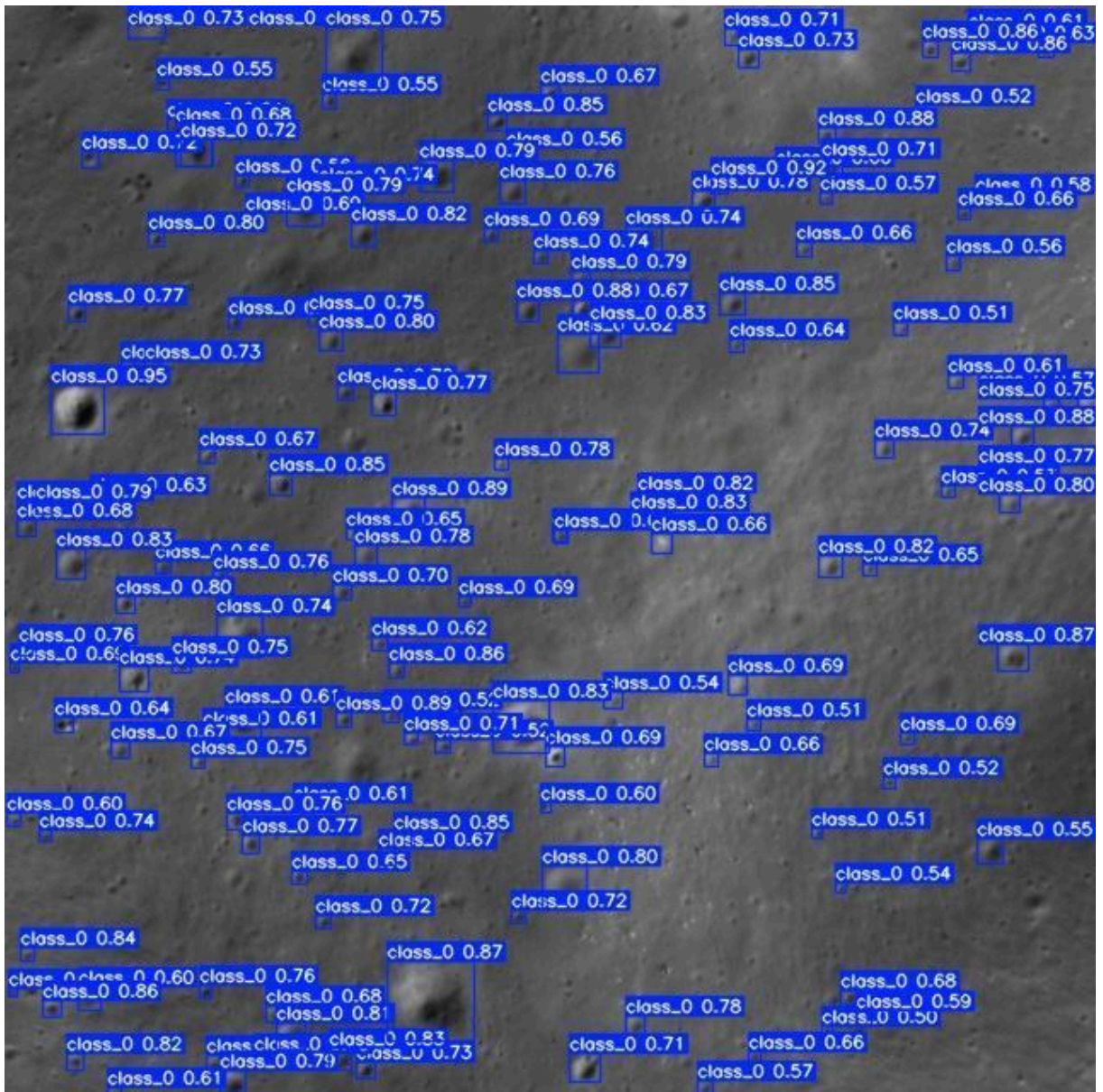


Figure 6.9 - Model version 3 inference test (*confidence* minimum of 0.5)



Figure 6.10 - Model version 4 inference test (*confidence* minimum of 0.5)

Figures 6.9 and 6.10 show model versions 3 and 4 results for the inference test. There is a notable improvement from each version to the next, but it is becoming increasingly more difficult to tell exactly what these improvements are. From version 3, it is easy to see that bounding boxes became more precise while the number of craters detected again increased. The changes made were simply refining the initial dataset which managed to make a noticeable difference in performance. Looking at version 4, again there are clear improvements in both consistency and spatial accuracy. The model is better at generalizing small to medium sized craters than it was before and shows fewer incorrect predictions. Both versions highlight just how important dataset improvements are and how much they can impact the models abilities to detect and classify.

