SOLAR PANEL DAMAGE IDENTIFICATION USING TENSORFLOW LITE

By

Garrick D. Muncie

SOLAR PANEL DAMAGE IDENTIFICATION USING TENSORFLOW LITE

By

Garrick D. Muncie

A Thesis Submitted to the Faculty of the University of
Tennessee at Chattanooga in Partial
Fulfillment of the Requirements of the Degree
Of Master of Science in Engineering

The University of Tennessee at Chattanooga
Chattanooga, Tennessee

August 2024

ABSTRACT

The number of utility-scale PV installations is rising, with a power capacity of 12.5 Gigawatts installed in 2021, 10.4 in 2022, and an estimated 24 Gigawatts installed in 2023 [1]. With larger-scale installations, quicker ways of identifying and locating damaged PV arrays are needed. The solution presented in this thesis is to use drones to capture aerial photos and TensorFlow-Lite and Keras deep learning methods to determine if a panel has defects, such as debris, cracked panels, and hotspots. The model features an execution time of 0.185 seconds per picture. In addition, the model will run on an embedded system with a relatively low impact on power consumption, minimizing the reduction of flight time. The Raspberry Pi has an approximate 0.1-minute effect on flight time while idling and with the worst-case scenario of affecting flight time by approximately two minutes if left running for the entire flight.

DEDICATION

I would like to dedicate this thesis to my loving family, who have pushed me to better myself throughout the years. I want to give a special callout to all my grandparents, who were always there for me and pushed me to explore the world, and my loving girlfriend, who had to put up with me while I furiously worked my way through my master's degree. It also convinced me not to give up and to keep pushing through while I did school and work.

ACKNOWLEDGMENTS

I would like to express my thanks to Dr. Ofoli for his guidance and the opportunity to work on this endeavor. Dr. Ahmed and Dr. Reising, thank you for your guidance throughout the year and for serving on the thesis committee on the short turnaround of this venture. The SMART Scholarship and Arnold Air Force Base for giving me the opportunity to work and be a student at the same time.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

PV,	Photo Voltaic
UAV,	Unmanned Aerial Vehicle
GPU,	Graphic Processing Unit
CNN,	Convolutional Neural Network
GPS,	Global Positioning System
UV,	Ultraviolet
EL,	Electroluminescence
IR,	Infrared
ReLU,	Rectified Linear Unit
VGG16,	Visual Geometry Group 16
MCC,	Matthews Correlation Coefficient
ResNet,	Residual Neural Network
YOLO,	You Only Look Once
SGD,	Stochastic Gradient Descent

LIST OF SYMBOLS

∑, Summation Symbol

CHAPTER I

INTRODUCTION

With the continuous growth of residential and utility-scale photovoltaic (PV) sources (solar panels), it is of growing importance for utilities to gain an increased understanding of solar panels and how the PV's status affects the utility system[2]. The number of utility-scale PV installations is rising, with a power capacity of 12.5 Gigawatts installed in 2021, 10.4 in 2022, and an estimated 24 Gigawatts installed in 2023 [1]. This growth requires improvements to current damage collection and repair diagnostics solutions and new solutions to diagnose issues. Current solutions in machine learning focus on certain subsets of damage collection and repair diagnostics [3-25]. The solution historically has been based on different photo types, electroluminescence, Infrared Imagery, still images, or a combination. These photos have been taken from unmanned aerial vehicles (UAV) or personnel. The deep learning on these images has happened mainly with off-site computing [4-18, 24, 26-30]. However, a currently sought-after solution is flying over a utility field with a UAV, taking pictures, Geo-tagging its location with Global Positioning Systems (GPS), and processing the image with a machine learning neural network. The images are then classified into areas for a technician to understand what is wrong with the panel and send them on their way to fix the identified panel. This has been done to some varying degrees [2-30]. A review of the literature reveals that few solutions seem to cover a large span of images and damage types and compare between different devices. This thesis seeks to close this gap.

Research Questions

In this thesis, the following research questions were asked:

- With the wide array of PV damage types, is there a way to classify types of damage with a single model?
- What is the power consumption differences between different edge devices? (such as a Raspberry Pi 4 or Nvidia Jetson Nano)
    - Will power consumption differences affect drone flight times?

CHAPTER II

BACKGROUND

Neural Networks

Neural networks represent a class of machine learning models loosely inspired by studies about the central nervous system. Each Neural Network comprises several interconnected "neurons" arranged into "layers." These neurons pass information messages to other neurons in the next layer. The first studies started in the early 1950s and have grown exponentially in recent years [31]. At its basic premise, machine learning is broken up into two subsets: unsupervised and supervised machine learning. In unsupervised learning, where there is no known output, the learning algorithm is given input data and is asked to extract knowledge from said data [32]. Supervised learning is one of the most common and successful types of machine learning. It is used whenever a given input is used to predict. For supervised learning algorithms to work, they require example inputs together with corresponding example outputs. [32].

Deep Learning

Deep learning is currently one of the most popular approaches to machine learning. Deep learning started getting its name when these types of neural networks utilized three to five layers. Research on the nature of deep learning has existed for quite some time. However, the complexity and computational power required of deep learning hindered its adoption and application until the mid-2000s. With the development of advanced processing units and large exponential increases in data, there has been a resurgence in deep learning research. These advancements have now allowed for networks with more than 200 layers [31,33].

Convolutional Neural Networks

Convolutional neural networks (CNNs) have been applied to visual tasks since the late 1980s. However, despite a few scattered applications, they were dormant until the mid-2000s when developments in computing power and the advent of large amounts of labeled data, supplemented by improved algorithms, contributed to their advancement and brought them to the forefront of a neural network renaissance that has seen rapid progression since 2012 [34]. CNNs consist of input, hidden, and output layers. The hidden layers are added to the network because the additional neurons can facilitate learning more complex patterns in the training data. They are called hidden because they do not directly connect with the input or the output layer of the Neural Network [35]. Hidden layers allow the model to calculate more coefficients (weights) for the model to learn [36]. These weights are numerical values that determine the strength of a connection or signal between nodes. The value of these weights is adjusted in the training phase of a Neural Network and ultimately helps define the output of the predictive system. Figure 2.1 provides a representative illustration of a CNN comprised of an input, a single hidden layer, and an output layer.

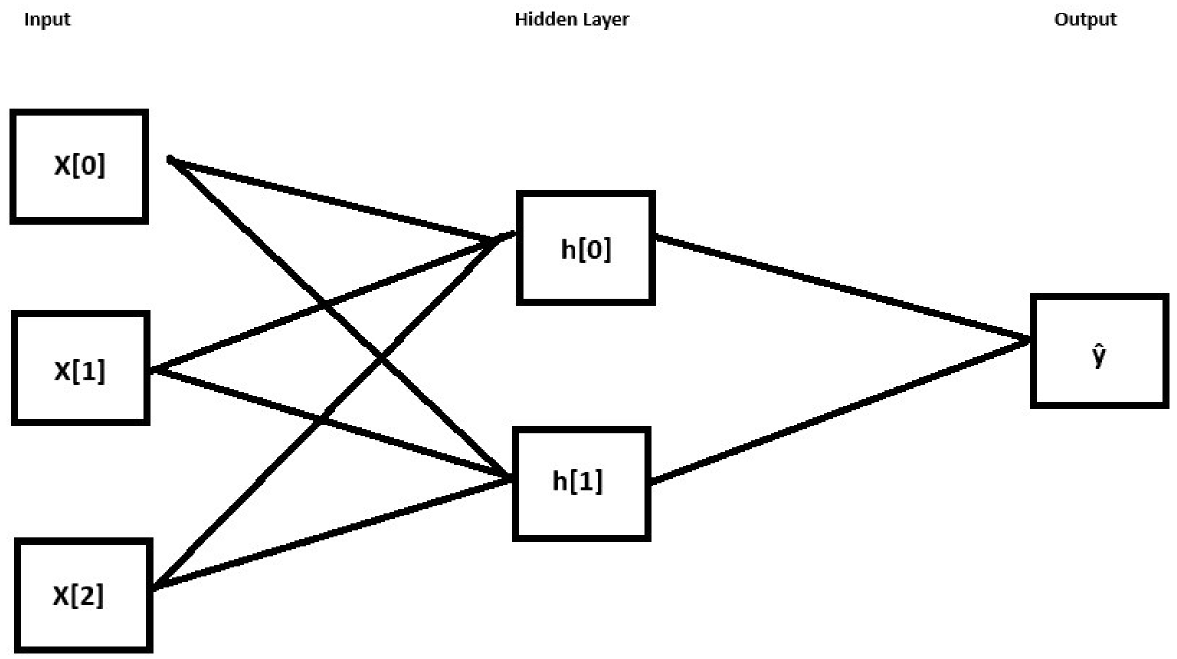


Figure 2.1

Illustration of CNN with a Single Hidden Layer

There are many different architectures and models from which to build a CNN, and various methods are mentioned in this paper, such as You Only Look Once (YOLO), Residual Networks (ResNets), and Very Deep Convolutional Networks (VGG). However, this paper focuses on the MobileNetV2 architecture.

MobileNetV2

The MobileNetV2 architecture is based on an inverted residual structure where the input and output of the residual block are thin bottleneck layers opposite to traditional residual models, which use expanded representations in the input. MobileNetV2 uses lightweight depth-wise convolutions to filter features in the intermediate expansion layer [2]. This allows for a smaller model size that can be created in a shorter time span while also allowing the achieved model accuracy to be competitive with more computationally expensive models [37]. MobileNet uses a bottleneck structure to reduce the computational cost by using 1x1 convolutions to reduce the number of channels before applying depth-wise separable convolutions [37,38]. A MobileNet adds additional layers before the classifier layer in typical CNNs. These new core layers are built upon separate depth-wise filters. These depth-wise filters are done through depth-wise convolution, allowing the model to shrink the hyperparameter's width multiplier and resolution multipliers [38,51]. The layered architecture of MobileNetV2 is shown in Figure 2.2.

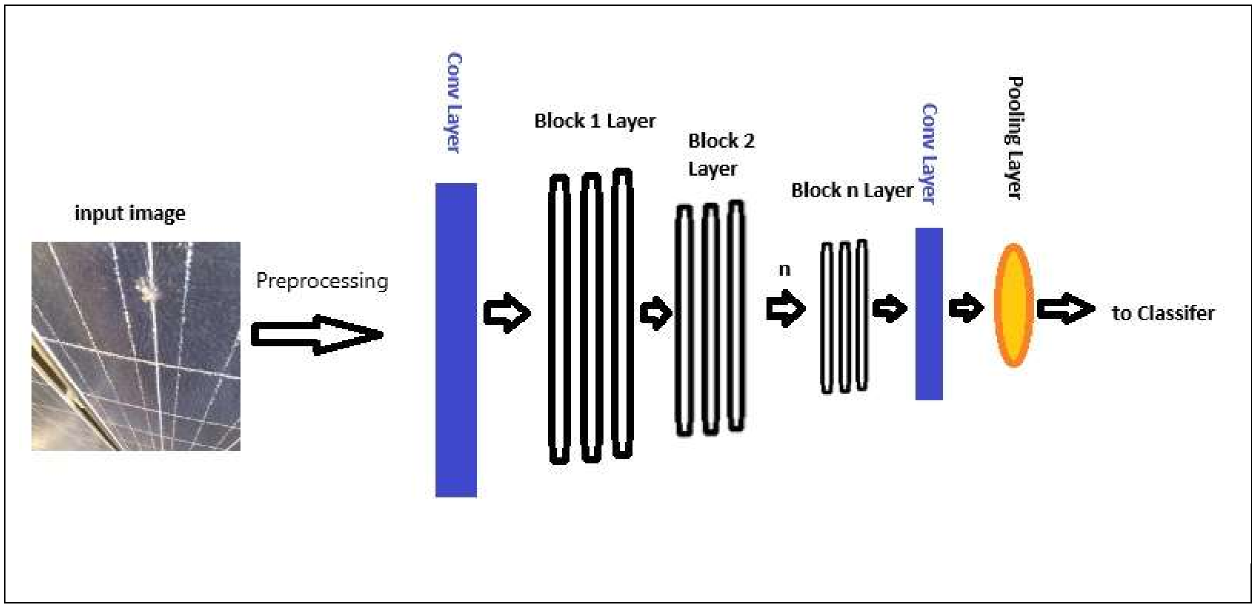


Figure 2.2

MobileNetV2 Architecture Sketch

Terminology

Some basic definitions of Machine Learning terms shall be relayed here.

- **Model**: The result of training an algorithm based on data supplied. The model is used to make inferences based on newly inputted data.
- **Batch size**: number of samples fed into the network algorithm at a time during model training.
- **Epoch**: One epoch is a single pass of batch data through the algorithm (neural network). After each pass, depending on the optimizer, weights to the algorithm will be adjusted for the next pass.
- **Optimiz**er: An algorithm to determine weight updates to the learning rates of the model during training. Traditionally, this is done using some form of stochastic gradient descent (SGD) procedure. However, newer algorithms like Adam use a combination of gradient algorithms and other methods. For instance, Adam uses the Adaptive Gradient Algorithm and Root Mean Square Propagation to determine weights effectively in deep learning models.
- **Categorical Cross-entropy**: Objective function of Keras, which defines a multiclass logarithmic loss. This compares the distribution of the predictions with the true distribution, with the probability of the true class set to a one and zero for other classes. If the true class is c and the prediction is y, then the categorical cross-entropy is defined as [19]:

$$L(c,p) = - \sum c_i \ln(p_i)$$

- **Accuracy**: The metric to evaluate the training of the model. It's the proportion of correct predictions concerning the total number of predictions.
- **Precision**: The metric used to evaluate the training of the model is the proportion of correct positive predictions concerning the number of correct and incorrect positive predictions.
- **Recall**: The metric to evaluate the training of the model is the proportion of correct positive predictions concerning the actual number of positive predictions
- **F1 Score**: The metric to evaluate the training of the model. It's the harmonic mean of the precision and recall values.
- **Softmax**: Layer type takes the feature as input and calculates the probabilities as outputs for each class. Then, select the highest probability score.
- **Rectified Linear Unit (ReLU) Activation**: a piecewise linear function that will output the input directly if the input is positive; else, it will output zero. Thus, if the input is negative, it gets converted to zero, and the inputted neuron does not get activated. ReLU is the default activation function for most neural networks.

PV Damage Types

There are many common types of PV damage; however, some more common effects are hotspots and broken glass [39]. Dust covering the PV and microcracks are also damage types affecting energy production.

- **Hotspots:** Defects that form small spots that dissipate the generated current in the form of heat [3].

- **Dust:** Dust is not just sand and dust particles; this can also be related to fallen tree limbs or any other obstruction that forms an obstruction between the panel and the Sun.
- **Microcracks:** tiny cracks that cannot be seen with the human eye.
- **Broken Glass:** As the name suggests, the PV's glass panel is damaged, preventing it from functioning properly.

K-Fold Cross Validation

With one test on model data, the algorithm could get lucky with the results and have great data; k-fold cross-validation allows one to train and evaluate a model multiple times to ensure the model acts as expected. The k is a variable that represents how many splits and tests one will do on the model's dataset. In general, the data set is split into k "folds." One of the folds will be selected as the validation dataset, while the remaining will become the training dataset. The model will be evaluated and trained on this data, and an evaluation score will be created using the validation dataset. Since this is a method to evaluate a model design, not a particular training, the fold data will report the average performance of the model [64]. Figure 2.3 shows a visual representation of the K-fold process for k, which equals four in this case.

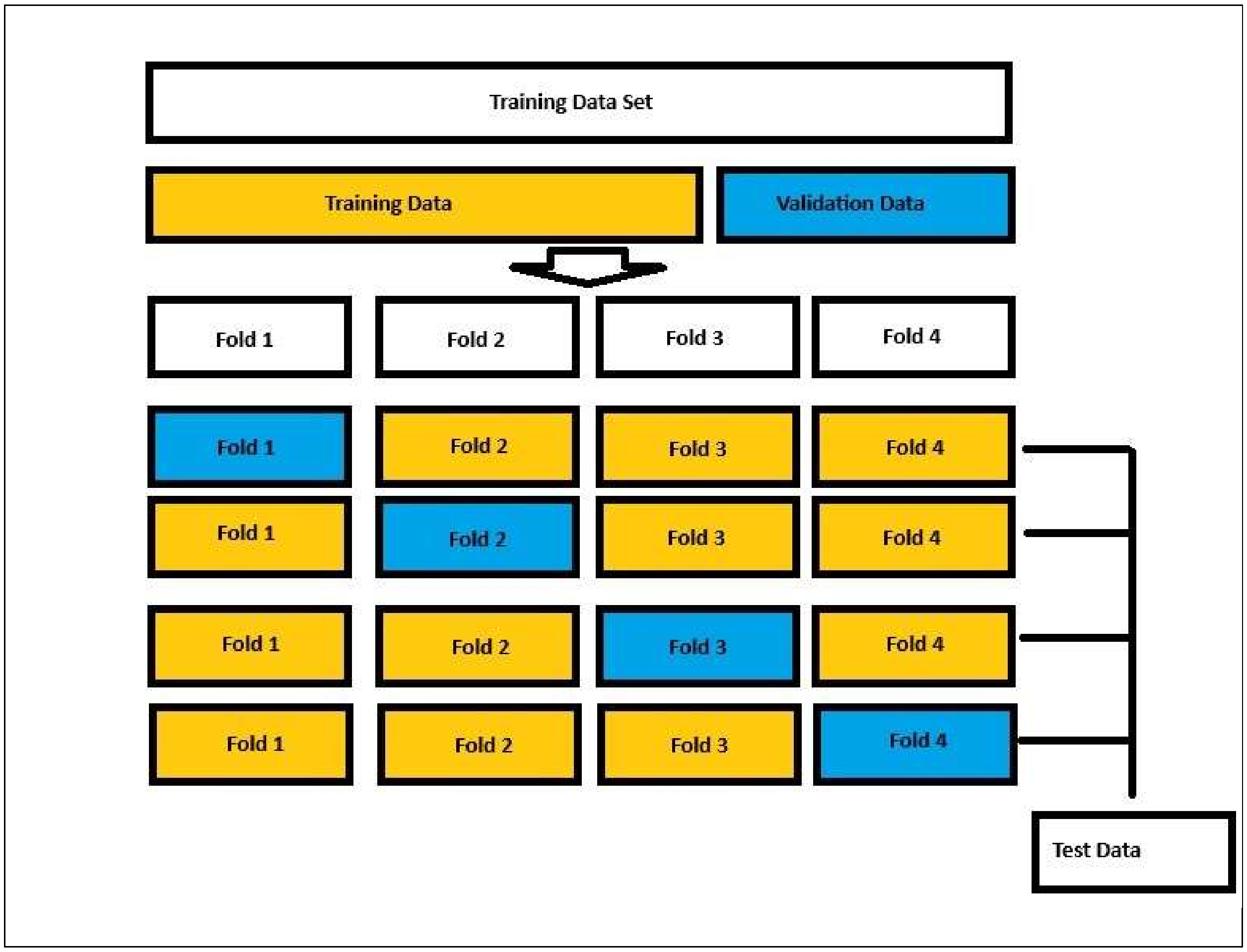


Figure 2.3

K-fold Cross-Validation k Equals Four Sketch

CHAPTER III

RELATED WORK

Chapter 3 explores works that have already been conducted that hold similar ground to image detection with deep learning. The following subsections discuss solutions and results of other lightweight machine learning algorithms.

Deep Learning with Solar Panel Damage

There are already a few approaches to solar panel damage with deep learning [3-29,40,41]. The authors of [5] used two databases organized by brand, an ambiguous damaged, and a good dataset for classifiers. One of these datasets was ten "mock" residential images, while the other dataset was 60,000 images. They used a Matthews Correlation Coefficient (MCC) to determine the accuracy and ran two experiments. First was a pre-trained approach using a CNN, where they had a 94.7% MCC accuracy. After this, they started retraining the CNN to achieve an approximate accuracy of 100% using the training data. The authors of [6] looked at damage to solar panels using EL images. They focused on intact panels, cracked, had intra-cell damage, solder issues, and oxygen bubbles. Their dataset consisted of 19,228 EL images comprised of 640x512 pixels. Then, the results of ResNet models (ResNet18,50,152) and a YOLO model were compared. The authors of [6] achieved an F1 score of 0.83 on ResNet18 and a 0.78 F1 score on YOLO. For the final model, ResNet was chosen based on this score. The authors of

[7] used MobileNetV1 to achieve an accuracy of 88.29% when classifying 45,469 augmented images with the following classes: clean, dust, cement, bird droppings, cracks, snow, soil, and shadows. Before augmentation, they had a total of 4,110 images. The authors of [27,29] reference issues in gathering images of damaged PVs as there is not much public data available, and both used image augmentation to generate results. The authors of [27] collected a total of 350 images and augmented them to create a total of approximately 60,000 images. Using TensorFlow and homemade code titled SolarDiagnostics, they classified shaded, dust, snow, and good PVs with an accuracy of 85.9%. Others, like [28], point out that a wide array of camera and noncamera options are needed as conditions can result in areas where one option cannot be used. They used EL and IR images to detect microcracks, dirt, hotspots, and bird droppings with an accuracy of 99.8%. This was achieved using Kera's deep convolutional network, Visual Geometry Group 16 (VGG16). The authors of [13] used 44 EL images augmented up to 2,624 images. The two classes used were good and defective. Using a VGG6-based CNN, the authors of [13] created their model using cross-validation with a fold size of four. This resulted in an average accuracy of 93.025%. The authors of [18] tested two different CNN's, a YOLOV4 and a ResNet, for model creation. These models were trained on IR images. They used a dataset of 3,582 augmented 640x512 pixel images to determine if a panel had hit a segmentation threshold or had a hotspot. The YOLOv4 achieved an average accuracy of 99.01%, while the ResNet model achieved an accuracy of 99.23%. The authors of [34] specifically use infrared to identify cells with hot spots or areas with higher temperatures than other solar panel areas. In contrast, the authors of [7,10,24] use a mix of traditional imagery and infrared images to classify not only PV's. The dataset [24] used comprised 1,136 images before augmentation; each of

these images was converted down to 336x256 pixels and to greyscale before training on the data. Ten classes were used: discoloration, delamination, corrosion, glass crack, IC failure, hotspot, cell crack, permanent soiling, good, and back sheet damage. The authors of [24] used multiple different models to train on the data, which resulted in 87.3% accuracy. However, no loss data was presented. Authors of [15] used Tensorflow and Keras to build their CNNs and tracked the accuracy with different model optimizers. With 600 IR images, the authors of [15] achieved accuracies of up to 98.93% with SGD and 94.52% with Adam on the IR images to detect damage; they also claim that the model with the Adam optimizer could get close to 100% at detecting hotspots. While the exact number of each image class was not identified, they had the following categories: normal, dirt, vegetation, hot spot, multi-hot spot, open circuit, cell crack, and DC box component. The authors of [24] had classifiers of clean, dust, cement, shadow, bird droppings, crack, snow, and soil. Shadow-covered PVs have a total of 56 images, the largest image class consisting of 1,204 images of dusty PVs, and a dataset comprised of 4,110 images in total. After this, they used image augmentation to generate a dataset of 45,469 images and achieved 94% accuracy with their proposed model. The authors of [25] had a dataset of 1,100 images and three classes: normal, damaged, and dusty, achieving an accuracy of 93.7% using an SGD optimizer. The image sizes and normalization process were not provided.