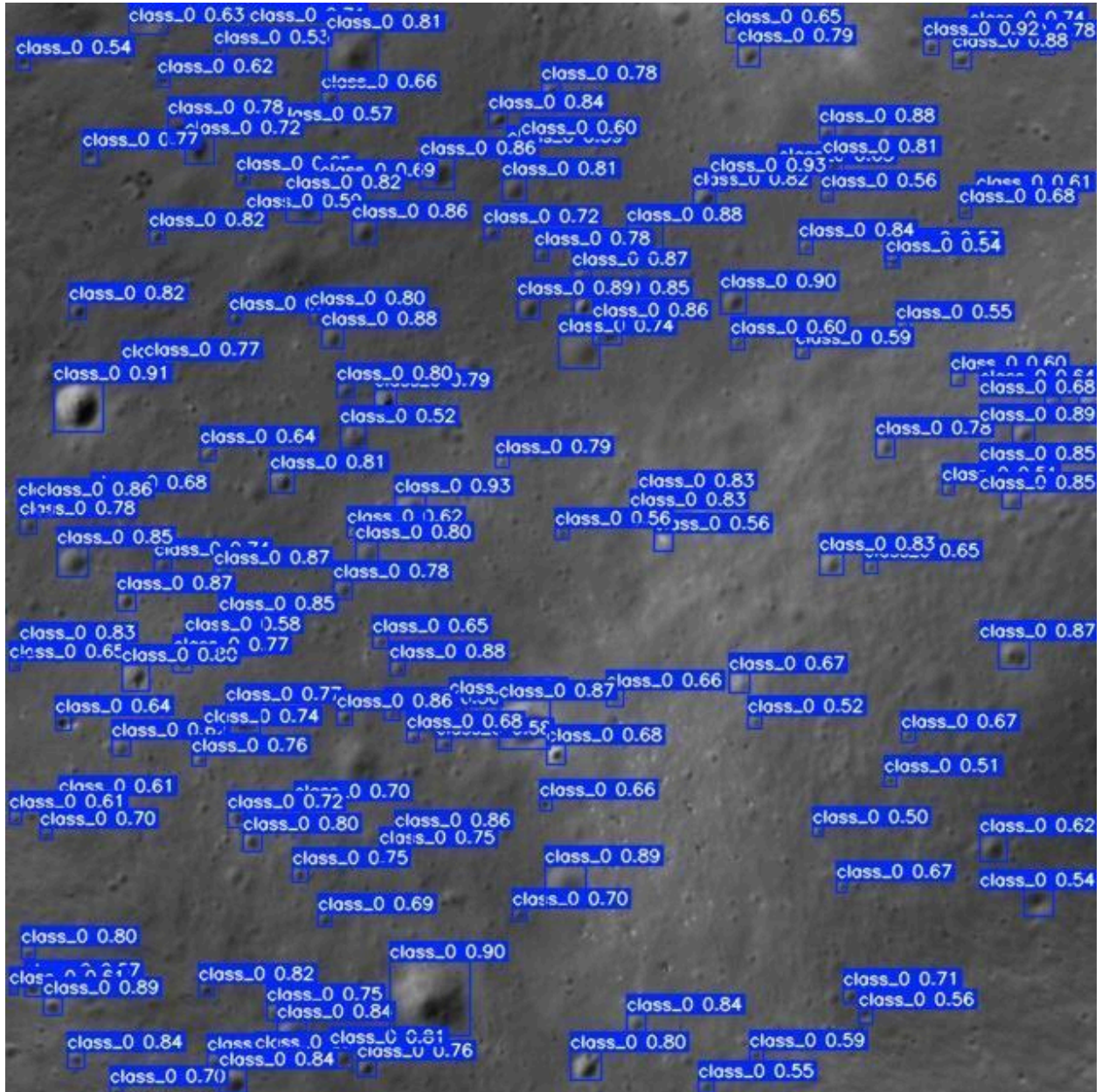


Figure 6.11 - Model version 5 inference test (*confidence* minimum of 0.5)

Looking at the inference test for model version 5, as seen in Figure 6.11, there is actually a drop off in the number of predictions made by the model. When comparing version 4 and 5 side by side, it is obvious to see that less detections were made, this could be as a result of the drop in *precision* performance at low *confidence*. This drop in performance could also be a result of autolabeling and bias. In order to reduce labeling time, version 4 of the model was used to make labels on unseen images which would then be refined by hand using Label Studio and finally used for training in version 5. Looking back at the results, although there is a drop in craters detected, the bounding boxes are still as precise as the last test while the model also makes fewer false detections.

Training becomes a difficult task especially as the model becomes more refined. Much like the performance metrics discussed in section 6.4, the model begins to converge in performance. When the model starts to converge, consistency in labeling is very important because adding new images with biased labeling could result in performance decreases or errors.

In conclusion, the model performance drastically changed from version 1 to 5 with visual results showing a large increase in performance. This inference testing confirms the performance metric results discussed previously. Across all five model versions, visually observable detection quality improved as the dataset was expanded and became more balanced in labeling quality. While version 5 did have a slight performance decrease, with more labeled images and bounding box refinements, the model could keep improving in an upwards direction. These results only continually confirm that dataset size and diversity play a key role in developing a crater detection model capable of identifying craters in new environments.

6.6 YOLO Nano vs YOLO Small

This section will be brief and provide a direct comparison between the YOLOv11 Nano and Small architectures to evaluate how model size influences crater detection performance. Looking back at Figure 6.2, there is a significant difference between the two models in terms of the parameters and computational GFLOPS. This comparison looks at just the basic performance metrics such as *precision-confidence* and *recall-confidence*.

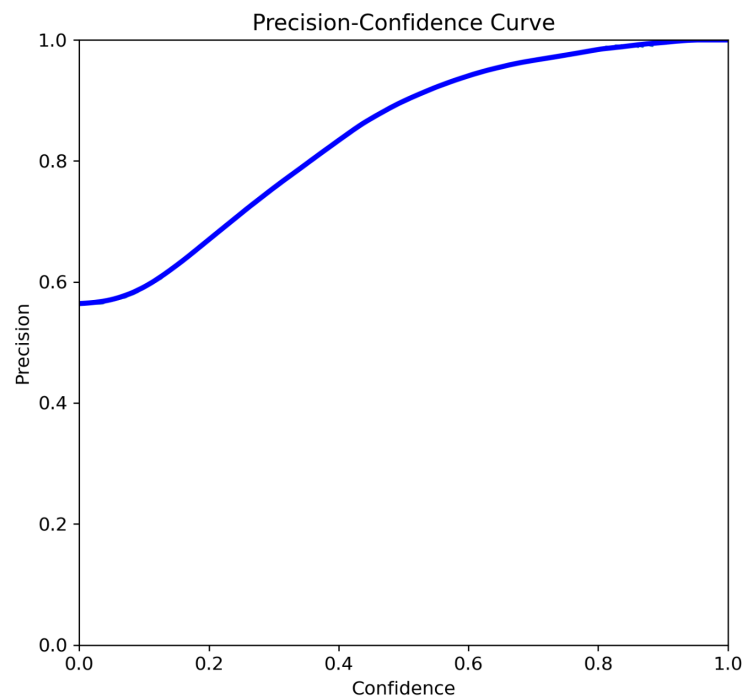


Figure 6.12 - Nano model *precision-confidence* curve

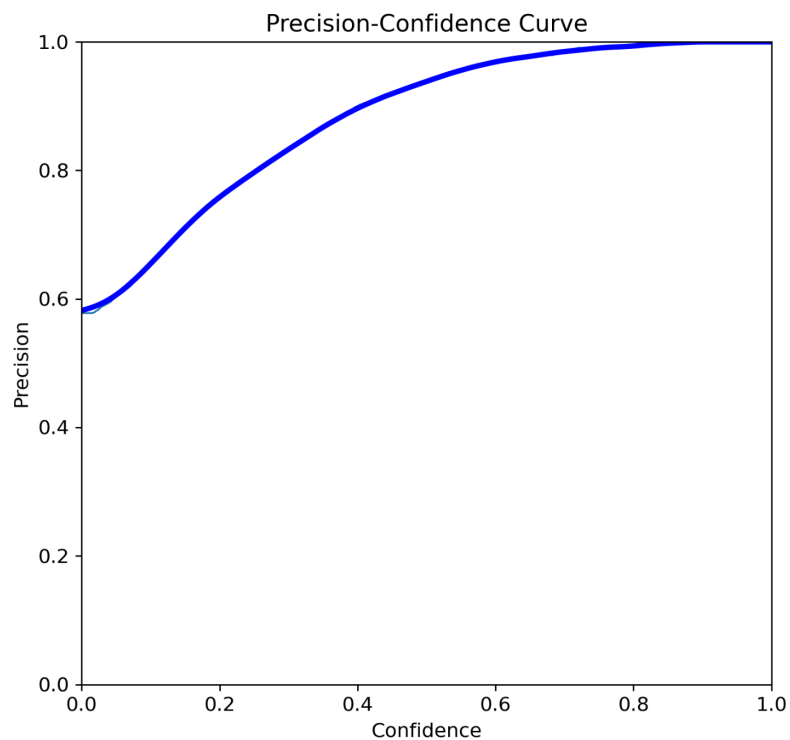


Figure 6.13 - Small model *precision-confidence* curve

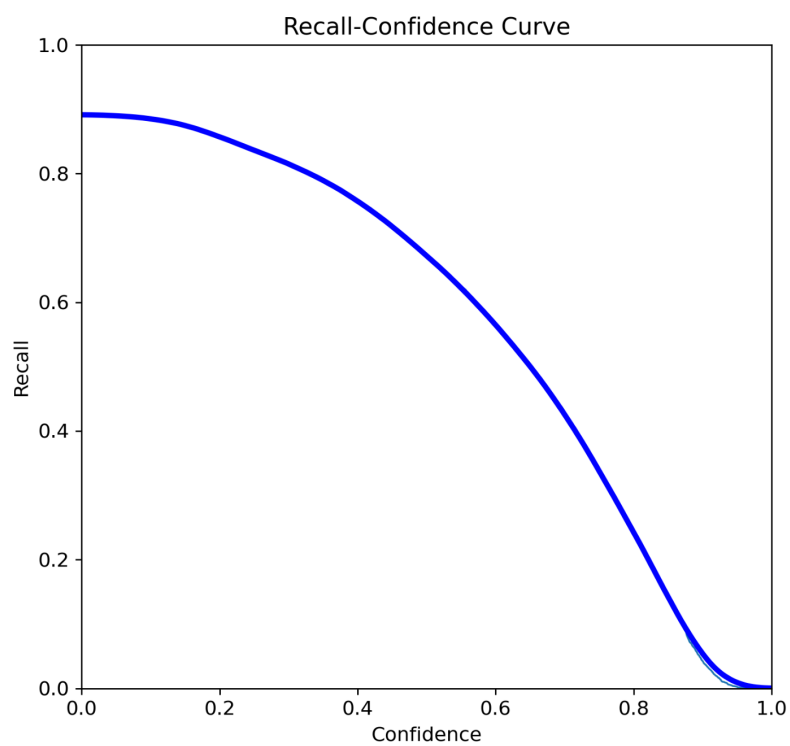


Figure 6.14 - Nano model *recall-confidence* curve

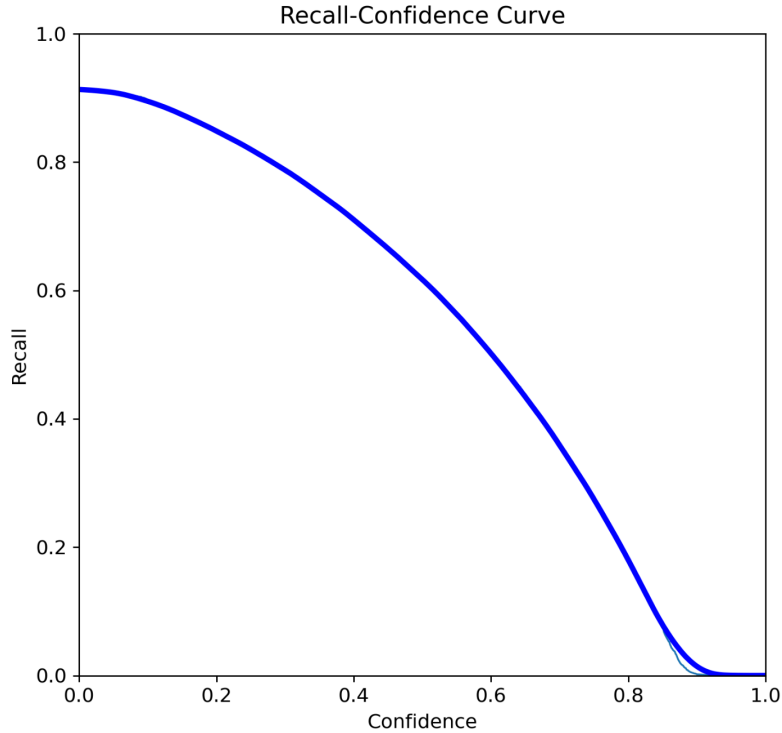


Figure 6.15 - Small model *recall-confidence* curve

When looking at the performance metrics between Nano and Small, there is only a slight increase in *precision* and *recall*. Figures 6.12 to 6.15 highlight these minor differences in performance which are expected even though Small is a larger model. Computational computing power plays an important role and means that the smaller model can maximize performance with the available resources giving it performance on par with a larger model like the YOLOv11 Small. The Small model still maintains higher *recall* at all *confidence* levels, suggesting reduced sensitivity to missing craters as *confidence* increases. Similarly, the *precision* is higher at all *confidence* levels, meaning more stable detections across the entire *confidence* threshold.

Overall, the YOLO Small architecture does outperform YOLO Nano in *precision* and *imetrics*. This is due more parameters being available to be trained to better learn the feature characteristics of lunar surface craters. Given the available computing power, the model is able to use all of it to its fullest potential to maximize performance. While the larger model does outperform YOLO Nano, it is important to recognize that this performance increase is not significant. This minimal performance increase means that inference testing will most likely not show a noticeable difference in the number of detections made by each model when compared.

Chapter 7 - Conclusion