Deep Learning on Edge Devices and Microcontrollers

Deep learning on edge devices has seen a lot of attention in the last seven years. High-performance model training is still mainly done on state-of-the-art computers due to resource consumption [42]. However, the option of implementing and using trained models on much less resource-intensive devices, such as microcontrollers and mobile devices [43] or even small GPUs [42], is quite incentivizing. Deep learning on edge devices has seen large swaths of use cases, from UAVs to many other applications. The authors of [22] designed a small CNN model using Tensorflow-Lite and Keras to develop a real-time fault detection system for PVs. The authors of [22] focused on five class labels: healthy, dirty, degraded, sand deposit, and overheated junction box. They had 670 IR images that were augmented up to 5,786 images. The smallest image count was "Overhead Junction Box" at 90 images. This was augmented up to 877 images. At the same time, the largest image pool of 150 dirty images was augmented to 1,425 images. Running their model on the Arduino Nano 33 BLE Sense, [22] achieved an accuracy of 94.8%. The authors of [44] have an edge device (Raspberry Pi 4) that can discover anomalies that could be malicious traffic on a network. The authors of [45] detail the challenges of developing edge cases to train the model on the device instead of using a state-of-the-art computer, which may be useful for updating the model on the fly. There are also methods to update and train a model through over-the-air updates [26,45].

Deep Learning with Unmanned Aerial Vehicles

There are various applications for using Machine Learning devices while flying UAVs (more commonly known as drones). This ranges from using still images to real-time image detection to Multilabel object detection [4-25,46,47,48]. There are also more precise applications like counting corn [40] or even determining cars via segmentation techniques. In the case of PV analysis, the way these images are typically handled is a UAV will take pictures of an area, then in some ways are transferred to a computer for classification [4-20]. For instance, the authors of [8] used Bluetooth to transfer the images from a UAV to their CPU. The authors of [20-23, 25, 46] have been able to do classification on an embedded system that is mounted on a UAV. The authors of [11] used a K-means classifier on a dataset of 4,969 RGB images and 4,143 IR images taken from UAVs. The authors of [11] focused on whether there was a faulty panel or if a panel was good, with an accuracy of 96.1%. The authors of [12] used UAVs to take and transfer images over 4G. They had a dataset of 6 kinds of images: 100 encapsulant delaminated, 100 encapsulant delaminated, 100 broken glass, 500 dust, 200 snail trail covered, and 2,000 good images. Their model achieved an accuracy of 78.61%. The authors of [14] used a dataset of 500 IR images, 63 of which had hotspots, 15 of which had bypass defects, and the remainder of which were good. These images were taken at 640x512 pixels and were used to form two classes, good and defective. Achieving a 98.47% with just a binary classifier. The authors of [17] used a VGG16 CNN to detect if bird droppings were on PVs. About 1,000 640-by-480 pixel images were taken by UAVs to train this data. The authors of [17] did not break down how many images were good and how many had bird droppings. The calculations of a percentage of PVs had bird droppings from the zones the images were taken from. The number of images per zone was not given. The authors of [19] look into using YOLOV3 as its CNN classifier and apply it to two different datasets to see if they can get comparable results. The first dataset consisted of 2,038 640x512 pixel IR images that looked to classify heated joints and hotspots in PVs, while the second dataset consisted of 1,500 1,600x1,200 images. The second dataset focused on soiling, strong soiling, raised panels, delamination, puddles, and bird droppings. This dataset had a breakdown of 428 strong soiling images, 475 raised panels, 525 delaminated panels, 1060 soiling, 3403 puddles, and 4095 bird-dropping images. For both these datasets, image augmentation was applied, though the number of images is not stated. In their first dataset, an accuracy of 66.9% is achieved, while 68.5% is achieved with their second dataset. The authors of [20] compared the performance of three different models designed on the YOLOv3 classifier on different hardware targets. The use case used a UAV equipped with onboard hardware to label the antenna, insulators, and vibration dampeners of power systems in real-time. They focused on which device and optimized model would produce the most efficient real-time data. They measured how many frames their hardware could process per second. In the end, the authors of [20] determined the YOLOv3 optimized model on the Nvidia Jetson AGX Xavier resulted in the best accuracy while processing fifty 288x288 pixel images per second. The authors of [21] continue to follow the real-time approach using IR cameras to detect cell, module, or panel faults. One-hundred and sixty-two images were used to train a YOLOv3 model on a Nvidia Jetson TX2. A breakdown of 54 images shows the model and 36-panel faults, and the rest show the device operating normally. Using these three classes, the model achieved an overall accuracy of 95%. The authors of [23] were another IR-focused PV expansion method that had 12 classes: cell damaged, multi-cell damage, crackling, diode damage, multi-diode damage, hotspot, multi-hotspot, offline module, shadowing, spoiling, vegetation, and good. The authors of [23] reference that most models up to the point of writing had a dataset of a range from 110 to 900 IR images, so they acquired 20,000 IR image data sets. This dataset had a wide range of images for each given type, as there were only 175 images for the multi-diode damage, while single-cell damage had 1,877 images. Data augmentation was used to increase the number of images, but the amount of this is not explained in the document. However, the authors mention that the images are normalized to 40x24 pixels before training. They also noted that for future models, it would be wise to augment some images to solve for imbalanced classes of the model. The final model ended with an accuracy of 85.4%. The author of [46] used a real-time system using MobilenetV1 to determine if concrete had been damaged to significant effect, having an approximate 95% accuracy on damage detection, which has some very similar methods to what happens further in this paper.

Summary of Literature

Overall, most of these documents cover a portion of the work carried out in this thesis. Table 3.1 compares what some of these documents cover versus this thesis. This thesis achieves a power-efficient multi-image, damage classification model that is on board a UAV.

Table 3.1 Literature Comparison

Ref No	K-Fold /Cross Validated Model	Multi Image Type	Single Model	Onboard Classification	Multiple Device Comparison	Power Usage (flight time)
[27,13]	X		X			
[15]	X	X	X			
[28,7,11,12,19,24]		X	X			
[5, 6, 9, 14, 17, 18, 29]			X			
[20]			x	x	x	
[21-23]			X	X		
[24]		x				
[10]	X	X				
[25]			x	x		
Thesis	X	X	X	X	x	X

CHAPTER IV

METHODOLOGY

Model Implementation

Dataset

For this work, a mix of provided and internet-found images (using keywords broken, cracked, microcracked, hotspot, dusty, solar panel, and PV) were used. The original dataset had no qualifiers when gathering data. For instance, one of the internet-gathered datasets of images was a set of "good" and "dusty" PV's from the image data set repository Kaggle [49,50]. Another dataset [49] contained a total of 2,562 images, 1,493 of which were deemed "clean" while the other 1,069 were considered "dusty," though dusty appears to cover bird droppings and other debris, not just dust. In contrast, the dataset provided by [50] contained 199 bird-dropping, 202 clean, 220 dusty, 98 electrical damaged, and 66 physically damaged PV Images. Both datasets contained images from a wide array of different angles. Since a UAV will not always have a consistent image angle on PVs, it was thought that additional angles could help the model recognize panels. However, the PV data from these datasets had a few perceived abnormalities, such as people in photos, ads built into the photos, markings depicting the damage, and watermarked images. Figure 4.1 depicts a few of these removed outliers. A model was created with these abnormalities included and removed, with results shown in Table 5.2. Different angles and aerial views were left since these would most likely be present from drone images. Other online images provided "microcrack" panels (which were captured with UV Cameras), "hot spot" panels, and "broken glass" panels. Figure 4.2 shows an example of each image category, and Table 4.1 shows the number of images after abnormalities were removed and before augmentation.

Figure 4.1

Outliers Removed from Dataset

Table 4.1 Number of Each Image Type

Good	Dusty	Hotspot	Broken Glass	Microcracked
1000	950	94	94	10

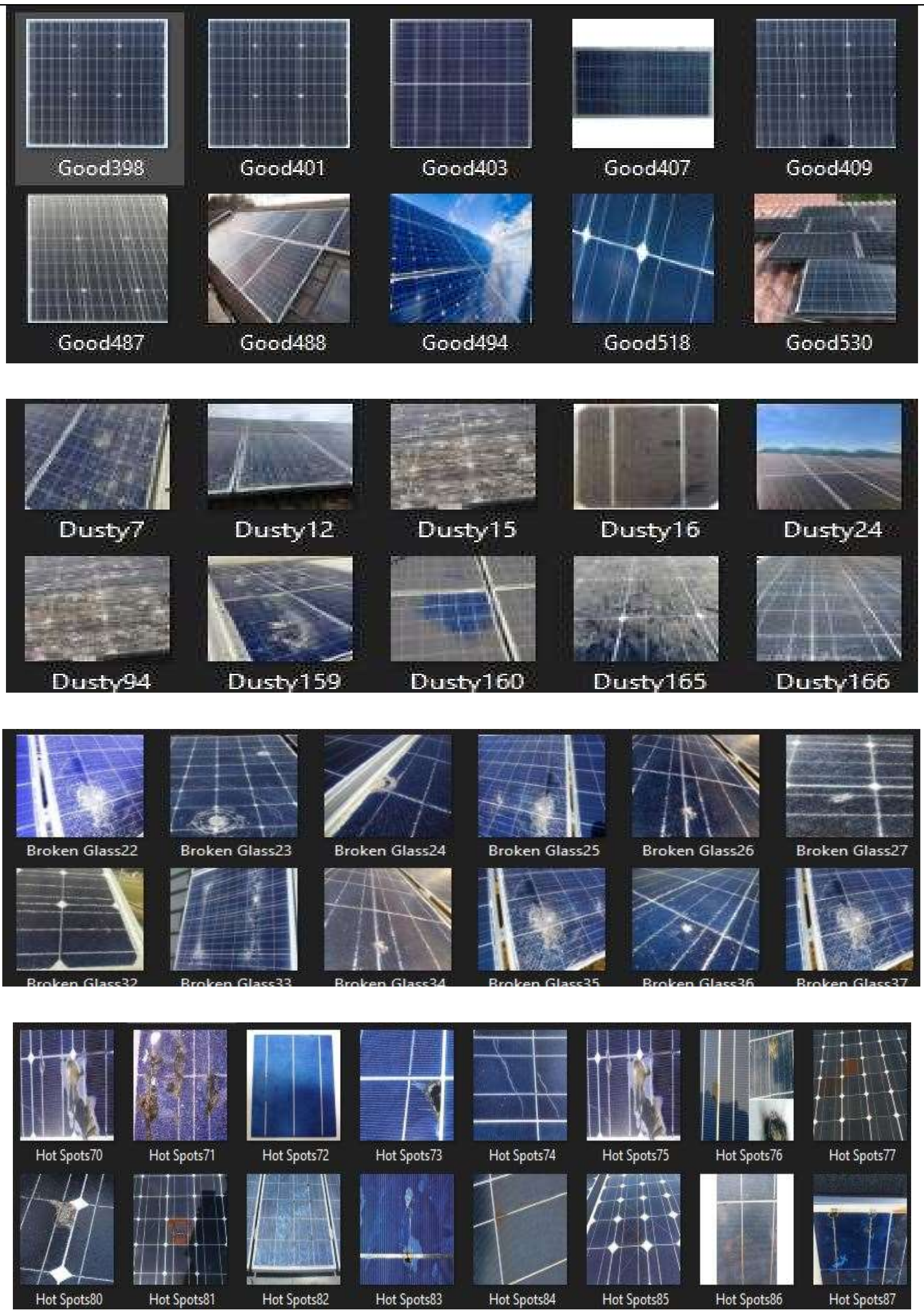


Figure 4.2

Collection of Sample Images

The number of dataset images for the different types of damage varied. Thus, augmentation of the image data pool was done to expand them. This augmenter code brought the image pool for each datatype to 3,000 images. The augmenter code was created with the Augmenter Python package and works as follows:

- Take an image from the base pool.
- Based on a probability, do at least one of the following: flip the image horizontally, flip the image vertically, rotate the image a maximum of five degrees left or by a maximum of ten degrees right, skew the image in one of twelve directions (Figure 4.3), and/or zoom into the image by a minimum factor of 1.1X zoom or a maximum factor of 1.5X zoom.
- Save the image into a new folder to be used for model training.
- Repeat the process until the desired sample size is reached.

A dataset of about 500 was initially used to speed up the process of original prototyping. These data types were split in a 70:15:15 ratio for training, validation, and test datasets.

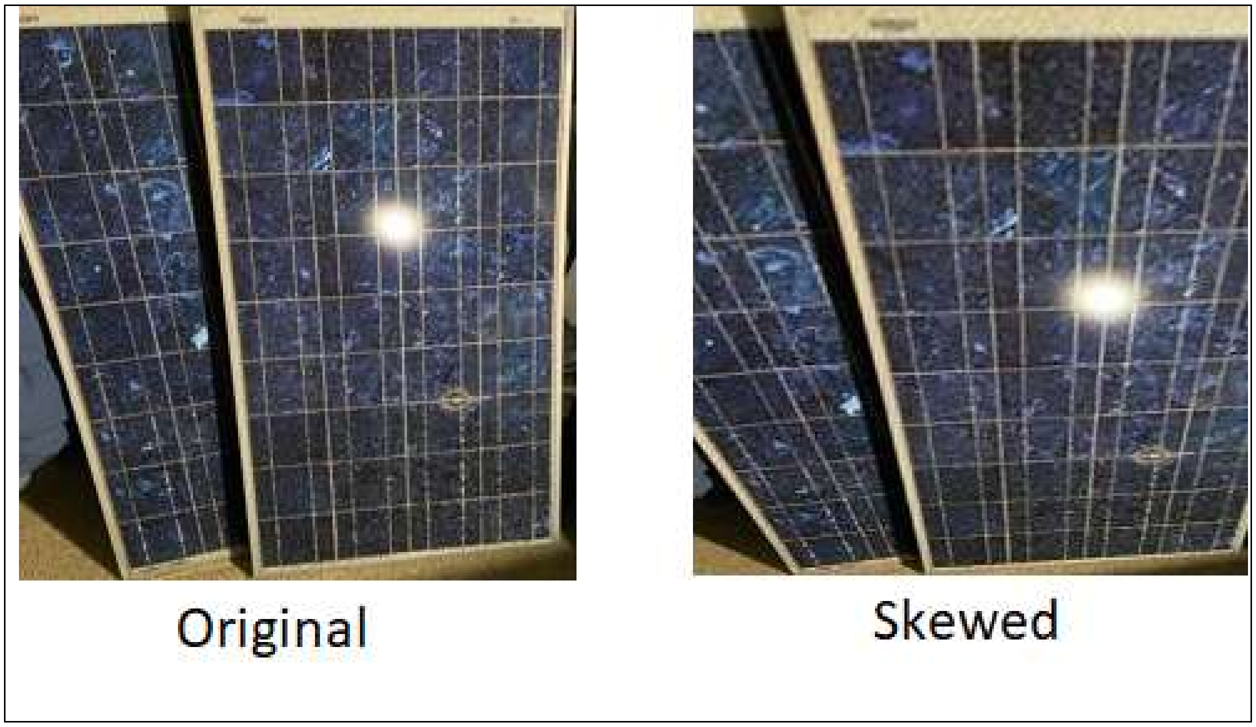
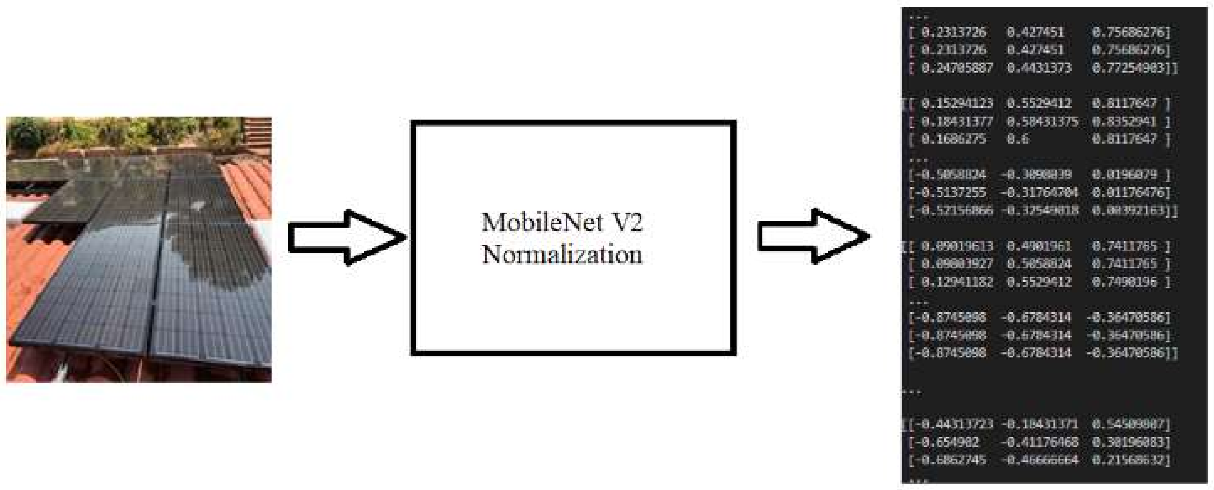


Figure 4.3

Skewed Image Example

Preprocessing

There was a wide array of image resolutions and sizes on which the model was to be trained. The largest size being 5,472 x 3,080 pixels and the smallest size being 389 by 258 pixels. To reduce model size and provide uniformity, all images were preprocessed. First, all images were resized to a 224 x 224-pixel square. Distortion of the images was not factored in. After this, each image was normalized on a scale from negative one to one. Figure 4.4 shows a normalized image's basic input and output to the human eye.



```
...
[ 0.2313726   0.427451    0.75686276]
[ 0.2313726   0.427451    0.75686276]
[ 0.24705887  0.4431373   0.77254903]]

[[ 0.15294123  0.5529412   0.8117647 ]
[ 0.18431377  0.58431375  0.8352941 ]
[ 0.1686275   0.6         0.8117647 ]
...
[-0.5058824  -0.3098039   0.0196079 ]
[-0.5137255  -0.31764704  0.01176476]
[-0.52156866 -0.32549018  0.00392163]]

[[ 0.09019613  0.4901961   0.7411765 ]
[ 0.09803927  0.5058824   0.7411765 ]
[ 0.12941182  0.5529412   0.7490196 ]
...
[-0.8745098  -0.6784314  -0.36470586]
[-0.8745098  -0.6784314  -0.36470586]
[-0.8745098  -0.6784314  -0.36470586]]

...

[[-0.44313723 -0.18431371  0.54509807]
[-0.654902   -0.41176468  0.38196083]
[-0.6862745  -0.46666664  0.21568632]]
...
```

Figure 4.4

Normalization of an Image

Neural Network Construction – TensorFlow

The initial model was built upon TensorFlow version 2.8 code base using Keras. Keras was used with TensorFlow since its primary purpose is to focus on debugging speed, code elegance, conciseness, maintainability, and deplorability [56]. It is a useful interface with TensorFlow and has many tool sets, allowing users to design different CNN models easily. For the classification model, *CNN MobileNetV2* is used. The Keras code base was used to implement the MobileNetV2 normalization in the model. Since image classification methods are the target use case, a two-dimensional convolutional layer (Conv2D) was used. The final model consists of the following: an Input Layer, sixteen Block layers, a Dense layer, and an Output layer. Each of the sixteen block layers is then broken up into subsections, which do various processes to help the model determine classification and then pass the data to the next sublayer. Figure 4.5 and Figure 4.6 show what one block of data can look like for a given image. This heatmap is a bit of code that allows humans to visualize what features the model visualizes. There are five distinct categories; thus, categorical labeling was used. The model will operate while using SoftMax activation.