The work done for this project presents a complete end-to-end framework for developing an autonomous lunar crater detection model using deep learning. This crater detection model was based on the pretrained YOLOv11 Nano architecture but was specifically trained on a custom dataset of lunar craters. This end-to-end framework includes data augmentation on images taken by the LRO to create this custom dataset. Data augmentation methodology involved splitting up large images into patches, performing random rotations, and randomly adjusting brightness. All of these methods made for a diverse dataset that was able to capture different landing environments that a spacecraft might encounter. Custom labeling was used to ensure that the data was consistent across all samples, reducing annotation bias and improving dataset quality.

Five successive versions of the YOLOv11 Nano model were trained and evaluated, each on an increasingly larger and more diverse dataset. The results showed the clear impact that dataset refinement can have on detection quality and performance metrics. These improvements in metrics such as *precision-confidence*, *recall-confidence*, etc., demonstrate how expanding the dataset for a model to train on improves generalization. Starting from version 1, which was trained on a small dataset of 20 images with relatively poor labeling quality, to the final version, which had 110 images with higher quality labeling. These later versions produced much smoother and more stable curves with *precision* and *recall* tradeoffs as well as in the F1 and *precision-recall* metrics. These trends indicated that the model was converging to some optimal performance point with ever diminishing returns. It is important to point out that performance increases between versions remained noticeable when observing the inference testing results. These inference tests proved that the final model version is capable of distinguishing between craters and the lunar surface background. Of course, more rigorous testing should be carried out to continually verify this conclusion.