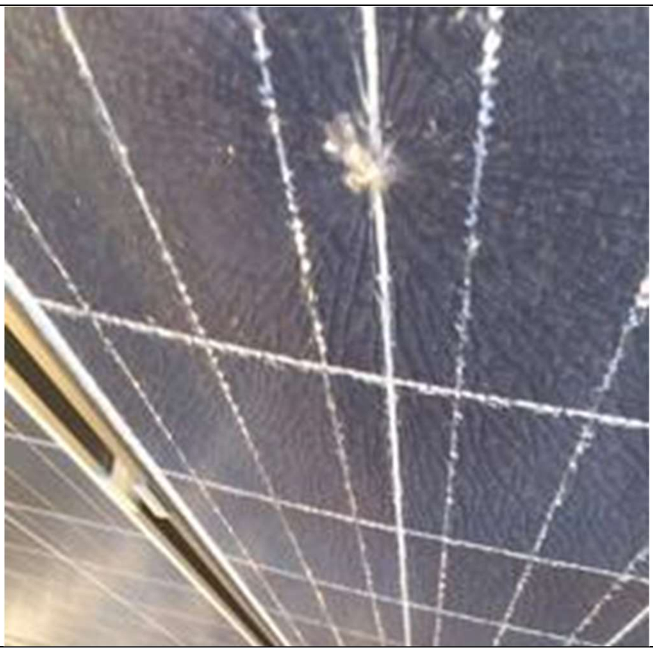


Figure 4.5

Image to be Modeled

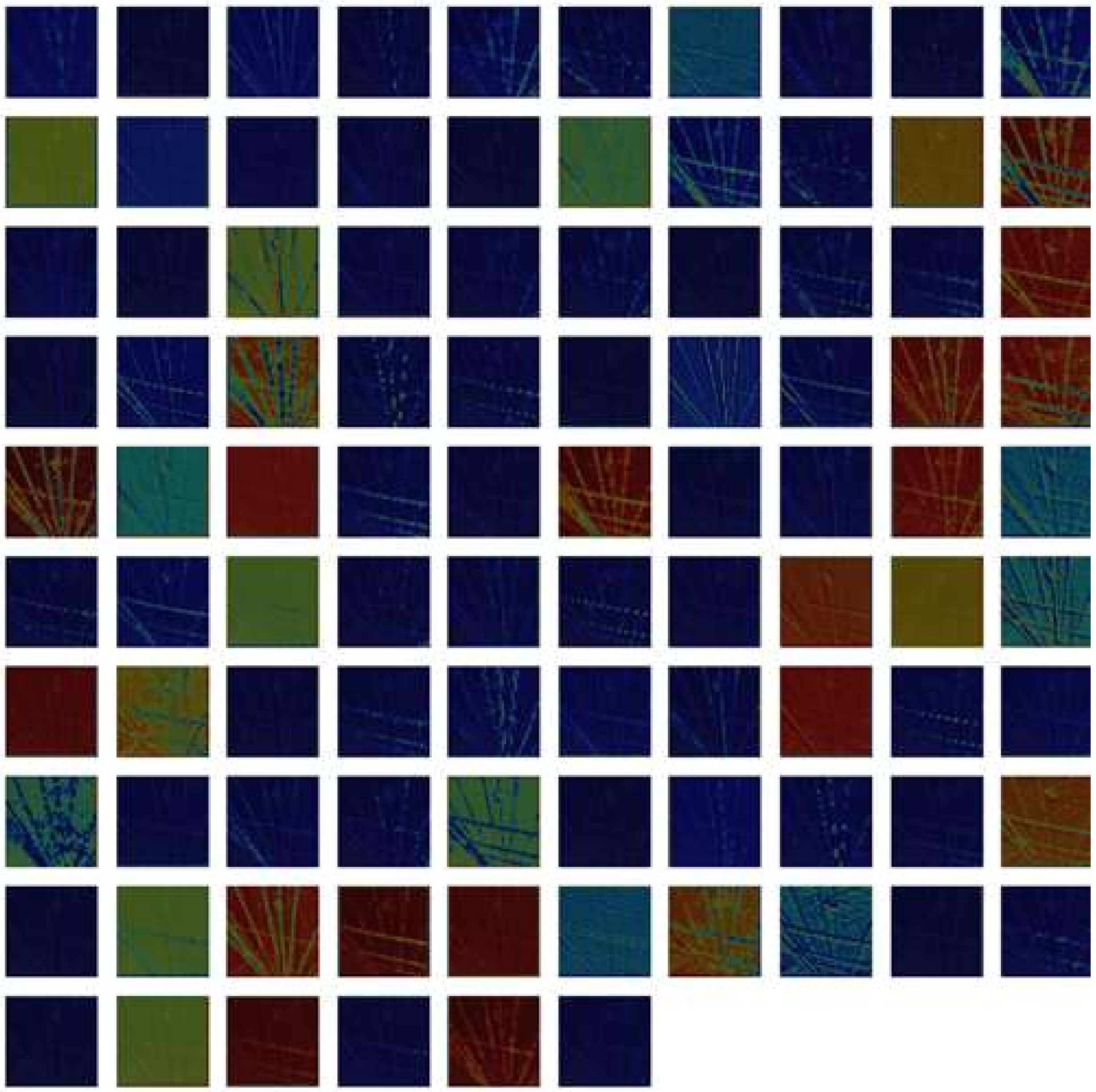


Figure 4.6

Block One Heatmap

The Adam optimizer was used to create the model, and categorical cross entropy was used as the objective function.

TensorFlow Model Testing

A set of programs were used to create and test the model, as shown in Figure 4.7.

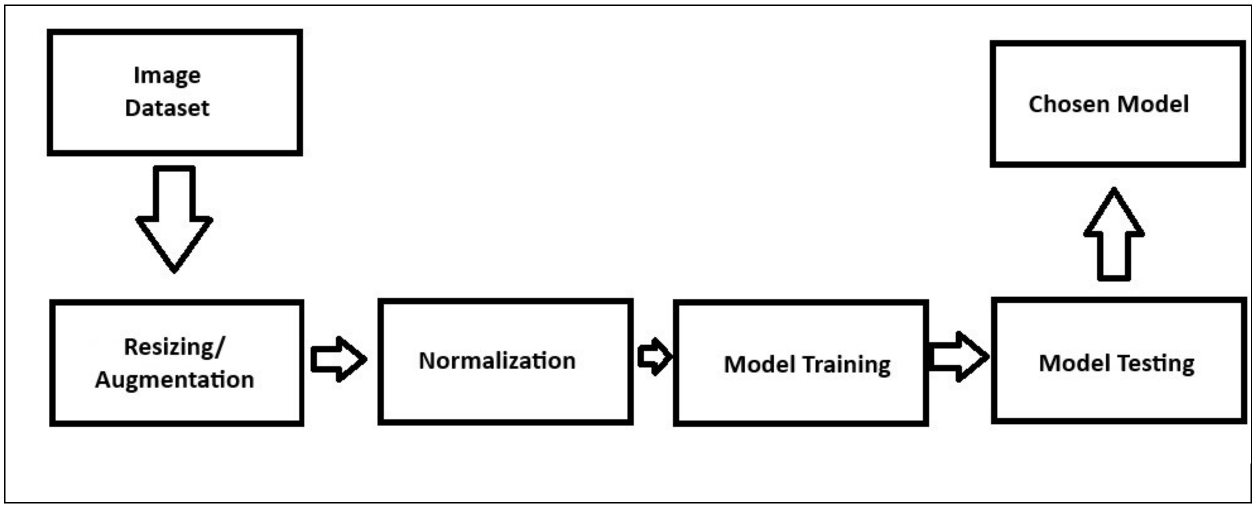


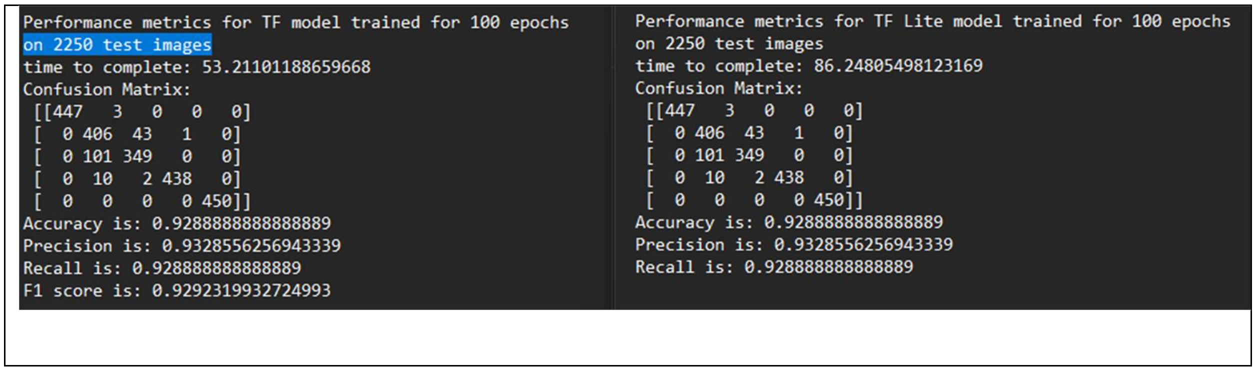
Figure 4.7

Program Flow for Model Creation and Testing

The general model creation and training process started with a low epoch number of five and a smaller image dataset (500 images). This was to prototype a model quickly and to start working on a good set of programs that could interface with the resulting model. As stated earlier, the image data was proportioned into three categories: 70% for training, 15% for validation, and 15% for testing. The model training code uses the training and validation data sets to create and train the model. After one complete pass of the training dataset (an epoch), the model is validated using the validation dataset. The model is updated based on the validation results, and another epoch is run using the updated weight values to improve model classification accuracy. For the prototyping, this process of adjusting epochs and training the model continued until an accuracy of at least 80% was achieved. After prototyping, the goal was to achieve an accuracy within the range of 90% and 95%. The model is stored for subsequent testing once an accuracy between 90% and 95% is achieved. The model was tested using the remaining test image data set and the SKlearn python library. The SKlearn library allowed the model to collect new data and get the accuracy, precision, recall, and F1 scores based on a supplied grand truth table. The output of this can be seen in Figure 4.8.

TensorFlow-Lite

After the TensorFlow model had been set up and had an acceptable accuracy of above 80%, this model was then converted into a TensorFlow Lite applicable model. TensorFlow Lite is the mobile library for deploying models on mobile devices, microcontrollers, and other edge devices [43]. This was done since TensorFlow Lite is better optimized for less powerful hardware and to save on file space. Like the testing process for the TensorFlow model, both programs achieved the desired accuracy value(s) and used the SKlearn Python library and a ground truth table to calculate the numbers presented in Figure 4.8. There was no apparent hit to the models, accuracy, precision, or recall values at the cost of converting to a Tensor Flow Lite model.



```
Performance metrics for TF model trained for 100 epochs
on 2250 test images
time to complete: 53.21101188659668
Confusion Matrix:
 [[447   3   0   0   0]
 [  0 406  43   1   0]
 [  0 101 349   0   0]
 [  0  10   2 438   0]
 [  0   0   0   0 450]]
Accuracy is: 0.9288888888888889
Precision is: 0.9328556256943339
Recall is: 0.928888888888889
F1 score is: 0.9292319932724993
```

```
Performance metrics for TF Lite model trained for 100 epochs
on 2250 test images
time to complete: 86.24805498123169
Confusion Matrix:
 [[447   3   0   0   0]
 [  0 406  43   1   0]
 [  0 101 349   0   0]
 [  0  10   2 438   0]
 [  0   0   0   0 450]]
Accuracy is: 0.9288888888888889
Precision is: 0.9328556256943339
Recall is: 0.928888888888889
```

Figure 4.8

TensorFlow vs TensorFlow Lite

Hardware Targets

The model building and original testing happened on a Windows Personal Computer. The code was then transitioned to a Raspberry Pi 4 and Nvidia Jetson Nano for further testing and experiment implementation. Each hardware requires different dependencies, and software version builds are needed to get this working. The setup for the two devices is explained further in Chapter VI. Both devices were chosen for their ease of access and ability to scale down the actionability of smaller hardware use. The two devices also provided a platform to compare a system that will do machine learning completely on its CPU (Raspberry Pi 4) and one that will offload some work onto its GPU (Nvidia Jetson Nano).

Raspberry Pi 4

The main implementation of the model was managed via the Raspberry Pi 4. The Pi 4 is built around the 64-bit Broadcom BCM2711BO quad-core A72 with 4GB LPDDR4 SD RAM [52]. Additional peripherals were a Raspberry Pi Camera Module 3 for the camera and a Samsung 128 Evo Plus Memory card for memory storage. Figure 3.9 shows this configuration.

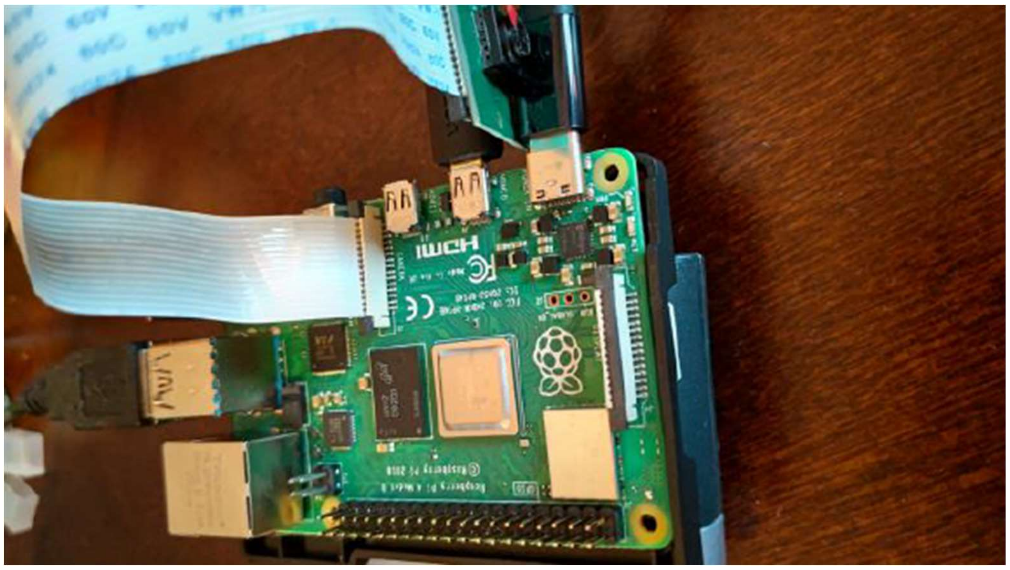


Figure 4.9

Raspberry Pi 4 Dev Unit

Nvidia Jetson Nano

The Jetson Nano is built around an ARM Cortex A57 MPCore processor and NVIDI's Maxwell architecture with 128 CUDA Cores. This system also has 4 GB of 64-bit LPDDR4 RAM [53]. Additional peripherals were a SanDisk Extreme Pro 256GB SD card and a Raspberry Pi Camera Module 2. The Pi Camera Module 3 could not be used at the time of writing because it was not supported. Figure 4.10 shows the layout of the dev kit. NVIDIA develops CUDA, a parallel computing platform and programming model designed for general computing on GPUs [54]. Nanocam was the most readily available camera codebase for the Jetson Nano. It was used to control and take pictures.

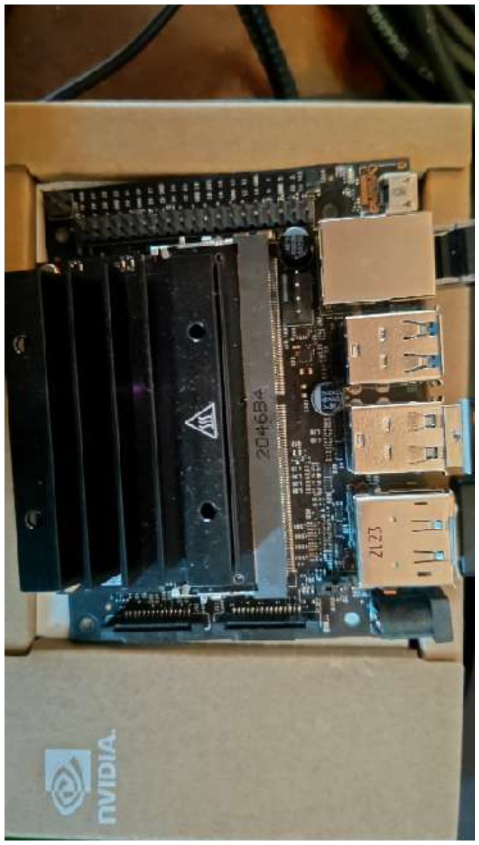


Figure 4.10

Jetson Nano Dev Kit

Since TensorFlow-Lite and TensorFlow both have CUDA processing support, there could be potential advantages it could bring compared to the Raspberry Pi. Implementing CUDA support was a simple addition since the codebase easily checks if it can be used, so implementation was straightforward. Both bits of hardware allowed for effective ways to start down-scaling hardware and implementing the model. It is worth noting that while the original software target for the hardware was TensorFlow 2.15 (which is supported on the Raspberry Pi), the latest version of TensorFlow that works with the Jetson Nano is 2.7 at the time of writing. A drop to TensorFlow 2.6 was required to have everything operational. Other software dependencies had to use different version numbers to become operational. Appendix B shows the general installation and setup process for the dev kit.

Model Implementation on Hardware

Once a satisfactory model (80% accuracy or better) was created, it was applied to the two target hardware platforms. A program was designed to read all images from a file folder (this came from the data split and is the test data), classify the images, and place each image into the model's correct folder. Each of these folders represents one of the five classes of PV types and contains organized images. To help with future model training, it would note and then store a copy in another folder. The code determines the closeness of two probabilities by looking at the two highest probable classifications. After this, it subtracts the lowest of the two options from the highest. If this remainder is under 58%, the code determines the model was close to picking one of two classifications. Since there were two likely classification candidates, whether the image was chosen correctly or not, it is important to earmark this image as it could have value in future iterations of training the model. Upon further review, these images could be added back into the next model training session, allowing for a more robust model.

Model Improvement and Additional Evaluation

Once hardware was tested and results were gathered, k-fold cross-validation was applied to the model. This was done to better, unbiasedly understand how the model was being implemented. Based on the results of k-fold cross-validation, additional changes to model training and image dataset choices were made. These were as follows:

- Reduce the image pool for each category to the same number of images (94) before augmentation.
    - A random number generated selected these images to reduce the chance of human bias.
- Try different numbers of augmented samples.
- Look at the effect of adding and removing different categories of images.
- Use a low number of epochs for rapid prototyping.

All of these were done to help remove bias and improve the model.

CHAPTER V

RESULTS

Model Testing and Creation

Each model is created with a five-class classification using categorical cross-entropy loss, a batch size of 32, and the Adam optimizer with a learning rate of 0.001. The initial model was created based on only five epochs, yielding a decent accuracy rating of 87.29%. The model ended up with 154 MobileNetV2 layers. A generalization of these layers is shown in Table 5.1.

Table 5.1 Model Block Layer Breakdown

Layer	Description
Input Layer	The base layer takes in the input
Convolution layer	Takes an input, does batch normalization, and then a ReLU activation passes to the next block.
Blocks 1 through 16 (bottleneck layers)	Filters out nodes from the previous layer to obtain a representation of the input with reduced dimensionality. [37,38]
Convolutional Layer	Takes an input, does batch normalization, and then a ReLU activation passes to the next block.
Global average pooling 2d	Structural regularizer to help prevent overfitting for the overall structure of the model. [65]
Drop out layer	Removes nodes from the NN to help reduce overfitting.
Dense layer	Layer that receives output from every neuron from the previous layer. Calculates the dot product of the input and the neuron weights.

During the training process, it was realized that the good and dusty databases had images with the same watermarks. There was worry that the model could perceive these watermarks as important features, adding an unwanted bias. The database was handpicked to fix this, and all watermarked images were removed. The model was then recreated, this time without watermarks. There was roughly a 2.2% accuracy increase and a 2.5% F1 score increase from removing all the watermarked images. With the proof of concept finished, a final pass was done. This last pass created the final model using the same classification and optimization methods, but this time with a validation accuracy process of one hundred epochs. This tightened the accuracy to 92.89% and the F1 score to 92.92%. (Table 5.2).

Table 5.2 Comparison of Model Tests based on Epoch and Watermark Removal

Epochs	Accuracy	Precision	Recall	F1 Score
Five w/ Watermarks	87.29	88.21	87.28	86.86
5 w/o Watermarks	89.51	89.86	89.51	89.37
100 w/o Watermarks	92.89	93.29	92.89	92.92
200 w/o Watermarks	92.31	92.67	92.31	92.36

After creating a model of acceptable accuracy (aiming around 90%), the TensorFlow model was converted into a TensorFlow Lite model. The model dropped from 12.2 MB to 8.47 MB in size.

Analysis of Model Implementation on Hardware

The program designed for analysis reads all images from a file folder and places them into what the model deemed the correct folder. If an image had two areas close in tolerance, it would note and store a copy in another folder. This process worked because the code would take the two highest probable categories and subtract the most probable from the second most probable. If this number was less than 0.58, the two probabilities were close enough that the image might not be sorted correctly. For instance, Figure 5.1 – Dusty Solar Panel Deemed Good had a probability difference of 0.087693125, meaning the image might not be sorted correctly. The model claimed a probability of 4.5615301e-01 for being a dusty solar panel and a probability of 5.4384613e-01 for being a good solar panel.

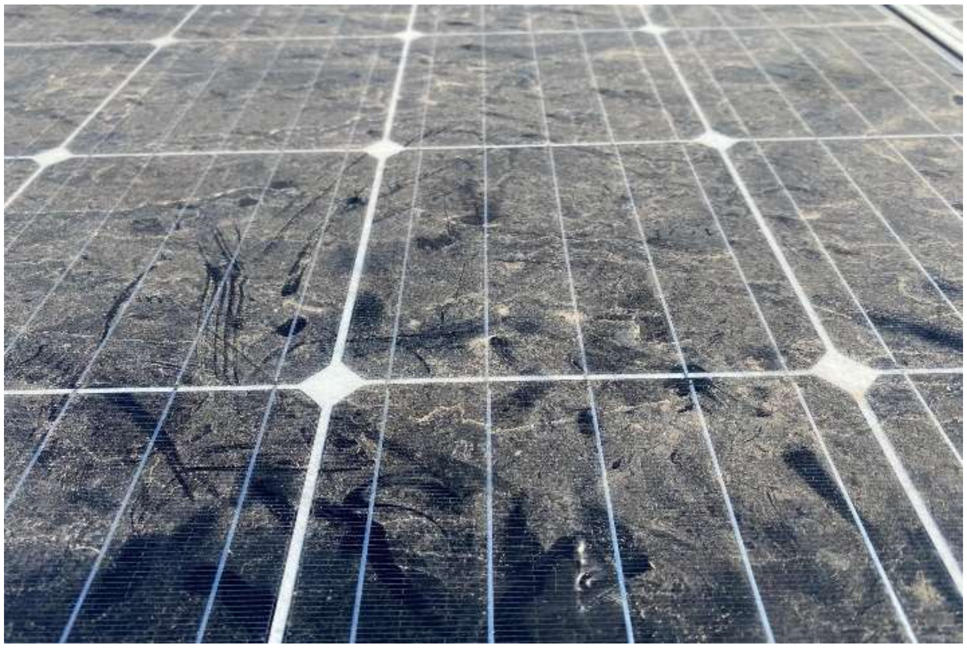


Figure 5.1

Dusty Solar Panel Deemed Good

While the image limit on the PC was not hit, the original image run of 2,250 images was beyond the limit for the target hardware. Thus, the image count was lowered to 1,576 for large-scale image testing as this number could still be allocated into one array in memory for the Raspberry Pi and Jetson Nano. Figure 5.2 shows a block diagram of how the code functions.

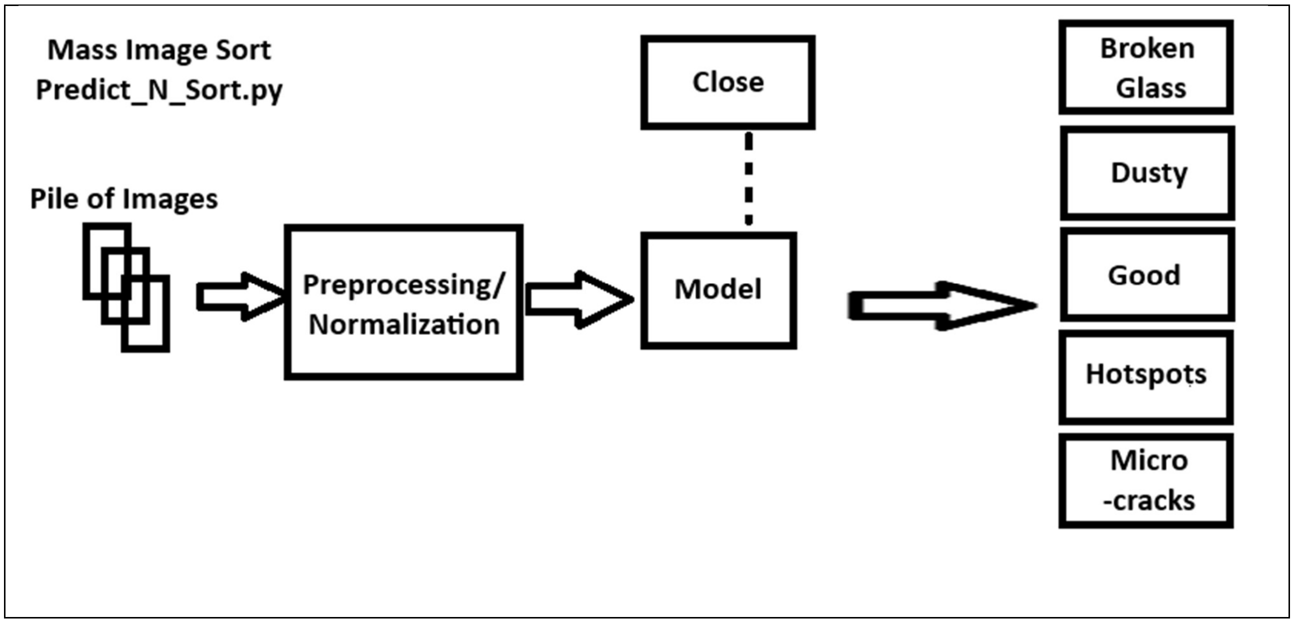


Figure 5.2

Block Diagram of Predict_N_Sort.py

Each run had the same images and was sorted into categories for each run. The code running on the Raspberry Pi with the Raspbian OS took the longest average processing time, 260.40 seconds (about 4.34 minutes) or 6.05 images per second. The Raspberry Pi running Bookworm processed and classified the images in approximately 129.1 seconds or 12.2 images per second, a substantial increase. The Jetson Nano processed these images at an average of roughly 195.9 seconds or 8.04 images per second. These times and values are the average calculated across ten runs, covering image selection, preprocessing, normalization, and classification (Table 5.3). The Raspberry Pi completes all these tasks in roughly 66.78103 seconds, faster than the Jetson Nano.

Table 5.3 Predict_N_Sort.py Time Trials

	Pi 4–- Raspbian	Pi 4–- Bookworm	Nano
Seconds (s) per run	253.4199	129.2129	200.5513
	250.4071	129.0781	198.8279
	251.1060	129.0683	196.7776
	247.9717	128.9392	194.4261
	246.3445	129.1950	192.7811
	276.718708	129.2045	193.2558
	274.5008	128.9511	194.9502
	274.6640	129.3329	194.6129
	252.4989	128.9323	193.5942
	276.3599	129.0188	198.9661
Average Time	260.3992	129.0933	195.8743

While the time trials were running, a power meter tracked the power draw, and the maximum power used during the prediction was recorded, as shown in Table 5.4. This worst-case scenario facilitates determining how much power would be drawn in the field if large amounts of images were processed simultaneously.

Table 5.4 Max Power Usage While Predicting

Run	Pi 4–- Raspbian (Watts)	Pi 4 – Bookworm (Watts)	Nano (Watts)
1	6.1	6.8	6.3
2	5.7	6.8	6.6
3	5.7	6.6	6.6
4	5.7	6.7	6.7
5	5.8	6.7	6.8
6	5.7	6.7	6.8
7	5.8	6.5	6.5
8	5.8	6.8	6.7
9	5.8	6.8	6.7
10	5.9	6.8	6.5
Average	5.8	6.72	6.62

The power draw between the Bookworm OS and the Nano was 0.1 watts. The Raspberry Pi running the Raspbian OS used slightly less power overall. However, it did have to run for much longer.

A total of fifty-five images were flagged as close. Out of these fifty-five images, twenty of them were mislabeled. Only 107 images were labeled incorrectly. This means the test dataset had a failure rate of approximately 6.79%. Out of these images, the majority of the mislabeled were either in the Good or Dusty categories. Sixty-nine good images were labeled Dusty, while 28 "Dusty" panels were labeled "Good." Table 5.5 shows a full breakdown of mislabeled images.

Table 5.5 Breakdown of Mislabeled Images

	Labeled				
# Incorrectly Labeled	Broken Glass	Dusty	Good	Hot Spots	Microcracks
Broken Glass	0	3	0	0	0
Dusty	0	0	28	1	0
Good	0	69	0	0	0
Hot Spot	0	5	1	0	0
Microcracks	0	0	0	0	0

Camera Tests

With Camera testing, the "Predict_N_Sort.py" Python program was modified to take a picture at 224 x 224 pixels resolution. If it is not the correct resolution, it will resize and normalize it. After that, it will use the TensorFlow Lite model and place that image into a folder based on the model's decision. Figure 5.3 shows the block diagram of how this code functions.

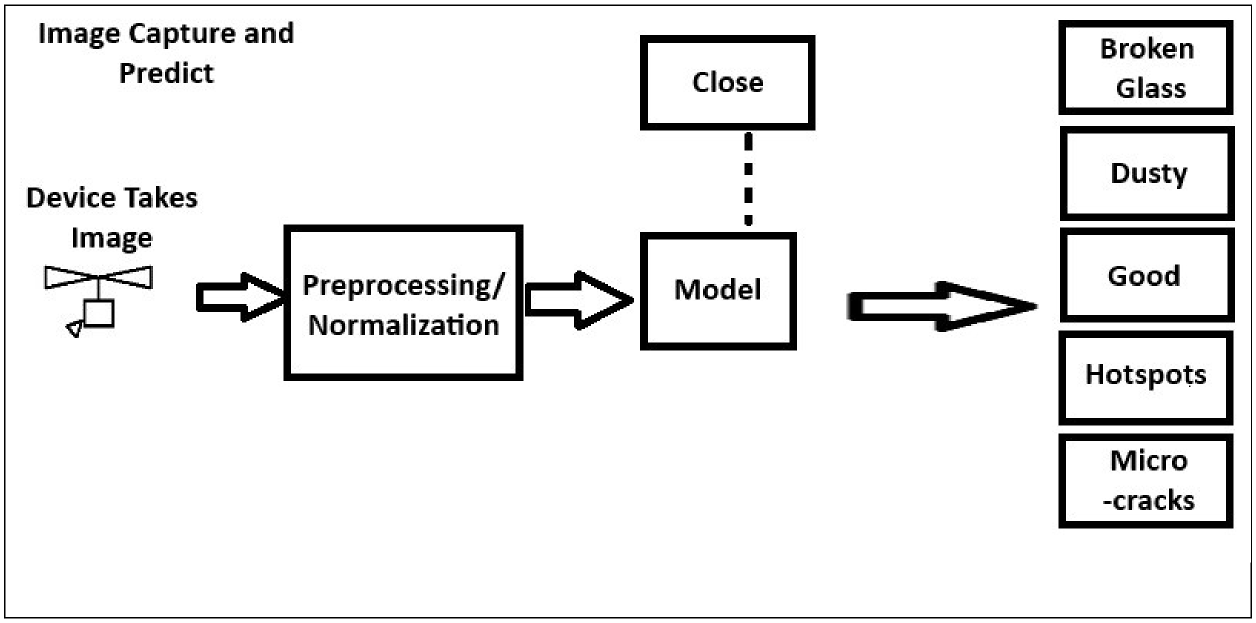


Figure 5.3

Programming Block Diagram for capImg_Predic_N_Sort.py

Ten pictures were taken, and the time it took to complete this process was recorded, which is presented in Table 5.6.

Table 5.6 Picture and Sort Time Trials

Image Capture	Raspberry Pi 4 - Raspbian Time (s)	Raspberry Pi 4 - Bookwork Time (s)	Jetson Nano Time (s)
1	0.3095	0.1929	0.5710
2	0.2948	0.1943	0.3926
3	0.2913	0.2330	0.3725
4	0.3035	0.1922	0.4443
5	0.3113	0.1967	0.4418
6	0.3110	0.1368	0.4358
7	0.2987	0.1891	0.4362
8	0.3134	0.1948	0.4373
9	0.3011	0.1381	0.3857
10	0.2940	0.1821	0.3373
Average	0.3029	0.1850	0.4255

The average time on the Raspberry Pi with Raspbian was roughly 0.303 seconds, which, if a human is in the loop taking pictures of solar panels, should give the system enough time to process and sort before the next image comes in. This is surprising, given how poorly it predicted a large swath of images. It even beat the Nano, which had an average of 0.425 seconds. However, the Raspberry Pi running with Bookworm and picam2 beat both with an average of 0.185 seconds. Like before, the maximum power usage was recorded during the program's run-time, as shown in Table 5.7. The Nano power draw is slightly less than the Raspberry Pi, beating it by an average of .14 Watts. It is worth noting that the power savings are offset by the additional code runtime on this device.

Table 5.7 Max Power Usage Single Image Capture and Inference

Run	Pi 4–- Raspbian (Watts)	Pi 4 – Bookworm (Watts)	Nano (Watts)
1	5.5	5.1	4.6
2	5.6	5.1	4.7
3	5.7	5.1	4.8
4	6.0	5.2	5.2
5	5.4	5.1	5.2
6	5.9	5.1	5.2
7	5.4	5.2	4.7
8	5.8	5.1	4.7
9	5.7	5.2	5.2
10	5.8	5.2	5.2
Average	5.68	5.14	4.95

Flight Times

The average power usage of the camera tests was lower than predicted. Power usage will impact flight time the least by keeping the run time short. This is because less wattage is used over a smaller amount of time, and less drain will be done on the battery. Table 5.8 shows an estimated flight time based on the additional power draw of our two devices while running at different throttle settings. In this table, the power usage of the code was calculated based on the code running for 1 minute straight, using the equation milliamp hours per minute/ max battery capacity.

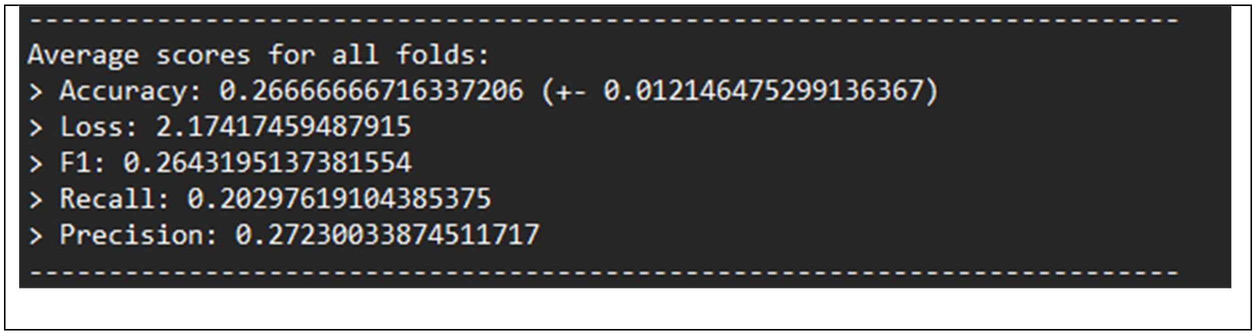
Table 5.8 Estimated Flight Times

Throttle usage	Flight Time - base (Min)	FT – Raspberry Pi 4 Idle	FT – Jetson Nano Idle	FT – Raspberry Pi 4 Image capture	FT – Jetson Nano Image Capture
100%	8.9286	8.8256	8.8974	6.8496	7.9143
65%	21.7391	21.1384	21.5553	12.5010	16.5691
50%	38.4615	36.6204	37.8900	16.6684	24.7811

It is worth noting that this estimate does not include the impact of additional weight on the UAV, so the estimate is not 100% accurate. However, these calculations were based on the four propellers datasheet [55] used by the 3DR X8+. However, from the estimates, the Raspberry Pi will affect the flight time more than the Nano. However, this additional power draw allows for the ability to take five images and classify them over a minute compared to the Nano, which can only do two images in the same amount of time.

K-fold Validation and Model Adjustments

The other tests for model inference on devices and power consumption have great results. The model was then put through a k-fold cross-validation of k=10. Figure 5.4 shows the results of the 100 epoch model. The average score for all ten folds was just 26.67% accuracy, much lower than the original 92.89%.



```
------------------------------------------------------------------------
Average scores for all folds:
> Accuracy: 0.26666666716337206 (+- 0.012146475299136367)
> Loss: 2.17417459487915
> F1: 0.2643195137381554
> Recall: 0.20297619104385375
> Precision: 0.27230033874511717
------------------------------------------------------------------------
```

Figure 5.4

Ten Fold Cross-Validation of the model

The resulting averages show an average validation loss of 2.17 and an average validation accuracy of 26.6% for the model. The first thing that was done was a review of the dataset. Upon review, there were less than four good microcracked images. There were so few microcracked images in general that the category was removed, and the model was rebuilt and tested again. The accuracy increased to 30.12% in this case, and the loss dropped to 1.64. The accuracy improvement makes sense since there is one less category. The worst-case scenario is a one-in-four chance of predicting versus the one-in-five chance of the original model. The loss rate drop shows that the microcracked images affected the model training. The next course of action was to equalize the image pool in the four-image type dataset. There were ample dusty and good PV images, while there were only ten images for each category. After searching for more images, the image count for hotspots and broken glass was increased to 94 images each. Since these images were close-up, aerial PV images, like those shown in Figure 5.5, were removed to remove the bias of additional features. For instance, a potential bias that could be perceived is the roadway shown in Figure 5.5. While the image contains solar panels, if enough roadways are added to the dataset, the model could potentially start to learn them as a potential feature.

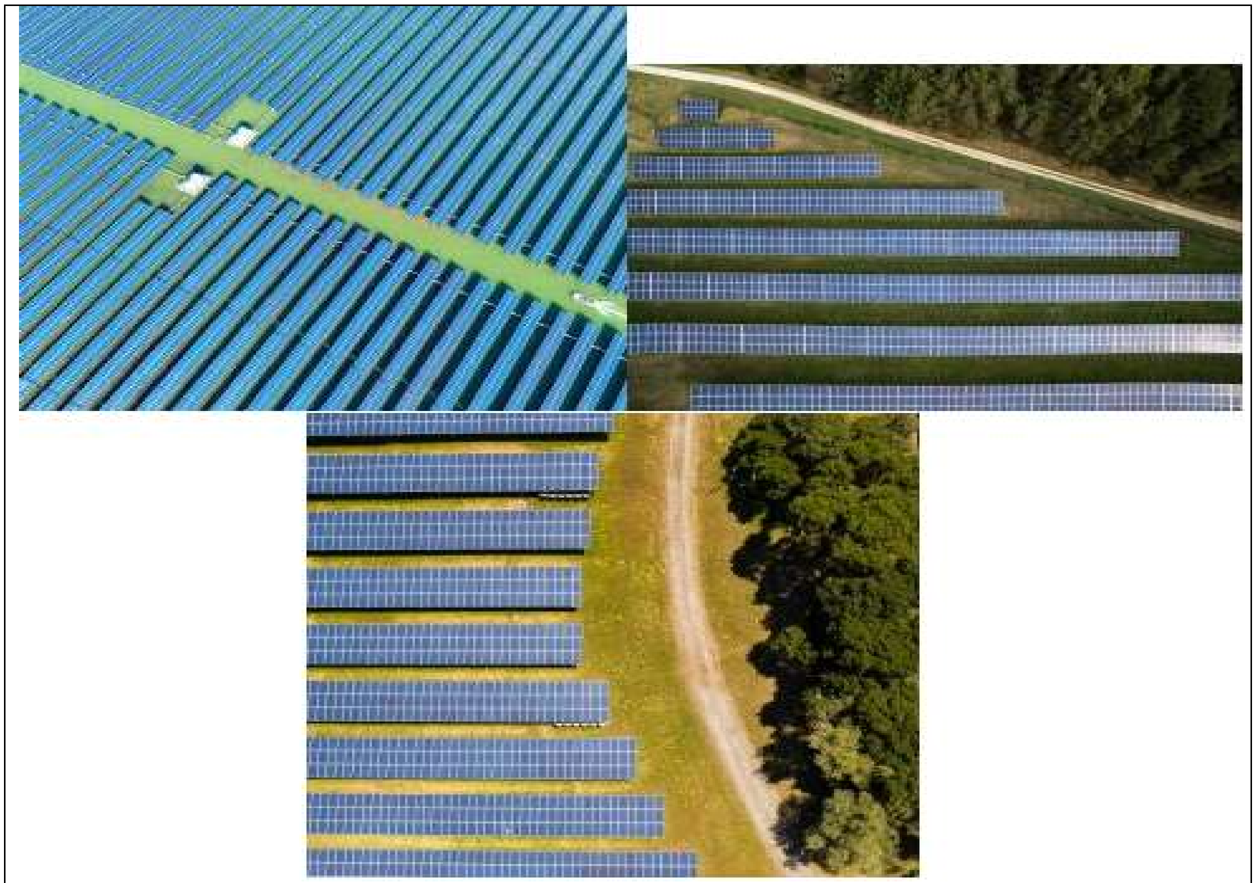


Figure 5.5

Example of Aerial Images Removed

After removing aerial images, a random number generator was used to pull 94 images for the good and dusty categories. For quick prototyping, the k-fold value was reduced to four from ten, and a total of ten epochs were used. The model was trained using k-fold validation on just the 94 images and an augmented image set comprised of 1,880 images. The original 3,000 images were compared to the original model for the final four categories. To understand how the model is learning from the classes provided, only two classes were trained. After the 94-image test was completed, another test with 1,880 images was performed. After this, the broken glass class was added to the training process, training on both 94 and 1,880 images each. After those tests were completed, the final class of hot spots was added. Table 5.9 shows the average accuracy results for each test with un-augmented and augmented images, as well as the accuracy after each class was added back into the model. Tables 5.10, 5.11, and 5.12 present accuracy results via a confusion matrix for two classes (dusty and good), three classes (broken glass, dusty, and good), and four classes (broken glass, dusty, good, and hotspots, respectively. These tables break down the accuracy of each class, the overall accuracy, and a breakdown of what the model inferred each test image.