Improvements and future work for this crater detection model could further improve performance bringing it up to par with a real world system. With more time and resources, a large enough dataset could be put together to produce even more consistent, accurate, and precise results. Other updates look at reworking the deep neural network making it fully customizable specifically for crater detection. This could mean reworking the YOLOv11 Nano architecture, or creating an entirely new architecture from scratch with the YOLOv11 framework. Other considerations could be changing the computer vision model type to something more precise such as segmentation rather than object detection. Synthetic data could be developed and sourced to help improve the data sharpness and variability of the dataset further. Other systems such as MiDas could be integrated to provide depth estimation capabilities which would provide valuable 3D terrain understanding.

References

- [1] Chen, M., “What Is Machine Learning?,” *Oracle*, 25 November 2024. Retrieved 25 February 2025 from <https://www.oracle.com/artificial-intelligence/machine-learning/what-is-machine-learning/>.
- [2] “Commercial lunar payload services,” *NASA*, 15 June 2024. Retrieved 24 February 2025 from <https://www.nasa.gov/reference/commercial-lunar-payload-services/>.
- [3] “Why is it so difficult to land on the Moon?,” *Astrobotic*, 22 December 2023. Retrieved 22 February 2025 from <https://www.astrobotic.com/why-is-it-so-difficult-to-land-on-the-moon/>.
- [4] Parisi, M., Panontin, T., Wu, S.-C., Mctigue, K., and Vera, A., “Effects of Communication Delay on Human Spaceflight Missions,” *AHFE International*, vol. 98, 2023, pp. 64–73. <https://www.doi.org/10.54941/ahfe1003920>.
- [5] Yadav, A., “Computer vision vs machine learning,” *Medium*, 6 August 2024. Retrieved 18 February 2025 from <https://medium.com/@amit25173/computer-vision-vs-machine-learning-979b7d75cd61>.
- [6] Holdsworth, J., and Scapicchio, M., “What is deep learning?,” *IBM*, 17 June 2024. Retrieved 17 February 2025 from <https://www.ibm.com/think/topics/deep-learning>.
- [7] Peterson, K., Duijn, B., “Leveraging artificial intelligence for enhanced seed testing: current trends and future prospects,” *International Seed Testing Association*, Apr. 2024, pp. 8-10. [\(PDF\) Leveraging Artificial Intelligence for Enhanced Seed Testing: Current trends and future prospects](#)
- [8] Purwono, P., Ma’arif, A., Rahmانيar, W., Fathurrahman, H. I., Frisky, A. Z., and Haq, Q. M., “Understanding of convolutional neural network (CNN): A Review,” *International Journal of Robotics and Control Systems*, vol. 2, Jan. 2023, pp. 739–748. <https://doi.org/10.31763/ijrcs.v2i4.888>
- [9] Reddy, S., Pillay, N., and Singh, N., “Comparative evaluation of Convolutional Neural Network object detection algorithms for vehicle detection,” *Journal of Imaging*, vol. 10, Jul. 2024, p. 162. <https://doi.org/10.3390/jimaging10070162>
- [10] Zhang, H., Zhang, Y., Feng, Q., and Zhang, K., “A semi-supervised object detection method for close range detection of spacecraft and Space Debris,” *International Journal of Aeronautical and Space Sciences*, vol. 26, Jun. 2024, pp. 773–784. [v8https://doi.org/10.1007/s42405-024-00756-y](https://doi.org/10.1007/s42405-024-00756-y)
- [11] Hussain, M., “Yolov1 to V8: Unveiling each variant—a comprehensive review of Yolo,” *IEEE Access*, vol. 12, Mar. 2024, pp. 42816–42833. <https://doi.org/10.1109/ACCESS.2024.3378568>
- [12] Mu, L., Xian, L., Li, L., Liu, G., Chen, M., and Zhang, W., “Yolo-crater model for small crater detection,” *Remote Sensing*, vol. 15, Oct. 2023, p. 5040. <https://doi.org/10.3390/rs15205040>

- [13] Di, K., Jia, M., Xin, X., Liu, B., Liu, Z., Peng, M., and Yue, Z., “High resolution seamless dom generation over Chang’E-5 Landing area using LROC NAC images,” *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. XLII–3, Apr. 2018, pp. 271–276.
<https://doi.org/10.5194/isprs-archives-XLII-3-271-2018>
- [14] Tewari, A., Verma, V., Srivastava, P., Jain, V., and Khanna, N., “Automated crater detection from co-registered optical images, elevation maps and slope maps using Deep Learning,” *Planetary and Space Science*, vol. 218, Sep. 2022, p. 105500.
<https://doi.org/10.1016/j.pss.2022.105500>
- [15] Silvestrini, S., Piccinin, M., Zanutti, G., Brandonisio, A., Bloise, I., Feruglio, L., Lunghi, P., Lavagna, M., and Varile, M., “Optical navigation for lunar landing based on Convolutional Neural Network Crater Detector,” *Aerospace Science and Technology*, vol. 123, Apr. 2022, p. 107503. <https://doi.org/10.1016/j.ast.2022.107503>
- [16] Zhao, X., Wang, L., Zhang, Y., Han, X., Deveci, M., and Parmar, M., “A review of Convolutional Neural Networks in computer vision,” *Artificial Intelligence Review*, vol. 57, Mar. 2024. <https://doi.org/10.1007/s10462-024-10721-6>
- [17] Alake, R., “Loss functions in machine learning explained,” *DataCamp* Available, 4 December 2024. Retrieved 8 April 2025 from
<https://www.datacamp.com/tutorial/loss-function-in-machine-learning>.
- [18] Li, Z., Liu, F., Yang, W., Peng, S., and Zhou, J., “A survey of Convolutional Neural Networks: Analysis, applications, and prospects,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, Dec. 2022, pp. 1–21.
<https://doi.org/10.1109/TNNLS.2021.3084827>
- [19] Taye, M., “Theoretical understanding of Convolutional Neural Network: Concepts, architectures, applications, Future Directions,” *SSRN Electronic Journal*, vol. 11, Mar. 2023, p. 52. <https://doi.org/10.3390/computation11030052>
- [20] Murel, J., and Kavlakoglu, E., “What is regularization?,” *IBM* Available, 16 November 2023. Retrieved 10 April 2025 from
<https://www.ibm.com/think/topics/regularization#:~:text=Regularization%20is%20a%20set%20of,overfitting%20in%20machine%20learning%20models>
- [21] Terven, J., Córdova-Esparza, D.-M., and Romero-González, J.-A., “A comprehensive review of YOLO architectures in Computer Vision: From YOLOV1 to Yolov8 and Yolo-Nas,” *Machine Learning and Knowledge Extraction*, vol. 5, Nov. 2023, pp. 1680–1716. <https://doi.org/10.3390/make5040083>
- [22] Rao, S. N., “Yolov11 architecture explained: Next-level object detection with enhanced speed and accuracy,” *Medium* Available, 22 October 2024. Retrieved 15 April 2025 from
<https://medium.com/@nikhil-rao-20/yolov11-explained-next-level-object-detection-with-enhanced-speed-and-accuracy-2dbe2d376f71>

- [23] “Yolov8 vs Yolo11: A technical comparison,” *Ultralytics YOLO Docs* Available, 25 April 2024. Retrieved 16 April 2025 from <https://docs.ultralytics.com/compare/yolov8-vs-yolo11/>
- [24] Robinson, M. S., Brylow, S. M., Tschimmel, M., Humm, D., Lawrence, S. J., Thomas, P. C., Denevi, B. W., Bowman-Cisneros, E., Zerr, J., Ravine, M. A., Caplinger, M. A., Ghaemi, F. T., Schaffner, J. A., Malin, M. C., Mahanti, P., Bartels, A., Anderson, J., Tran, T. N., Eliason, E. M., McEwen, A. S., Turtle, E., Jolliff, B. L., and Hiesinger, H., “Lunar Reconnaissance Orbiter Camera (LROC) Instrument Overview,” *Space Science Reviews*, vol. 150, Mar. 2010, pp. 81–124. <https://doi.org/10.1007/s11214-010-9634-2>
- [25] “Lunar Orbital Data Explorer,” *Washington University in St. Louis*, Retrieved 15 April 2025 from <https://ode.rsl.wustl.edu/moon/index.aspx>

Appendix A - Python Script for Converting IMG to PNG Format: 'IMG2PNG.py'

```
import os
from osgeo import gdal
from data_augmentation import image_augmentation

def convert_IMG_to_PNG(input_folder, output_folder, augment_brightness=False, divide=False, augment_rotation=False):
    """Convert .IMG file to .PNG image for image augmentation"""
    # Creates output directory
    os.makedirs(output_folder, exist_ok=True)
    gdal.UseExceptions()
    folder_count = 0

    for file_name in os.listdir(input_folder):
        if file_name.endswith('.IMG'):
            # Joining input and output folder paths with the filename in .xml and .png respectively
            input_path = os.path.join(input_folder, file_name)
            output_path = os.path.join(output_folder, file_name.replace('.IMG', '.png'))
            print('In Directory')

            # Open the image dataset
            dataset = gdal.Open(input_path)

            # Get band statistics (min/max pixel values)
            band = dataset.GetRasterBand(1)
            min_val, max_val = band.ComputeRasterMinMax()

            # Convert dataset to 8-bit PNG
            gdal.Translate(output_path, dataset, format='PNG', outputType=gdal.GDT_Byte,
                           noData=0, scaleParams=[[min_val, max_val, 0, 255]])

            if divide:
                print('Dividing')
                # Create output directory/folder for each image at the given output path
                folder_dir = f"{output_folder}/patches_{folder_count}"
                os.makedirs(folder_dir, exist_ok=True)

                # Call the divide function with the input path of the image and folder directory to store patches
                image_augmentation.divide(output_path, folder_dir, resolution=640,
                                           augment_brightness=augment_brightness, augment_rotation=augment_rotation)

                # Iterate to the next folder name
                folder_count += 1

            print(f'Converted {file_name} to {output_path}')

    return

if __name__ == "__main__":
    # LROC NAC file directories
    input_folder = 'C:/Users/Matthew C/Desktop/AE 295A/Test NASA Data/LROC Images/Experimental/NACFiles'
    output_folder = 'C:/Users/Matthew C/Desktop/AE 295A/Test NASA Data/LROC Images/Experimental/NACImages'
    convert_IMG_to_PNG(input_folder, output_folder, augment_brightness=True, divide=True, augment_rotation=True)
```