Table 5.9 K-fold Results

Average	Validation Accuracy (%)	Validation Loss
Dusty/Good @94 images	60.07	0.88
Dusty/Good @1880 images	56.55	0.88
Broken/Dusty/Good @94 images	30.56	1.47
Broken/Dusty/Good @1880 images	35.12	1.35
Broken/Dusty/Good/Hotspot @94 images	25.30	1.86
Broken/Dusty/Good/Hotspot @1880 images	28.02	1.53
Broken/Dusty/Good/Hotspot @3000 images	28.12	1.49

Table 5.10 Confusion Matrix: Dusty and Good

94 images			
	Dusty	Good	Accuracy
Dusty	35	3	0.921053
Good	14	24	0.631579
			0.517544

1880 Images			
	Dusty	Good	Accuracy
Dusty	37	1	0.973684
Good	21	17	0.447368
			0.473684

Table 5.11 Confusion Matrix: Broken Glass, Dusty, and Good

94 Images				
	Broken	Dusty	Good	Accuracy
Broken	29	15	16	0.483333
Dusty	32	13	15	0.216667
Good	36	17	7	0.116667
				0.272222

1880 Images				
	Broken	Dusty	Good	Accuracy
Broken	22	30	8	0.366667
Dusty	13	34	13	0.566667
Good	7	30	23	0.383333
				0.438889

Table 5.12 Confusion Matrix: Broken Glass, Dusty, Good, and Hotspots

94 Images					
	Broken	Dusty	Good	Hotspot	Accuracy
Broken	24	6	20	10	0.4000
Dusty	15	18	18	9	0.3000
Good	11	6	35	8	0.5833
HotSpot	10	15	10	25	0.4167
					0.5667

1880 Images					
	Broken	Dusty	Good	Hotspot	Accuracy
Broken	36	13	8	3	0.6000
Dusty	8	29	14	9	0.4833
Good	13	17	28	2	0.4667
HotSpot	22	12	14	12	0.2000
					0.5833

3000 Images					
	Broken	Dusty	Good	Hotspot	Accuracy
Broken	20	14	11	15	0.3333
Dusty	14	34	6	6	0.5667
Good	15	17	27	1	0.4500
HotSpot	3	25	12	20	0.3333
					0.5611

The average loss decreased except for the dusty and good PV categories (Table 5.9). For example, broken glass, dusty, and good dropped 1.47% to 1.35%, and the average accuracy went up from 30.56% to 35.12%, which is expected. As more data is used to train the model, the model learns a more accurate representation of the feature distributions used to separate one class from another. Overall, the model's loss improved, which is a good indicator that over-fitting is not occurring. However, with such a finite number of images, the data was augmented to 2,000% for 1,880 and up to 3,191% for 3,000 images. There is concern that augmenting the base dataset by such a large percentage is biasing the model. Table 5.12 shows there is an actual trend of dropping accuracy when the model is trained on 3,000 augmented images compared to 1,880 augmented images. The original model had a loss rate of 2.17 and an accuracy of 26.6%, with the original five-class dataset of ten images for the broken glass, hotspot, and microcracked classes. Removing the microcracked images resulted in a loss of 1.64 and an accuracy of 30.12%. The model's accuracy drops when the number of images per class is kept the same. Data augmentation may create a bias that results in higher accuracy. For a less biased model, it is suggested that the number of images per class be the same and the augmentation of the dataset be limited. A better approach would be to collect and add additional images to each class. These images could include any test images labeled "close" from previous tests, but further research is needed to ensure the model is not overfitting. One of the options to help ensure overfitting is not occurring is just to get more images. However, since that can be an issue, lowering the number of MobileNet layers and simplifying the model is one option to help prevent overfitting with smaller datasets. The 154 MobileNet layers may be too deep of a CNN for the model to learn properly. Another option could be expanding the number of classes, adding classes like bird droppings, vegetation growth, etc. This could give the model more data and potential new features to help differentiate the current classes.

CHAPTER VI

FUTURE WORK

There are still concerns about how small the image pool is for each class within the dataset before augmentation. One possible approach to dealing with this concern is the selection of a classifier that works better with smaller datasets. This is in addition to updating the model with more images as they are captured by onboard cameras or discovered through additional searches. Continuously updating the model will allow for a better, more robust inference system. An investigation into image distortion, resulting from resizing images to 224-by-224 pixel images, impacts on model performance. The current device choice running our model can classify images correctly while having a minor impact on flight time. This model can be used to classify and report damage to PVs. Given that the end goal is to deploy and integrate this model on a UAV, an additional hardware choice could still be made. Smaller edge devices could still be explored as the weight of the current hardware choices on top of the potential weight of any other instrumentation could cause an additional impact on flight time. TensorFlow Lite allows even smaller edge devices to use less power and weight. A device like Espressif's ESP32-S3-EYE or a similar edge device could be considered if power and weight are issues with the original hardware targets. However, switching the software codebase from Python to C++ could be difficult on some edge devices like the Espressif. A choice of whether to use the UAV's camera or an additional mounted camera to acquire images still needs to be made. If using the UAV's onboard camera, a process to interface and use these images still needs to be created. After the camera interface is developed and integrated into the UAV, an image Geo-tagging process will need to be made, integrated, and tested.

CHAPTER VII

CONCLUSION

This study demonstrated that a MobileNetV2 Convolution Neural Network classification model for Solar Panel health could be determined in real-time. This model can classify four types of panel issues and determine if a panel has no issues, all while having a short inference time. Currently, the latest OS on the Raspberry Pi is the best option. The Pi, using this small 5-class classification model and program, can take, normalize, and process a new image in roughly 0.185 seconds. This allowed the next steps to putting this on a UAV, taking real-world photos, and processing on the fly. Using a TensorFlow Lite model also allows scaling down for other lightweight, low-power systems that could further extend the product's battery life. Meanwhile, the 92.89% accuracy of the originally tested model appears to be misleading based on the k-fold validation results. Future dataset tuning and more image data should allow for a more robust model. The next step should be to start overlaying GPS data on images. The GPS data will be processed to allow the user to quickly locate a given panel after identifying a damaged panel. Overall, the proposed method will enable users to find and diagnose solar panel issues rapidly, thus allowing quicker troubleshooting and repair.

REFERENCES

[1] Lawerence Berkely National Laboratory, et al. "Utility-Scale Solar, 2023 Edition Empirical Trends in Deployment, Technology, Cost, Performance, PPA Pricing, and Value in the United State" emp.lbl.gov, Oct. 2023, emp.lbl.gov/sites/default/files/utility_scale_solar_2023_edition_slides.pdf. Accessed 1 Nov. 2023.

[2] E. Cook, S. Luo, and Y. Weng, "Solar Panel Identification Via Deep Semi-Supervised Learning and Deep One-Class Classification," in *IEEE Transactions on Power Systems*, vol. 37, no. 4, pp. 2516-2526, July 2022, doi: 10.1109/TPWRS.2021.3125613.

[3] Majdi, Abdulrhman, et al. "Fundamental study related to the development of modular solar panel for improved durability and repairability." *IET Renewable Power Generation* 15.7 (2021): 1382-1396.

[4] Wolfgang Muehleisen a, et al. "Outdoor Detection and Visualization of Hailstorm Damages of Photovoltaic Plants." *Renewable Energy*, Pergamon, 8 Nov. 2017, www.sciencedirect.com/science/article/pii/S0960148117311114?casa_token=UV9Rkt50PrYAAAAA%3AdTybnJtjTvVaGNXBo_eR_CRPt_zKgDWJNI3O4Rlql1aLleNFXe4YjmC2FCbm3c0NrGPOgJNnFwaw.

[5] Li, Qi, et al. "Automatic damage detection on rooftop solar photovoltaic arrays." *Proceedings of the 7th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, 18 Nov. 2020, https://doi.org/10.1145/3408308.3431130.

[6] Chen, Xin, et al. "Automated defect identification in electroluminescence images of solar modules." *Solar Energy*, vol. 242, Aug. 2022, pp. 20–29, https://doi.org/10.1016/j.solener.2022.06.031.

[7] Dwivedi, Divyanshi, et al. "Identification of surface defects on solar PV panels and wind turbine blades using attention based deep learning model." *Engineering Applications of Artificial Intelligence*, vol. 131, May 2024, p. 107836, https://doi.org/10.1016/j.engappai.2023.107836.

[8] Padmavathi, N., and A. Chilambuchelvan. "Fault detection and identification of solar panels using Bluetooth." *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, Aug. 2017, https://doi.org/10.1109/icecds.2017.8390096.

[9] Segovia Ramírez, Isaac, et al. "Fault detection and diagnosis in photovoltaic panels by radiometric sensors embedded in unmanned aerial vehicles." *Progress in Photovoltaics: Research and Applications*, vol. 30, no. 3, 3 Oct. 2021, pp. 240–256, https://doi.org/10.1002/pip.3479.

[10] Zefri, Yahya, et al. "Thermal infrared and visual inspection of photovoltaic installations by UAV photogrammetry—application case: Morocco." *Drones*, vol. 2, no. 4, 23 Nov. 2018, p. 41, https://doi.org/10.3390/drones2040041.

[11] Zhao , Raymond. "Photovoltaic (PV) Solar Panel Identification and Fault Detection Using Unmanned Aerial Vehicles (UAVs): A Case Study of a 0.5 MW PV System ." *Yale University*, 2022.

[12] Li, Xiaoxia, et al. "Intelligent fault pattern recognition of aerial photovoltaic module images based on deep learning technique." *J. Syst. Cybern. Inf* 16 (2018): 67-71.

[13] Akram, M. Waqar, et al. "CNN based automatic detection of photovoltaic cell defects in electroluminescence images." *Energy*, vol. 189, Dec. 2019, p. 116319, https://doi.org/10.1016/j.energy.2019.116319.

[14] Fernández, Alberto, et al. "Robust detection, classification and localization of defects in large photovoltaic plants based on unmanned aerial vehicles and infrared thermography." *Applied Sciences*, vol. 10, no. 17, 27 Aug. 2020, p. 5948, https://doi.org/10.3390/app10175948.

[15] Jeffrey Kuo, Chung-Feng, et al. "Automatic detection, classification and localization of defects in large photovoltaic plants using unmanned aerial vehicles (UAV) based infrared (IR) and RGB imaging." *Energy Conversion and Management*, vol. 276, Jan. 2023, p. 116495, https://doi.org/10.1016/j.enconman.2022.116495.

[16] Kirsten Vidal de Oliveira, Aline, et al. "Aerial infrared thermography for low-cost and fast fault detection in utility-scale PV power plants." *Solar Energy*, vol. 211, Nov. 2020, pp. 712–724, https://doi.org/10.1016/j.solener.2020.09.066.

[17] Moradi Sizkouhi, Amirmohammad, et al. "A deep convolutional encoder-decoder architecture for autonomous fault detection of PV plants using multi-copters." *Solar Energy*, vol. 223, July 2021, pp. 217–228, https://doi.org/10.1016/j.solener.2021.05.029.

[18] Ruan, Chenjie, et al. "Deep learning-based method for PV panels segmentation and defects detection with infrared images." *2021 China Automation Congress (CAC)*, 22 Oct. 2021, https://doi.org/10.1109/cac53003.2021.9728350.

[19] Di Tommaso, Antonio, et al. "A multi-stage model based on yolov3 for defect detection in PV panels based on IR and visible imaging by Unmanned Aerial Vehicle." *Renewable Energy*, vol. 193, June 2022, pp. 941–962, https://doi.org/10.1016/j.renene.2022.04.046.

[20] Ayoub, Naeem, and Peter Schneider-Kamp. "Real-time onboard detection of components and faults in an autonomous UAV system for power line inspection." *Proceedings of the 1st International Conference on Deep Learning Theory and Applications*, 2020, https://doi.org/10.5220/0009826700680075.

[21] KAYCI, Barış, et al. "İHA Tarafından Elde Edilen termal görüntüler kullanılarak fotovoltaik sistemde derin öğrenme Tabanlı Arıza Tespiti ve Teşhisi." *Journal of Polytechnic*, 10 June 2022, https://doi.org/10.2339/politeknik.1094586.

[22] Ksira, Zakaria, et al. "A novel embedded system for real-time fault diagnosis of photovoltaic modules." *IEEE Journal of Photovoltaics*, vol. 14, no. 2, Mar. 2024, pp. 354–362, https://doi.org/10.1109/jphotov.2024.3359462.

[23] Le, Minhhuy, et al. "Thermal inspection of photovoltaic modules with deep convolutional neural networks on edge devices in AUV." *Measurement*, vol. 218, Aug. 2023, p. 113135, https://doi.org/10.1016/j.measurement.2023.113135.

[24] Shihavuddin, ASM, et al. "Image based surface damage detection of renewable energy installations using a unified deep learning approach." *Energy Reports*, vol. 7, Nov. 2021, pp. 4566–4576, https://doi.org/10.1016/j.egyr.2021.07.045.

[25] Özer, Tolga, and Ömer Türkmen. "An approach based on deep learning methods to detect the condition of solar panels in solar power plants." *Computers and Electrical Engineering*, vol. 116, May 2024, p. 109143, https://doi.org/10.1016/j.compeleceng.2024.109143.

[26] T. J. O'Shea, T. Roy, and T. C. Clancy, "Over-the-Air Deep Learning Based Radio Signal Classification," in *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 168-179, Feb. 2018, doi: 10.1109/JSTSP.2018.2797022. keywords: {Modulation;Feature extraction;Wireless communication;Neural networks;Machine learning;Fading channels;Decision trees;Cognitive radio;deep learning;modulation;neural networks;pattern recognition;sensor systems and applications;wireless communication},

[27] Li, Qi, Keyang Yu, and Dong Chen. "SolarDiagnostics: Automatic damage detection on rooftop solar photovoltaic arrays." *Sustainable Computing: Informatics and Systems* 32 (2021): 100595.

[28] Meribout, Mahmoud, et al. "Solar panel inspection techniques and prospects." *Measurement* (2023): 112466.

[29] Lydia, MD Dafny, K. Sri Sindhu, and K. Gugan. "Analysis on solar panel crack detection using optimization techniques." *Journal of Nano-and Electronic Physics* 9.2 (2017): 2004-1.

[30] Cardinale-Villalobos, Leonardo, Renato Rimolo-Donadio, and Carlos Meza. "Solar panel failure detection by infrared UAS digital photogrammetry: a case study." *Int. J. Renew. Energy Res. (IJRER)* 10.3 (2020): 1154-1164.

[31] S. Voghoei, N. Hashemi Tonekaboni, J. G. Wallace and H. R. Arabnia, "Deep Learning at the Edge," 2018 International Conference on Computational Science and Computational Intelligence (CSCI), Las Vegas, NV, USA, 2018, pp. 895-901, doi: 10.1109/CSCI46756.2018.00177. Keywords: {Computational modeling; Quantization (signal); Deep learning; Internet of Things; Edge computing; Performance evaluation; Image edge detection; Edge Computing, Internet of Things (IoT), Deep Learning (DL), Deep Neural Networks (DNN)},

[32] Müller, Andreas C., and Sarah Guido. *Introduction to Machine Learning with Python: A Guide for Data Scientists*. O'Reilly Media, 2018.

[33] C.Gambella, B.Ghaddar, and J.Naoum-Sawaya, "Optimization problems for machine learning: A survey," European Journal of Operational Research, vol. 290, no. 3, pp. 807–828, 2021. [Online].Available: [1901.05331] Optimization Problems for Machine Learning: A Survey (arxiv.org)

[34] Rawat, Waseem, and Zenghui Wang. "Deep convolutional neural networks for image classification: A comprehensive review." *Neural Computation* 29.9 (2017): 2352-2449.

[35] Kapoor, Amita, et al. *Deep Learning with TensorFlow and Keras: Build and Deploy Supervised, Unsupervised, Deep, and Reinforcement Learning Models*. Packt Publishing, 2022.

[36] J. Brownlee, "Gentle introduction to the Adam optimization algorithm for deep learning," MachineLearningMastery.com, https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/ (accessed Dec. 20, 2023).

[37] Sharma, Nitika. "What Is Mobilenetv2? Features, Architecture, Application and More." *Analytics Vidhya*, 29 Dec. 2023, www.analyticsvidhya.com/blog/2023/12/what-is-mobilenetv2/#:~:text=The%20use%20of%20MobileNetV2%20for%20image%20classification%20offers,compared%20to%20larger%20and%20more%20computationally%20expensive%20models.

[38] Howard, Andrew G., et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications." *arXiv.Org*, 17 Apr. 2017, arxiv.org/abs/1704.04861.

[39] Andriana, Shehani. "The Most Common Types of Solar Panel Defects." *Engineering*, 23 Nov. 2023, engineerinc.io/the-most-common-types-of-solar-panel-defects/#:~:text=The%20Most%20Common%20Types%20of%20Solar%20Panel%20Defects,4%20Electrical%20Problems%20...%205%20Manufacturing%20Defects%20.

[40] B. T. Kitano, C. C. T. Mendes, A. R. Geus, H. C. Oliveira and J. R. Souza, "Corn Plant Counting Using Deep Learning and UAV Images," in *IEEE Geoscience and Remote Sensing Letters*, doi: 10.1109/LGRS.2019.2930549. keywords: {Agriculture;Computer architecture;Unmanned aerial vehicles;Training;Image segmentation;Deep learning;Cameras;Deep learning (DL);plant counting;precision agriculture.}

[41] Ammour, Nassim, et al. "Deep learning approach for car detection in UAV imagery." *Remote Sensing* 9.4 (2017): 312.

[42] Qu, Zhongnan. "Enabling Deep Learning on Edge Devices." *arXiv.Org*, 6 Oct. 2022, arxiv.org/abs/2210.03204.

[43] "Tensorflow Lite: ML for Mobile and Edge Devices." TensorFlow, www.tensorflow.org/lite. Accessed Dec. 2023.

[44] Hunter, Jona than, "Deep learning-based anomaly detection for edge-layer devices" (2022). *Masters Theses and Doctoral Dissertations.* https://scholar.utc.edu/theses/740

[45] Chen and X. Ran, "Deep Learning With Edge Computing: A Review," in *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655-1674, Aug. 2019, doi: 10.1109/JPROC.2019.2921977. keywords: {Deep learning;Edge computing;Cloud computing;Computational modeling;Training;Servers;Neural networks;Computer vision;Machine learning;Artificial intelligence;Artificial intelligence;edge computing;machine learning;mobile computing;neural networks},

[46] Qurishee, Murad Al, "Low-cost deep learning UAV and Raspberry Pi solution to real-time pavement condition assessment" (2019). *Master Theses and Doctoral Dissertations.* https://scholar.utc.edu/theses/601

[47] A. Zeggada, F. Melgani, and Y. Bazi, "A Deep Learning Approach to UAV Image Multilabeling," in *IEEE Geoscience and Remote Sensing Letters*, vol. 14, no. 5, pp. 694-698, May 2017, doi: 10.1109/LGRS.2017.2671922. keywords: {Unmanned aerial vehicles;Training;Neural networks;Feature extraction;Histograms;Image segmentation;Computer architecture;Convolutional neural networks (CNNs);image multilabel;Otsu's algorithm;unmanned aerial vehicles (UAVs);urban monitoring},

[48] Mittal, Payal, Raman Singh, and Akashdeep Sharma. "Deep learning-based object detection in low-altitude UAV datasets: A survey." *Image and Vision Computing* 104 (2020): 104046.

[49] "Solar Panel Dust Detection." Kaggle, 20 Nov. 2022, www.kaggle.com/datasets/hemanthsai7/solar-panel-dust-detection.

[50] AFROZ (2024). "Solar Panel Clean and Faulty Images." 2024, from https://www.kaggle.com/datasets/pythonafroz/solar-panel-clean-and-faulty-images.

[51] Sandler, Mark. "MobileNetV2: Inverted Residuals and Linear Bottlenecks." arXiv.org, 13 Jan. 2018, arxiv.org/abs/1801.04381.

[52] "Raspberry Pi 4 Specs and Benchmarks — the MagPi Magazine." The MagPi Magazine, magpi.raspberrypi.com/articles/raspberry-pi-4-specs-benchmarks.

[53] "NVIDIA Jetson Nano." NVIDIA Developer, developer.nvidia.com/embedded/jetson-nano.

[54] "Cuda Zone - Library of Resources." *NVIDIA Developer*, developer.nvidia.com/cuda-zone. Accessed 16 Feb. 2024.

[55] "Sunnysky V2216 Multirotor Brushless Motors." *SunnySky USA*, sunnyskyusa.com/products/sunnysky-v2216-motors. Accessed 17 Mar. 2024.

[56] Team, Keras. "Simple. Flexible. Powerful." Keras, keras.io/. Accessed 11 Dec. 2023.

[57] S. Kim, S. Chon, J. -K. Kim, J. Kim, Y. Gil and S. Jung, "Lightweight Convolutional Neural Network for Real-Time Arrhythmia Classification on Low-Power Wearable Electrocardiograph," *2022 44th Annual International Conference of the IEEE Engineering in Medicine & Biology Society (EMBC)*, Glasgow, Scotland, United Kingdom, 2022, pp. 1915-1918, doi: 10.1109/EMBC48229.2022.9871156.

[58] E. Arnaudo *et al.*, "A Comparative Evaluation of Deep Learning Techniques for Photovoltaic Panel Detection From Aerial Images," in *IEEE Access*, vol. 11, pp. 47579-47594, 2023, doi: 10.1109/ACCESS.2023.3275435.

[59] Piazza, Maria Carmela di, et al. "Identification of photovoltaic array model parameters by robust linear regression methods." *Renewable energy & power quality journal* 1 (2009): 143-149.

[60] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected Convolutional Networks," arXiv.org, https://arxiv.org/abs/1608.06993

[61] Li, En, Zhi Zhou, and Xu Chen. "Edge intelligence: On-demand deep learning model co-inference with device-edge synergy." *Proceedings of the 2018 Workshop on Mobile Edge Communications*. 2018.

[62] Chen, Jiasi, and Xukan Ran. "Deep learning with edge computing: A review." *Proceedings of the IEEE* 107.8 (2019): 1655-1674.

[63] Warden, Pete, and Daniel Situnayake. *TinyML: Machine Learning with Tensorflow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly, 2020.