Appendix B - Python Script for Creating Patches and Performing Data Augmentation: 'image_augmentation.py'

```
import random
import cv2
import numpy as np
import os
from PIL import Image

def divide(input_png, output_folder, resolution, augment_brightness=False, augment_rotation=False):
    """Divide .PNG image into multiple images, each time checking if the
    patch brightness and rotation needs to be changed"""
    # Assign the PNG path to a variable and create an array with the data from that image
    image_path = input_png
    Image.MAX_IMAGE_PIXELS = None
    image = np.array(Image.open(image_path))

    # Define image resolution for each patch size
    patch_size = resolution
    overlap = resolution

    # Create output directory/folder to store the patches given the input of the function
    output_dir = output_folder
    os.makedirs(output_dir, exist_ok=True)

    patch_count = 0
    for y in range(0, image.shape[0] - patch_size + 1, overlap):
        for x in range(0, image.shape[1] - patch_size + 1, overlap):
            # Create the patch with the given size and patch resolution
            patch = image[y:y + patch_size, x:x + patch_size]

            # Label each patch numerically starting from 0 and store in the output folder
            patch_filename = f"{output_dir}/patch_{patch_count}.png"

            if augment_rotation:
                # Rotate the patch with the same size, add a reflected border to avoid cropping
                patch = random_rotation(patch)

            if augment_brightness:
                # Randomly adjust the brightness of the patch
                patch = random_brightness(patch)

            # patch = cv2.bilateralFilter(patch, d=3, sigmaColor=20, sigmaSpace=20)
            Image.fromarray(patch).save(patch_filename)
            patch_count += 1

    print(f"saved {patch_count} patches in {output_dir}")

    return

def random_rotation(image):
    """Get random angle from 0 to 360 degrees and define other parameters for rotation"""
    angle = random.uniform(0, 360)
    h, w = image.shape[:2]
    center = (w // 2, h // 2)

    # Get rotation matrix for random angle
    M = cv2.getRotationMatrix2D(center, angle, scale=1.0)

    return cv2.warpAffine(image, M, (w, h), flags=cv2.INTER_LINEAR, borderMode=cv2.BORDER_REFLECT)

def random_brightness(image):
    """Randomly select the contrast adjustment values"""
    alpha = random.uniform(0.8, 1.5)
    beta = random.randint(-30, 30)

    return cv2.convertScaleAbs(image, alpha=alpha, beta=beta)
```

Appendix C - Python Main Script for Training and Inference Testing: 'main.py'

```
import torch
import os
import cv2
import random
import matplotlib.pyplot as plt
import yaml
from ultralytics import YOLO

def visualize_labels(image_path, label_path, n=5):
    img_dir = f"{image_path}"
    label_dir = f"{label_path}"
    img_files = random.sample(os.listdir(img_dir), n)

    for img_file in img_files:
        img = cv2.imread(f"{img_dir}/{img_file}")
        h, w = img.shape[:2]

        label_file = img_file.replace('.png', '.txt').replace('.jpg', '.txt')
        with open(f"{label_dir}/{label_file}") as f:
            for line in f.readlines():
                cls, x, y, bw, bh = map(float, line.split())
                x1, y1 = int((x - bw/2) * w), int((y - bh/2) * h)
                x2, y2 = int((x + bw/2) * w), int((y + bh/2) * h)
                cv2.rectangle(img, (x1, y1), (x2, y2), (255,0,0), 2)

        plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
        plt.axis('off')
        plt.show()

if __name__ == "__main__":
    with open('config.yaml', 'r') as dirs:
        directories = yaml.safe_load(dirs)

    # Directories
    image_path = directories['paths']['train_images']
    label_path = directories['paths']['train_labels']

    # Checks
    Visualization = True
    CUDA_Check = False

    # Custom model name
    pretrained_model = 'yolo11n.pt'
    custom_model = 'yolo11n_custom.pt'

    # Tasks
    Train = False
    Load_Custom_Model = False
    Predict = False
    Autolabel = False

    if Visualization:
        # Visualize the labels to ensure that they are loaded in properly for training
        visualize_labels(image_path, label_path)
```

```

if CUDA_Check:
    # Check if CUDA is available, if it is use CUDA with the GPU, if not use CPU
    if torch.cuda.is_available():
        device = torch.device('cuda')
        print(f'Using GPU: {torch.cuda.get_device_name(0)}')
        print(torch.cuda.memory_allocated())
    else:
        device = torch.device('cpu')
        print(f'Using CPU')
        print(torch.cuda.memory_allocated())

if Train:
    # Load base YOLOv11n model and train
    model = YOLO(pretrained_model)
    model.train(
        data=directories['paths']['dataset'],
        imgsz=640,
        batch=8,
        epochs=100,
        workers=0,
        device=0)

if Load_Custom_Model:
    # Load custom model
    model = YOLO(custom_model)
    model.info(detailed=False)

if Predict:
    # Make baseline predictions
    model.predict(
        source=directories['paths']['test_image'],
        show=False,
        save=True,
        line_width=1,
        show_labels=True,
        show_conf=True)
    model.predict(
        source=directories['paths']['test_image'],
        show=False,
        save=True,
        conf=0.5,
        line_width=1,
        show_labels=True,
        show_conf=True)
    model.predict(
        source=directories['paths']['test_flyover'],
        show=False,
        save=True,
        conf=0.5,
        line_width=1,
        show_labels=True,
        show_conf=True)
    model.predict(
        source=directories['paths']['test_zoom'],
        show=False,
        save=True,
        conf=0.5,
        line_width=1,
        show_labels=True,
        show_conf=True)