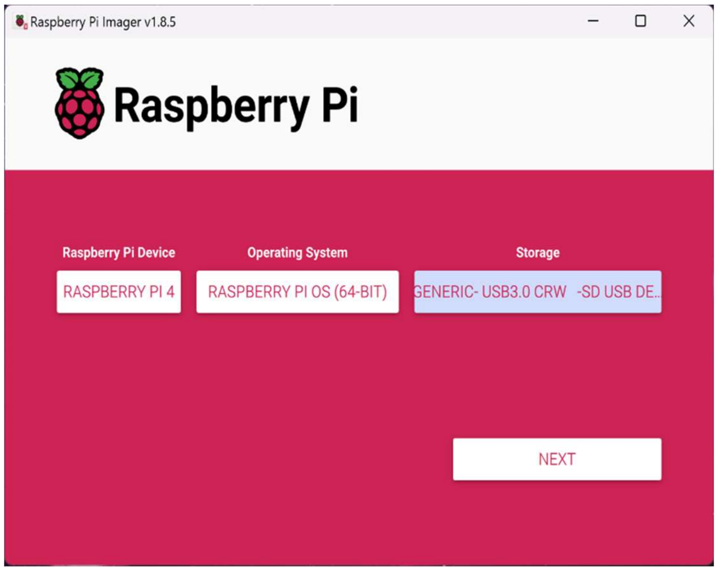
[64] Brownlee, J. (2023). "A Gentle Introduction to k-fold Cross-Validation." 2024, from https://machinelearningmastery.com/k-fold-cross-validation/.

[65] Lin, M., et al. (2013). "Network in network." arXiv preprint arXiv:1312.4400.

APPENDIX A

SETUP RASPBERRY PI 4

Before the Raspberry Pi is hooked up, either download the Raspberry Pi Imager software or be prepared to install it over the ethernet via SSH. This example will be using the 64-bit Raspberry Pi OS. Select RASPBERRY PI OS (64-bit) FULL (the current version as of writing is Bookworm). Launch Raspberry Pi Imager and select RASPBERRYPI 4 for the device and RASPBERRY PI OS (64-BIT) for the Operating system. Also, select the location of the SD CARD. See Figure A.1 SD Card Set up – Raspberry Pi Imager.

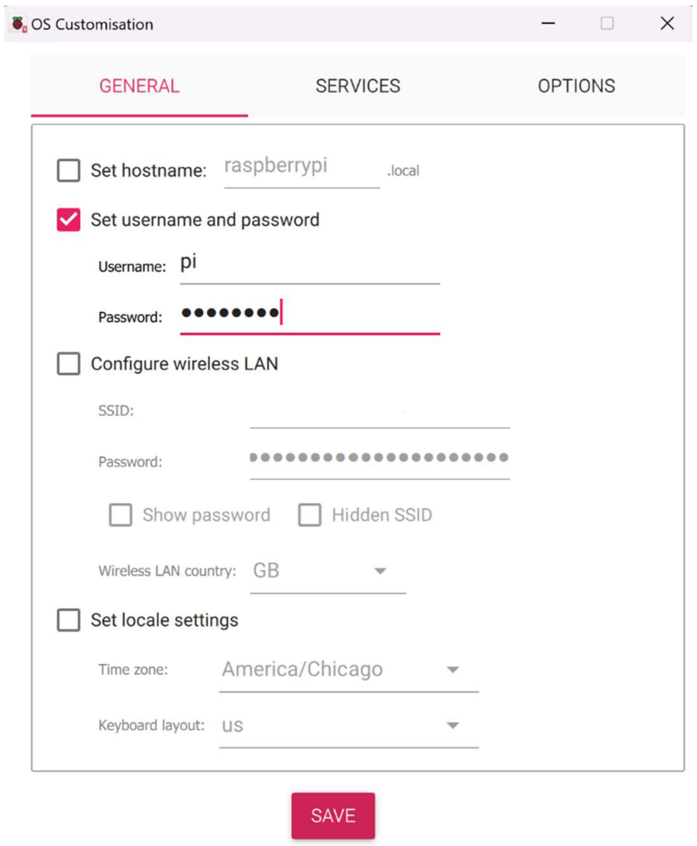


Raspberry Pi Device: RASPBERRY PI 4
Operating System: RASPBERRY PI OS (64-BIT)
Storage: GENERIC- USB3.0 CRW -SD USB DE..

Figure A.1

SD Card Set up – Raspberry Pi Imager

When asked to customize OS settings, select EDIT SETTINGS and update the following: Set the username to pi and a password that can be remembered. See Figure A.2—Raspberry Pi OS Customization Settings.



☐ Set hostname: raspberrypi .local

☑ Set username and password

Username: pi

Password: ●●●●●●●●●

☐ Configure wireless LAN

SSID:

Password: ●●●●●●●●●●●●●●●●●●●●●

☐ Show password ☐ Hidden SSID

Wireless LAN country: GB

☐ Set locale settings

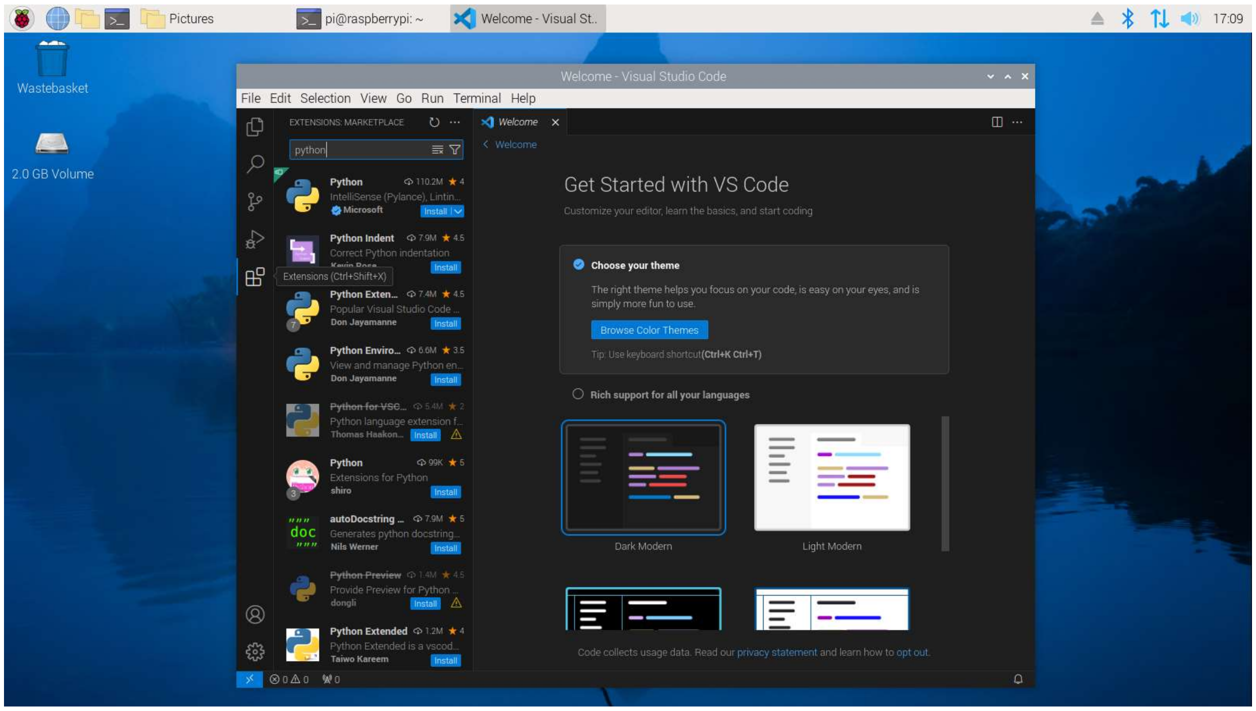
Time zone: America/Chicago

Keyboard layout: us

Figure A.2

Raspberry Pi OS Customization Settings

Continue with the installation process. Once the SD card is imaged, insert it into the Raspberry Pi 4 and turn it on. Log in and open the terminal to run the following commands: sudo apt update && sudo apt upgrade -y. This will make sure that the device is fully up to date. Install the dev environment in the terminal with the Sudo apt-get install code. Once installed, launch Visual Studio Code by typing the code in the terminal and hitting enter. Click the extensions button, type in Python, and click install for the Python extension. See Figure A.3 – Visual Studio Code Python Extension Install on Raspberry Pi 4. Once the code is installed, install the Python extension to debug or edit the codebase.



Python – IntelliSense (Pylance), Lintin... – Microsoft – 110.2M ★ 4
Python Indent – Correct Python indentation – Kevin Rose – 7.9M ★ 4.5
Python Exten... – Popular Visual Studio Code ... – Don Jayamanne – 7.4M ★ 4.5
Python Enviro... – View and manage Python en... – Don Jayamanne – 6.6M ★ 3.5
Python for VSC... – Python language extension f... – Thomas Haakon... – 5.4M ★ 2
Python – Extensions for Python – shiro – 99K ★ 5
autoDocstring ... – Generates python docstring... – Nils Werner – 7.9M ★ 5
Python Preview – Provide Preview for Python ... – dongli – 1.4M ★ 4.5
Python Extended – Python Extended is a vscod... – Taiwo Kareem – 1.2M ★ 4

Customize your editor, learn the basics, and start coding

The right theme helps you focus on your code, is easy on your eyes, and is simply more fun to use.

Tip: Use keyboard shortcut(Ctrl+K Ctrl+T)

Code collects usage data. Read our privacy statement and learn how to opt out.

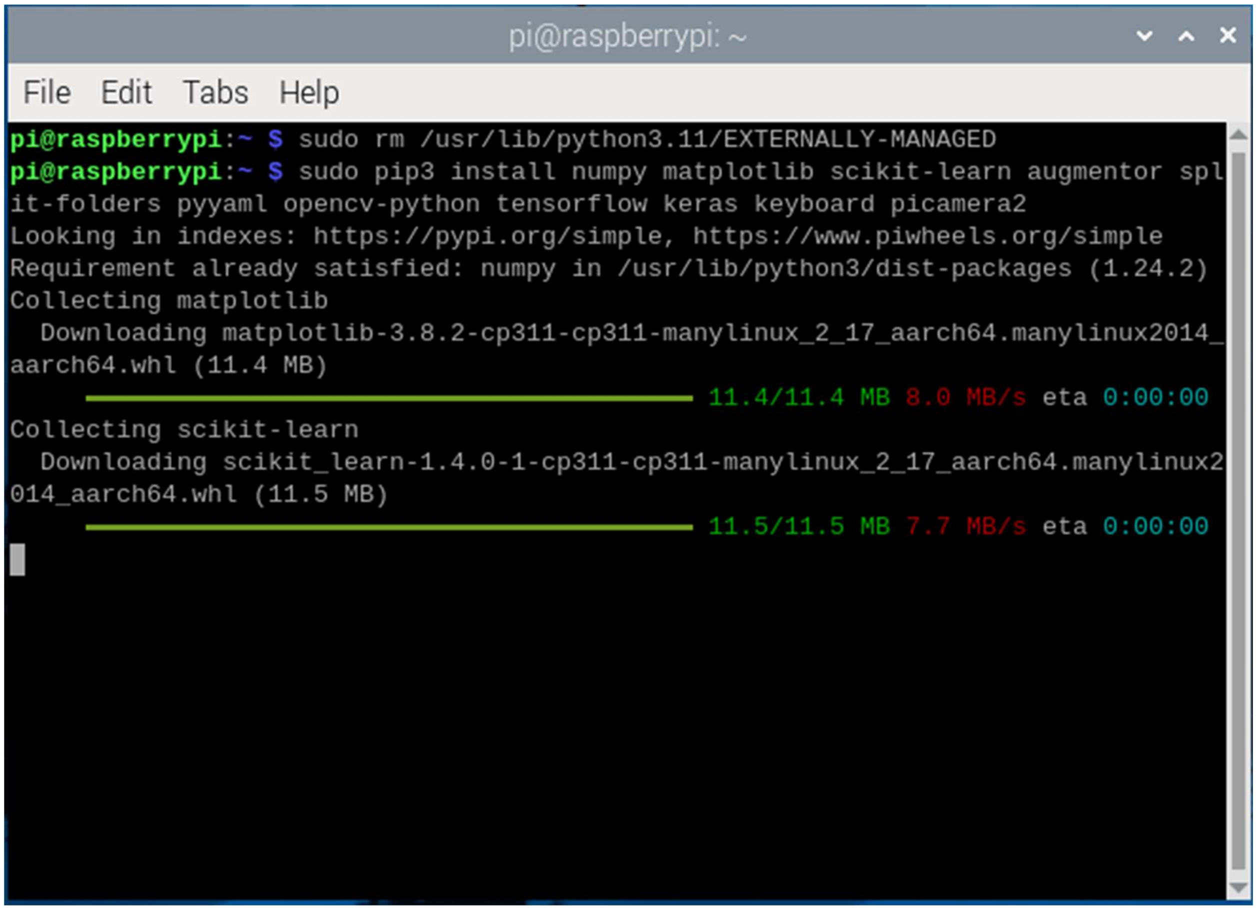
Figure A.3

Visual Studio Code Python Extension Install

On Raspberry Pi 4

While the extension installs the rest of the code dependencies, go back to the terminal and run Sudo rm /usr/lib/python3.XX/EXTERNALLY-MANAGED (replace xx with python version number;

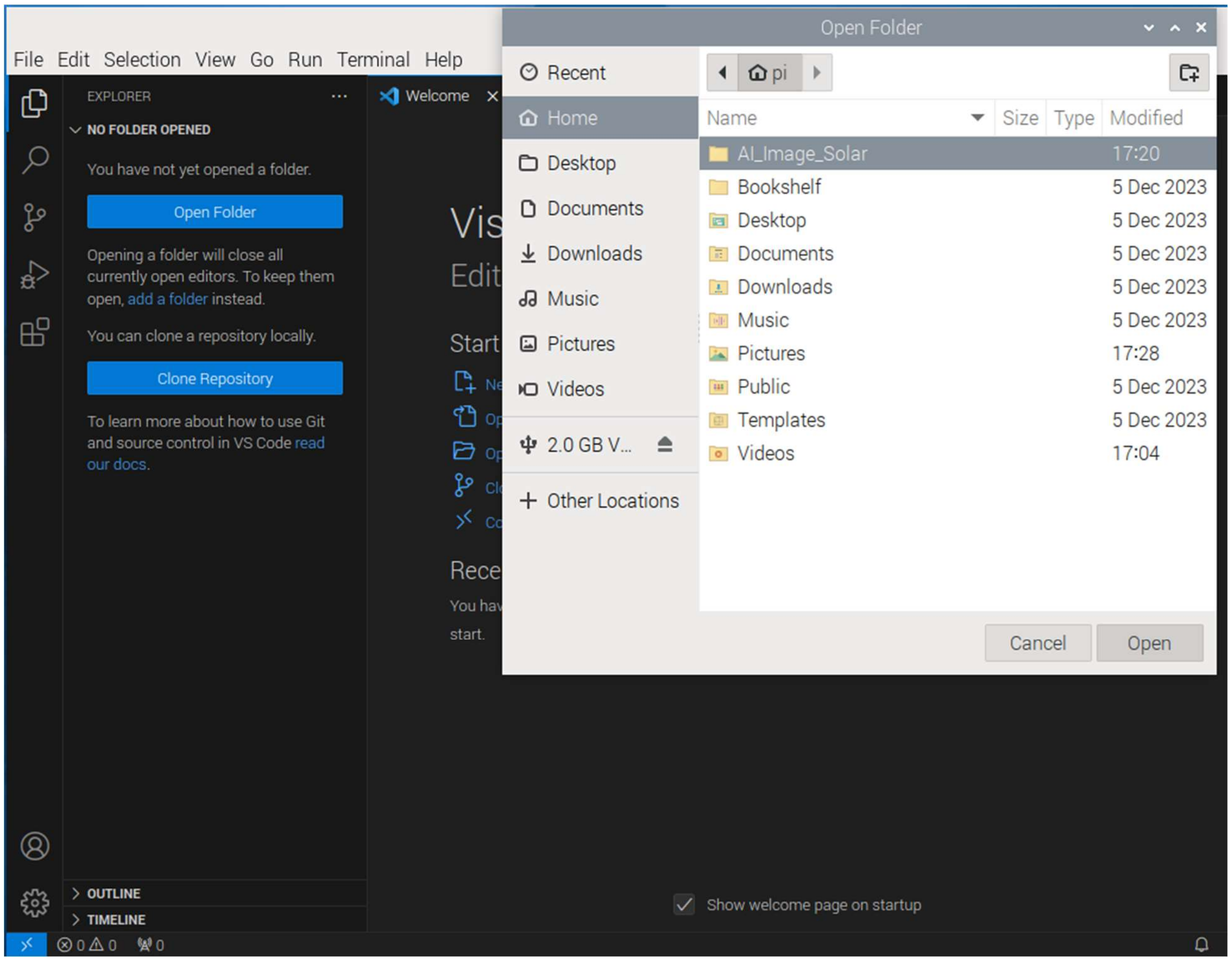
currently it is 11.) and Sudo pip3 install numpy matplotlib scikit-learn augmentor split-folders pyyaml opencv-python tensorflow keras keyboard picamera2. See Figure A.4 – Installing Dependencies on Raspberry Pi 4.



```
pi@raspberrypi:~ $ sudo rm /usr/lib/python3.11/EXTERNALLY-MANAGED
pi@raspberrypi:~ $ sudo pip3 install numpy matplotlib scikit-learn augmentor spl
it-folders pyyaml opencv-python tensorflow keras keyboard picamera2
Looking in indexes: https://pypi.org/simple, https://www.piwheels.org/simple
Requirement already satisfied: numpy in /usr/lib/python3/dist-packages (1.24.2)
Collecting matplotlib
  Downloading matplotlib-3.8.2-cp311-cp311-manylinux_2_17_aarch64.manylinux2014_
aarch64.whl (11.4 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 11.4/11.4 MB 8.0 MB/s eta 0:00:00
Collecting scikit-learn
  Downloading scikit_learn-1.4.0-1-cp311-cp311-manylinux_2_17_aarch64.manylinux2
014_aarch64.whl (11.5 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 11.5/11.5 MB 7.7 MB/s eta 0:00:00
```

Figure A.4
Installing Dependencies on Raspberry Pi 4

Once all the dependencies are installed, copy the AI_Image_Solar Folder from AI_Image_Solar_pi_Bookworm to /home/pi. Return to Visual Studio Code, go to the explorer tab, and open the AI_Image_Solar Folder by clicking open folder, selecting the AI_Image_Solar folder, and clicking open. Figure A.5 – Raspberry Pi Visual Studio Developer Environment Set up.



You have not yet opened a folder.

Opening a folder will close all currently open editors. To keep them open, add a folder instead.

You can clone a repository locally.

To learn more about how to use Git and source control in VS Code read our docs.

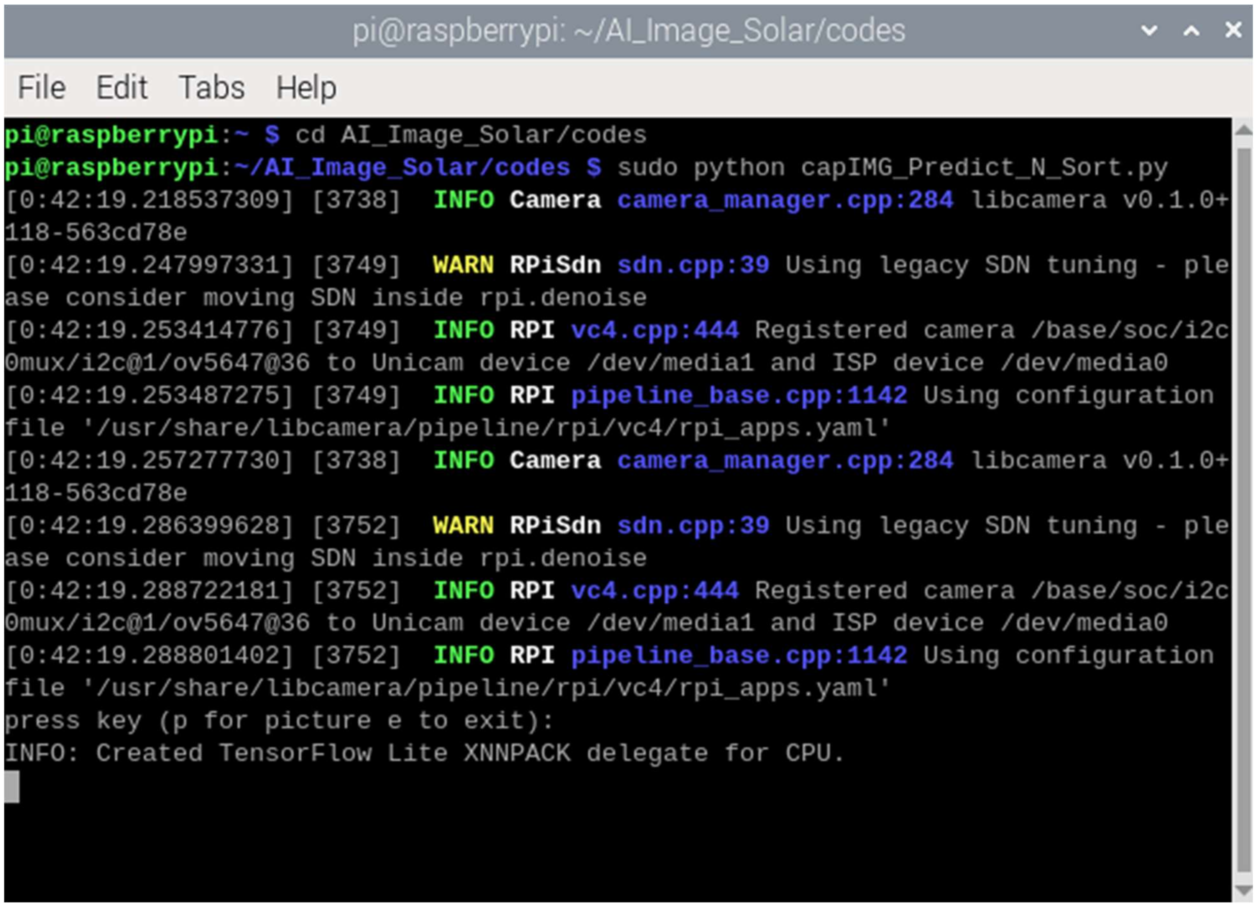
Name	Size	Type	Modified
AI_Image_Solar			17:20
Bookshelf			5 Dec 2023
Desktop			5 Dec 2023
Documents			5 Dec 2023
Downloads			5 Dec 2023
Music			5 Dec 2023
Pictures			17:28
Public			5 Dec 2023
Templates			5 Dec 2023
Videos			17:04

Figure A.5

Raspberry Pi Visual Studio Developer Environment Set Up

With the dev environment fully loaded, go to codes Predict_N_Sort.py and click the run python file to verify everything is working correctly in Visual Studio code. There should be some files in the unsorted folder for it to sort through. Once it's complete, there should see some of the Sort

Folders now have five new folders, correlating to the five-panel types, and that all the images from the unsorted file are moved into them. To test if the camera is operational, we will need to run the Python code as root. Running either image_capture_picam2.py, capIMG_Predict_N_Sort.py, or capIMG_Predict_N_Sort_instant.py as root. By using sudo python <pythonfilename.py>. To do this, open or reuse the existing instance of the terminal, use commands cd /AI_Image_Solar/codes, and then sudo python capIMG_Predict_N_Sort.py. See Figure A.6 Demonstration of capIMG_Predict_N_Sort.py. There will be a command asking to press p to take a picture or e to exit the code.



```
pi@raspberrypi:~ $ cd AI_Image_Solar/codes
pi@raspberrypi:~/AI_Image_Solar/codes $ sudo python capIMG_Predict_N_Sort.py
[0:42:19.218537309] [3738]  INFO Camera camera_manager.cpp:284 libcamera v0.1.0+
118-563cd78e
[0:42:19.247997331] [3749]  WARN RPiSdn sdn.cpp:39 Using legacy SDN tuning - ple
ase consider moving SDN inside rpi.denoise
[0:42:19.253414776] [3749]  INFO RPI vc4.cpp:444 Registered camera /base/soc/i2c
0mux/i2c@1/ov5647@36 to Unicam device /dev/media1 and ISP device /dev/media0
[0:42:19.253487275] [3749]  INFO RPI pipeline_base.cpp:1142 Using configuration
file '/usr/share/libcamera/pipeline/rpi/vc4/rpi_apps.yaml'
[0:42:19.257277730] [3738]  INFO Camera camera_manager.cpp:284 libcamera v0.1.0+
118-563cd78e
[0:42:19.286399628] [3752]  WARN RPiSdn sdn.cpp:39 Using legacy SDN tuning - ple
ase consider moving SDN inside rpi.denoise
[0:42:19.288722181] [3752]  INFO RPI vc4.cpp:444 Registered camera /base/soc/i2c
0mux/i2c@1/ov5647@36 to Unicam device /dev/media1 and ISP device /dev/media0
[0:42:19.288801402] [3752]  INFO RPI pipeline_base.cpp:1142 Using configuration
file '/usr/share/libcamera/pipeline/rpi/vc4/rpi_apps.yaml'
press key (p for picture e to exit):
INFO: Created TensorFlow Lite XNNPACK delegate for CPU.
```

Figure A.6

Demonstration of capIMG_Predict_N_Sort.py

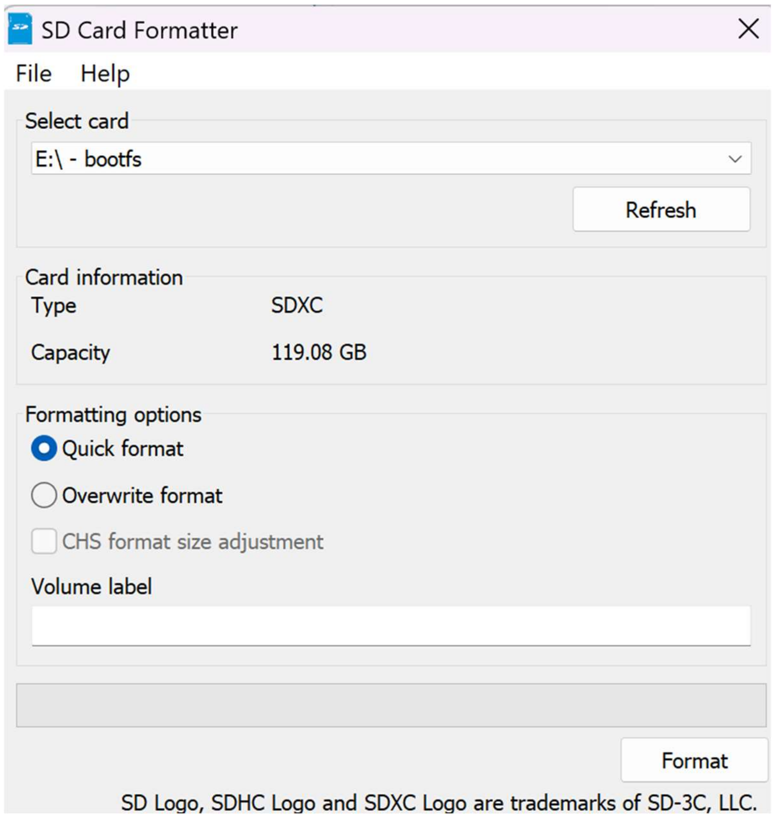
The image will be shown sorted into a folder. The process of setting up the Raspberry Pi 4 is finished.

APPENDIX B

SETUP JETSON NANO

The Jetson Nano is a GPU-focused device with quite a few more steps to set up compared to the Raspberry Pi. Three bits of software are needed to set up the Nano. First, an SD Card Formatter can be found at https://www.sdcard.org/downloads/formatter_4/eula_windows/. The latest version of Jetpack (4.6.3 at the time of writing) is https://developer.nvidia.com/jetson-nano-sd-card-image, and Etcher, the imaging software is located at https://www.balena.io/etcher.

Launch the SD Card formatting software shown in Figure B.1. Select the hard drive (drive E:/ in the example). Select "quick format," then leave the volume label section blank. Click format.



Select card: E:\ - bootfs

Card information
Type SDXC
Capacity 119.08 GB

Formatting options
[✓ Quick format]
Overwrite format
CHS format size adjustment
Volume label

SD Logo, SDHC Logo and SDXC Logo are trademarks of SD-3C, LLC.

Figure B.1

SD Card Formatting

Unzip the downloaded image folder for the Jetson Nano SDK from wherever it has been downloaded. Then launch Etcher. Click "Select image" and go to the extracted images folder. Select "sd-blob-b01.img. Then click "Select Target," selecting the formatted SD card. Then click "Flash". See Figure B.2 for what this should look like.

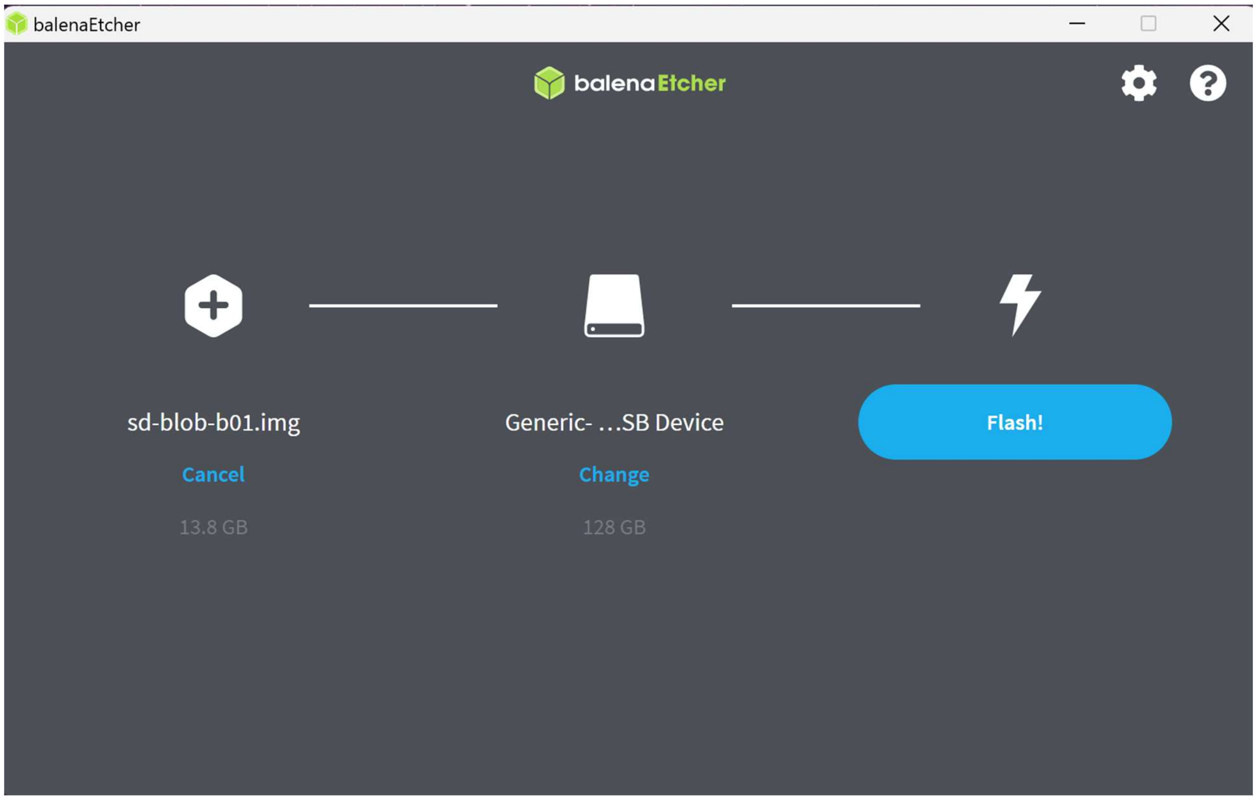
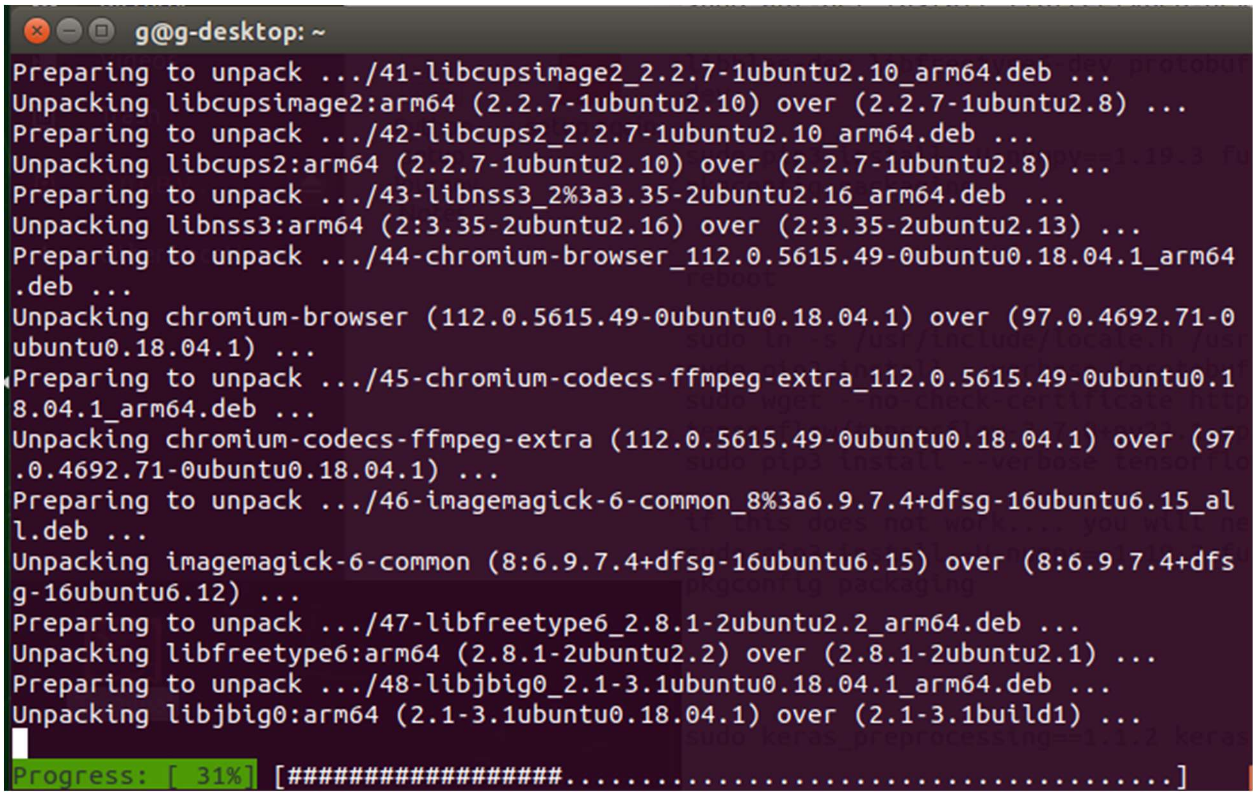


Figure B.2

Balena Etcher Setup

Once finished flashing the device, take the SD card out and plug it into the Jetson Nano. Power on the Nano; a prompt will appear with the first-time setup information. Accept the terms of Nvidia's licenses and the other system configurations like language and English keyboard. For the username, we used g and set a password; for the APP, the partition size is set to the maximum megabytes. For Nvpmodel mode select MAXN. This should begin the system configuration process. As mentioned earlier, the Nano is a bit trickier to set up than the Raspberry Pi, so we have a few more commands to execute. Once logged in, run sudo apt update && sudo apt full-upgrade. Figure B.3 – Full System Update – Linux



```
Preparing to unpack .../41-libcupsimage2_2.2.7-1ubuntu2.10_arm64.deb ...
Unpacking libcupsimage2:arm64 (2.2.7-1ubuntu2.10) over (2.2.7-1ubuntu2.8) ...
Preparing to unpack .../42-libcups2_2.2.7-1ubuntu2.10_arm64.deb ...
Unpacking libcups2:arm64 (2.2.7-1ubuntu2.10) over (2.2.7-1ubuntu2.8) ...
Preparing to unpack .../43-libnss3_2%3a3.35-2ubuntu2.16_arm64.deb ...
Unpacking libnss3:arm64 (2:3.35-2ubuntu2.16) over (2:3.35-2ubuntu2.13) ...
Preparing to unpack .../44-chromium-browser_112.0.5615.49-0ubuntu0.18.04.1_arm64
.deb ...
Unpacking chromium-browser (112.0.5615.49-0ubuntu0.18.04.1) over (97.0.4692.71-0
ubuntu0.18.04.1) ...
Preparing to unpack .../45-chromium-codecs-ffmpeg-extra_112.0.5615.49-0ubuntu0.1
8.04.1_arm64.deb ...
Unpacking chromium-codecs-ffmpeg-extra (112.0.5615.49-0ubuntu0.18.04.1) over (97
.0.4692.71-0ubuntu0.18.04.1) ...
Preparing to unpack .../46-imagemagick-6-common_8%3a6.9.7.4+dfsg-16ubuntu6.15_al
l.deb ...
Unpacking imagemagick-6-common (8:6.9.7.4+dfsg-16ubuntu6.15) over (8:6.9.7.4+dfs
g-16ubuntu6.12) ...
Preparing to unpack .../47-libfreetype6_2.8.1-2ubuntu2.2_arm64.deb ...
Unpacking libfreetype6:arm64 (2.8.1-2ubuntu2.2) over (2.8.1-2ubuntu2.1) ...
Preparing to unpack .../48-libjbig0_2.1-3.1ubuntu0.18.04.1_arm64.deb ...
Unpacking libjbig0:arm64 (2.1-3.1ubuntu0.18.04.1) over (2.1-3.1build1) ...

Progress: [ 31%] [##################...................................]
```

Figure B.3

Full system Update - Linux

A prompt will ask if you want to continue typing y and hitting enter. During the update, a prompt asking to update nvidia-tegra.conf and nv-oem-config-post. Select Y. Another propt will also ask to restart the Docker daemon. Select Yes. Once done, run reboot, go back into the terminal, and run sudo apt install --fix-broken -o Dpkg::Options::="--force-overwrite". Then, type reboot. Once rebooted, we will begin installing the dependencies. To install the dependinces, we have a few lines of code we will need to type up. Run the following lines of code in this exact order, as the dependency manager on the Nano does not appear to be as robust as the Raspberry Pi's. Thus, if programs are not installed in the right order, things may not work as intended. Run the following lines of code:

```
sudo apt-get install git cmake

sudo apt-get install python3-dev

sudo apt-get install python3-pip

python3 -m pip install --upgrade pip

pip3 install -U pip testresources setuptools

sudo apt-get install libfreetype6-dev python3-setuptools libatlas-base-dev libhdf5-serial-dev hdf5-tools libhdf5-dev gfortran libc-ares-dev libeigen3-dev zlib1g-dev zip libjpeg8-dev liblapack-dev libblas-dev libfreetype6-dev protobuf-compiler libprotobuf-dev openssl libssl-dev libcurl4-openssl-dev
```

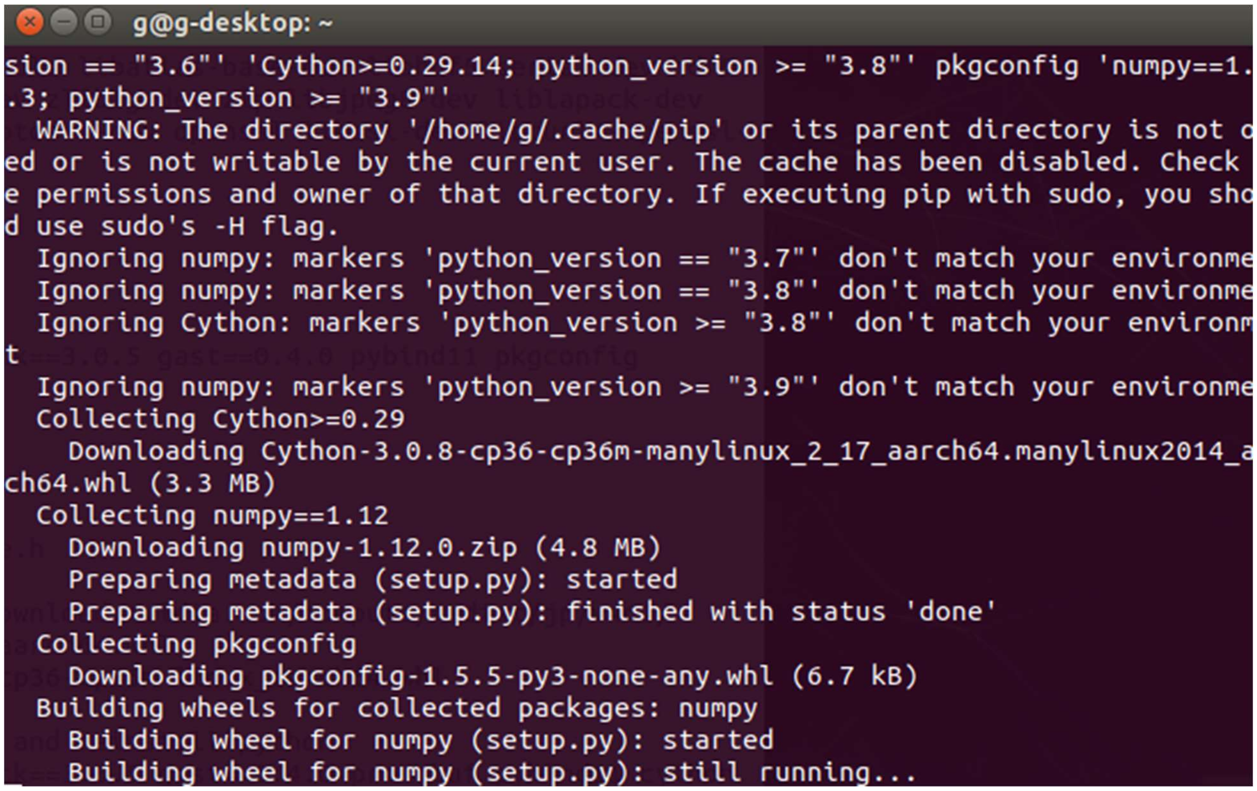
```
pip3 install -U numpy==1.19.3 future==0.18.2 mock==3.0.5 gast==0.4.0 protobuf pybind11
pkgconfig packaging
```

After running these lines of commands, type reboot.