```

```
if Autolabel:
    # Autolabeling
    output_folder = 'C:/Users/Matthew C/Desktop/AE 295A/Dataset/autolabels'
    os.makedirs(
        output_folder,
        exist_ok=True)
    source_path = 'C:/Users/Matthew C/Desktop/AE 295A/Dataset/autolabels/images/'
    model.predict(
        source=source_path,
        save=True,
        save_txt=True,
        conf=0.5,
        show_labels=False,
        show_conf=False,
        line_width=1,
        project=output_folder,
        name='predictions')
```

Appendix D - Python YAML Configuration File: 'config.yaml'

```
# =====  
# PROJECT DESCRIPTION  
# =====  
project:  
  name: "Crater Detection for Autonomous Lunar Approach Using Deep Learning"  
  author: "Matthew C"  
  description: "YOLO-based lunar crater detection model"  
  versions: 5  
  
# =====  
# IMAGE PATHS  
# =====  
paths:  
  dataset: "datasets/custom_dataset.yaml"  
  train_images: "C:/Users/Matthew C/Desktop/AE 295A/Dataset/train/images"  
  train_labels: "C:/Users/Matthew C/Desktop/AE 295A/Dataset/train/labels"  
  test_image: "1.png"  
  test_flyover: "flyover.mp4"  
  test_zoom: "zoom_video.mp4"
```

Appendix E - Python YAML File Showing General YOLO11 Object Detection Model Architecture from Ultralytics: 'yolo11.yaml'

```
# Parameters
nc: 80 # number of classes
scales: # model compound scaling constants, i.e. 'model=yolo11n.yaml' will call yolo11.yaml with scale 'n'
  # [depth, width, max_channels]
  n: [0.50, 0.25, 1024] # summary: 181 layers, 2624080 parameters, 2624064 gradients, 6.6 GFLOPs
  s: [0.50, 0.50, 1024] # summary: 181 layers, 9458752 parameters, 9458736 gradients, 21.7 GFLOPs
  m: [0.50, 1.00, 512] # summary: 231 layers, 20114688 parameters, 20114672 gradients, 68.5 GFLOPs
  l: [1.00, 1.00, 512] # summary: 357 layers, 25372160 parameters, 25372144 gradients, 87.6 GFLOPs
  x: [1.00, 1.50, 512] # summary: 357 layers, 56966176 parameters, 56966160 gradients, 196.0 GFLOPs

# YOLO11n backbone
backbone:
  # [from, repeats, module, args]
  - [-1, 1, Conv, [64, 3, 2]] # 0-P1/2
  - [-1, 1, Conv, [128, 3, 2]] # 1-P2/4
  - [-1, 2, C3k2, [256, False, 0.25]]
  - [-1, 1, Conv, [256, 3, 2]] # 3-P3/8
  - [-1, 2, C3k2, [512, False, 0.25]]
  - [-1, 1, Conv, [512, 3, 2]] # 5-P4/16
  - [-1, 2, C3k2, [512, True]]
  - [-1, 1, Conv, [1024, 3, 2]] # 7-P5/32
  - [-1, 2, C3k2, [1024, True]]
  - [-1, 1, SPPF, [1024, 5]] # 9
  - [-1, 2, C2PSA, [1024]] # 10

# YOLO11n head
head:
  - [-1, 1, nn.Upsample, [None, 2, "nearest"]]
  - [[-1, 6], 1, Concat, [1]] # cat backbone P4
  - [-1, 2, C3k2, [512, False]] # 13

  - [-1, 1, nn.Upsample, [None, 2, "nearest"]]
  - [[-1, 4], 1, Concat, [1]] # cat backbone P3
  - [-1, 2, C3k2, [256, False]] # 16 (P3/8-small)

  - [-1, 1, Conv, [256, 3, 2]]
  - [[-1, 13], 1, Concat, [1]] # cat head P4
  - [-1, 2, C3k2, [512, False]] # 19 (P4/16-medium)

  - [-1, 1, Conv, [512, 3, 2]]
  - [[-1, 10], 1, Concat, [1]] # cat head P5
  - [-1, 2, C3k2, [1024, True]] # 22 (P5/32-large)

  - [[16, 19, 22], 1, Detect, [nc]] # Detect(P3, P4, P5)
```