Now that most of the dependencies are installed. Begin downloading TensorFlow and Keras onto the Jetson Nano by opening up a terminal and running the following commands:

```
Sudo ln -s /usr/include/locale.h /usr/include/xlocale.h

pip3 install --verbose 'protobuf<4' 'Cython<3'


pip3 install --extra-index-url https://developer.download.nvidia.com/compute/redist/jp/v46
tensorflow==2.6.2+nv21.12

pip3 install keras==2.6
```

During the installation of Tensorflow 2.6.2, there will be a long wait for a wheel of numpy version 1.12 to be built. This is shown in Figure B.4 – Numpy Wheel. Trust the process, and it will eventually get done.



```
sion == "3.6"' 'Cython>=0.29.14; python_version >= "3.8"' pkgconfig 'numpy==1.
.3; python_version >= "3.9"'
  WARNING: The directory '/home/g/.cache/pip' or its parent directory is not o
ed or is not writable by the current user. The cache has been disabled. Check
e permissions and owner of that directory. If executing pip with sudo, you sho
d use sudo's -H flag.
  Ignoring numpy: markers 'python_version == "3.7"' don't match your environme
  Ignoring numpy: markers 'python_version == "3.8"' don't match your environme
  Ignoring Cython: markers 'python_version >= "3.8"' don't match your environm
t
  Ignoring numpy: markers 'python_version >= "3.9"' don't match your environme
  Collecting Cython>=0.29
    Downloading Cython-3.0.8-cp36-cp36m-manylinux_2_17_aarch64.manylinux2014_a
ch64.whl (3.3 MB)
  Collecting numpy==1.12
    Downloading numpy-1.12.0.zip (4.8 MB)
    Preparing metadata (setup.py): started
    Preparing metadata (setup.py): finished with status 'done'
  Collecting pkgconfig
    Downloading pkgconfig-1.5.5-py3-none-any.whl (6.7 kB)
  Building wheels for collected packages: numpy
    Building wheel for numpy (setup.py): started
    Building wheel for numpy (setup.py): still running...
```

Figure B.4

Numpy Wheel

After Tensorflow and Keras are installed, we will continue to download a few more dependencies for the code base by running the following commands:

pip3 install matplotlib augmentor split-folders pyyaml keyboard scikit-learn

sudo pip install scipy==1.5.4

sudo apt-get install nano

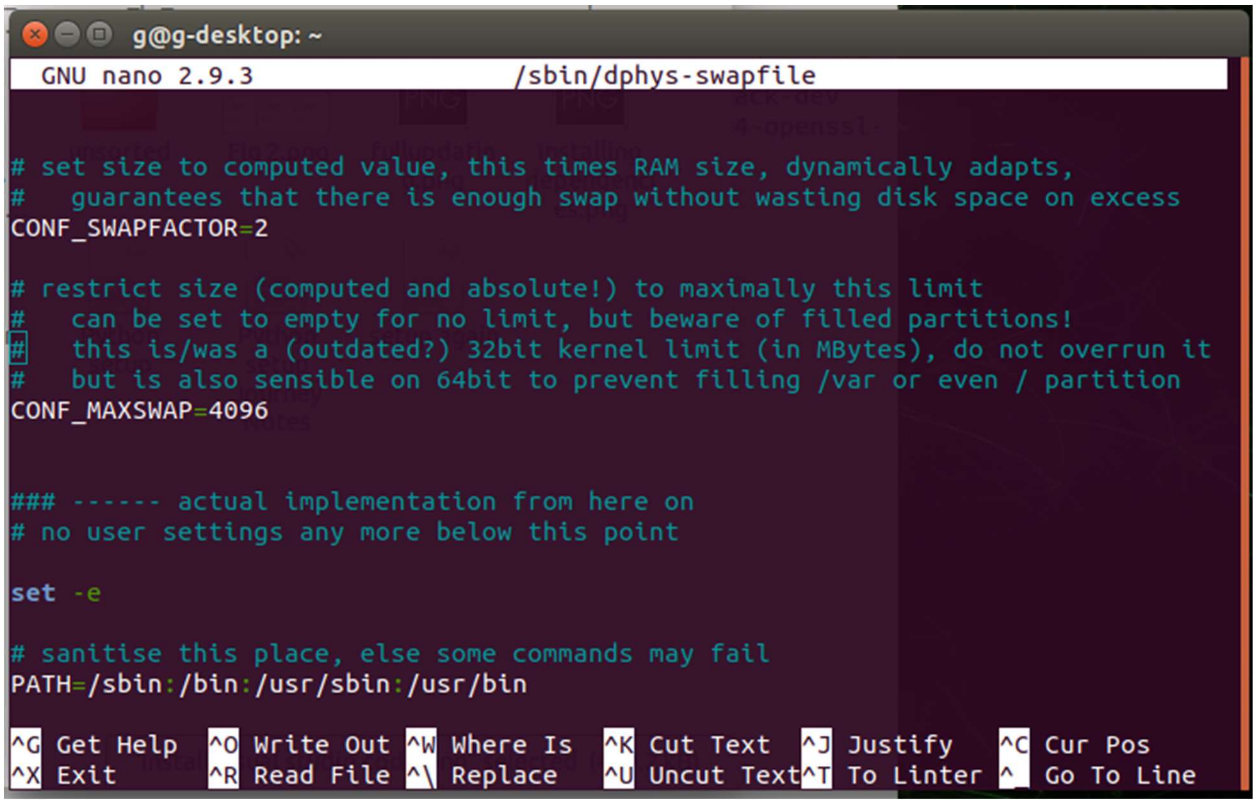
sudo apt-get install dphys-swapfile

pip3 install nanocamera

reboot

After these are installed, the final two dependencies can be installed: a OpenCV version allowing us to use Nvidia's CUDA cores. First, build OpenCV code provided by QEngineering.eu. This section will cover the steps to install this code; otherwise, for more information, please go to their website and read more about it.

Run the command sudo nano /sbin/dphys-swapfile. This will open the physical swapfile location and scroll down to the CONF_MAXSWAP variable. Change its value to 4096. Figure B.5 – sbin/dphys-swapfile edit shows what this should look like. Hit "CTRL+X" to save and exit the file.



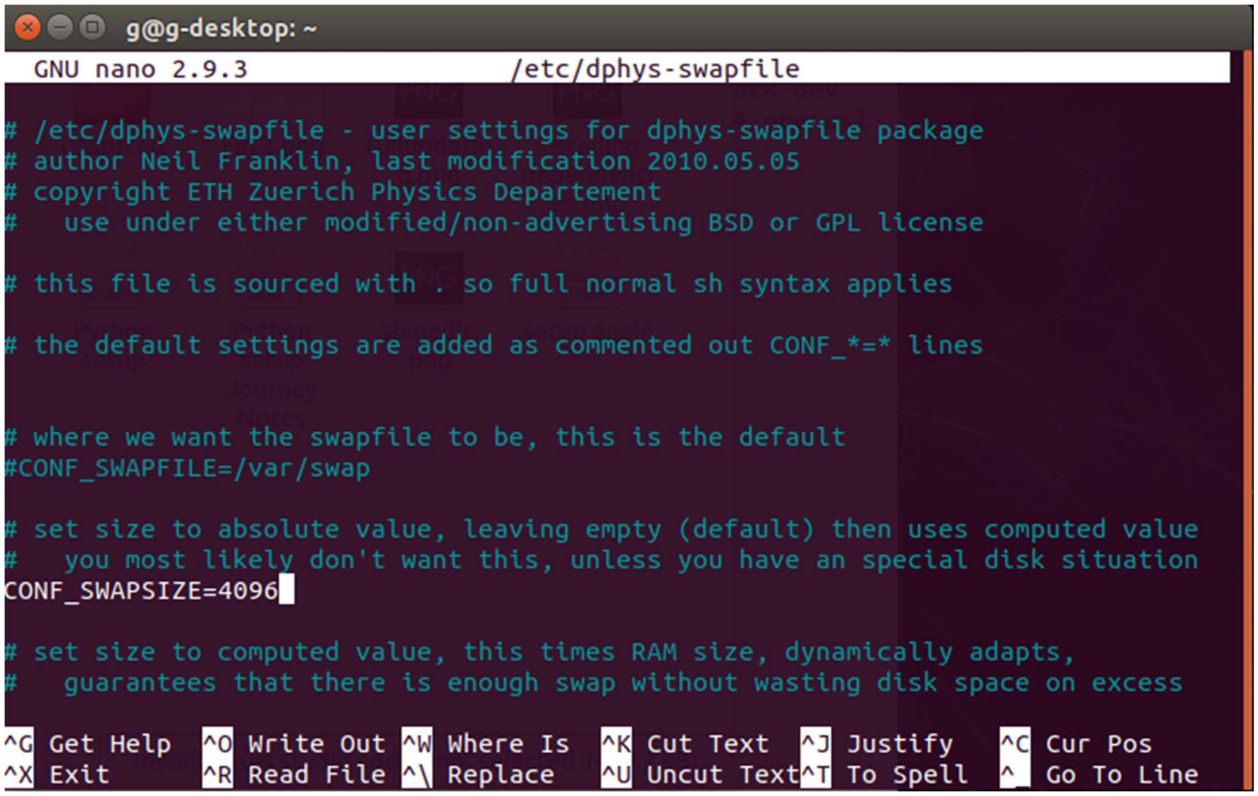
```
g@g-desktop: ~
  GNU nano 2.9.3                  /sbin/dphys-swapfile

# set size to computed value, this times RAM size, dynamically adapts,
#   guarantees that there is enough swap without wasting disk space on excess
CONF_SWAPFACTOR=2

# restrict size (computed and absolute!) to maximally this limit
#   can be set to empty for no limit, but beware of filled partitions!
#   this is/was a (outdated?) 32bit kernel limit (in MBytes), do not overrun it
#   but is also sensible on 64bit to prevent filling /var or even / partition
CONF_MAXSWAP=4096


### ------ actual implementation from here on
# no user settings any more below this point

set -e

# sanitise this place, else some commands may fail
PATH=/sbin:/bin:/usr/sbin:/usr/bin

^G Get Help   ^O Write Out  ^W Where Is   ^K Cut Text   ^J Justify    ^C Cur Pos
^X Exit       ^R Read File  ^\ Replace    ^U Uncut Text^T To Linter   ^_ Go To Line
```

Figure B.5

sbin/dphys-swapfile edit

We now also need to edit the /etc/dphys-swapfile settings as well. Run the command sudo nano /etc/dphys-swapfile.

Find and uncomment (remove the #) in the file and add 4096 to the variable, as shown in Figure B.6 – etc/dphys-swapfile edit. Just like before, hit "Ctrl+x" to exit and save the file.

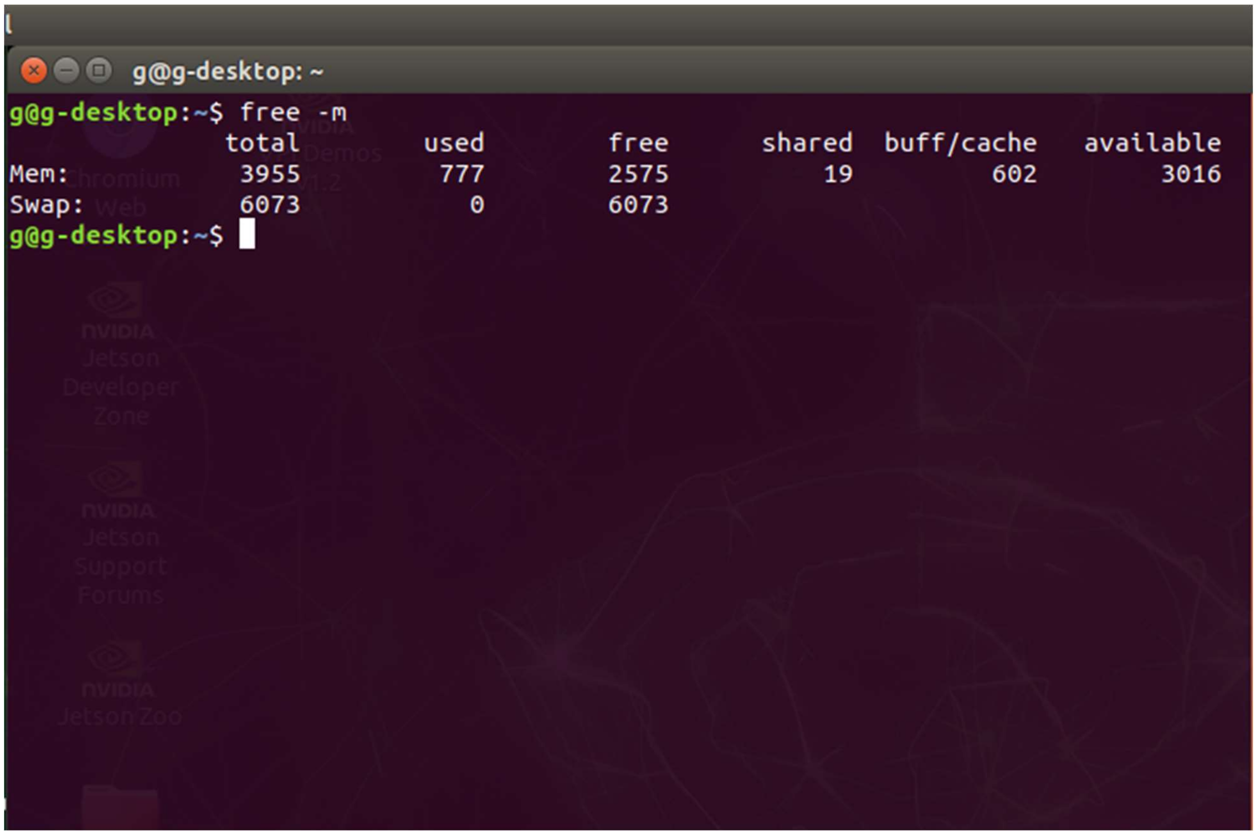


```
g@g-desktop: ~
  GNU nano 2.9.3                    /etc/dphys-swapfile

# /etc/dphys-swapfile - user settings for dphys-swapfile package
# author Neil Franklin, last modification 2010.05.05
# copyright ETH Zuerich Physics Departement
#   use under either modified/non-advertising BSD or GPL license

# this file is sourced with . so full normal sh syntax applies

# the default settings are added as commented out CONF_*=* lines


# where we want the swapfile to be, this is the default
#CONF_SWAPFILE=/var/swap

# set size to absolute value, leaving empty (default) then uses computed value
#   you most likely don't want this, unless you have an special disk situation
CONF_SWAPSIZE=4096

# set size to computed value, this times RAM size, dynamically adapts,
#   guarantees that there is enough swap without wasting disk space on excess

^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Uncut Text^T To Spell  ^_ Go To Line
```

Figure B.6

etc/dphys-swapfile edit

After saving the etc/dphys-swapfile file, run the reboot command. Once rebooted, reopen the terminal and run the command free -m. The output of which should match Figure B.7 – Correct Memory Management.



```
g@g-desktop:~$ free -m
              total        used        free      shared  buff/cache   available
Mem:           3955         777        2575          19         602        3016
Swap:          6073           0        6073
g@g-desktop:~$
```

Figure B.7

Correct Memory Management

Next, we will run the commands to start installing the custom OpenCV install for the Jetson Nano. A word of warning: this, at a minimum, will take three and a half hours. In some cases, it takes up to twelve hours to build and install. Run the following commands in the terminal:

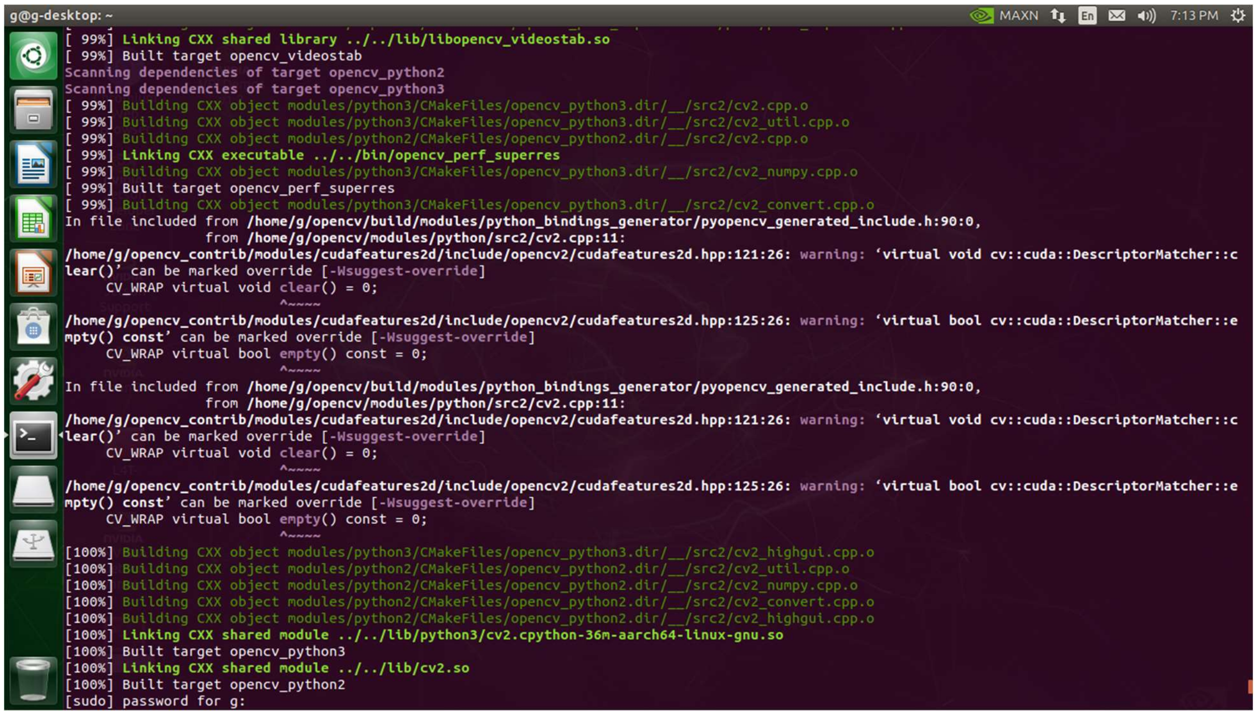
wget https://github.com/Qengineering/Install-OpenCV-Jetson-Nano/raw/main/OpenCV-4-9-0.sh

sudo chmod 755 ./OpenCV-4-9-0.sh

./OpenCV-4-9-0.sh

Once the codebase has been built successfully, a screen similar to that in Figure B.8 will appear.

Enter the root password and let it finish installing.



```
g@g-desktop: ~
[ 99%] Linking CXX shared library ../../lib/libopencv_videostab.so
[ 99%] Built target opencv_videostab
Scanning dependencies of target opencv_python2
Scanning dependencies of target opencv_python3
[ 99%] Building CXX object modules/python3/CMakeFiles/opencv_python3.dir/__/src2/cv2.cpp.o
[ 99%] Building CXX object modules/python3/CMakeFiles/opencv_python3.dir/__/src2/cv2_util.cpp.o
[ 99%] Building CXX object modules/python2/CMakeFiles/opencv_python2.dir/__/src2/cv2.cpp.o
[ 99%] Linking CXX executable ../../bin/opencv_perf_superres
[ 99%] Building CXX object modules/python3/CMakeFiles/opencv_python3.dir/__/src2/cv2_numpy.cpp.o
[ 99%] Built target opencv_perf_superres
[ 99%] Building CXX object modules/python3/CMakeFiles/opencv_python3.dir/__/src2/cv2_convert.cpp.o
In file included from /home/g/opencv/build/modules/python_bindings_generator/pyopencv_generated_include.h:90:0,
                 from /home/g/opencv/modules/python/src2/cv2.cpp:11:
/home/g/opencv_contrib/modules/cudafeatures2d/include/opencv2/cudafeatures2d.hpp:121:26: warning: ‘virtual void cv::cuda::DescriptorMatcher::c
lear()’ can be marked override [-Wsuggest-override]
     CV_WRAP virtual void clear() = 0;
                          ^~~~~
/home/g/opencv_contrib/modules/cudafeatures2d/include/opencv2/cudafeatures2d.hpp:125:26: warning: ‘virtual bool cv::cuda::DescriptorMatcher::e
mpty() const’ can be marked override [-Wsuggest-override]
     CV_WRAP virtual bool empty() const = 0;
                          ^~~~~
In file included from /home/g/opencv/build/modules/python_bindings_generator/pyopencv_generated_include.h:90:0,
                 from /home/g/opencv/modules/python/src2/cv2.cpp:11:
/home/g/opencv_contrib/modules/cudafeatures2d/include/opencv2/cudafeatures2d.hpp:121:26: warning: ‘virtual void cv::cuda::DescriptorMatcher::c
lear()’ can be marked override [-Wsuggest-override]
     CV_WRAP virtual void clear() = 0;
                          ^~~~~
/home/g/opencv_contrib/modules/cudafeatures2d/include/opencv2/cudafeatures2d.hpp:125:26: warning: ‘virtual bool cv::cuda::DescriptorMatcher::e
mpty() const’ can be marked override [-Wsuggest-override]
     CV_WRAP virtual bool empty() const = 0;
                          ^~~~~
[100%] Building CXX object modules/python3/CMakeFiles/opencv_python3.dir/__/src2/cv2_highgui.cpp.o
[100%] Building CXX object modules/python2/CMakeFiles/opencv_python2.dir/__/src2/cv2_util.cpp.o
[100%] Building CXX object modules/python2/CMakeFiles/opencv_python2.dir/__/src2/cv2_numpy.cpp.o
[100%] Building CXX object modules/python2/CMakeFiles/opencv_python2.dir/__/src2/cv2_convert.cpp.o
[100%] Building CXX object modules/python2/CMakeFiles/opencv_python2.dir/__/src2/cv2_highgui.cpp.o
[100%] Linking CXX shared module ../../lib/python3/cv2.cpython-36m-aarch64-linux-gnu.so
[100%] Built target opencv_python3
[100%] Linking CXX shared module ../../lib/cv2.so
[100%] Built target opencv_python2
[sudo] password for g:
```

Figure B.8

OpenCV Successfully Built

After OpenCV has been successfully installed. Run the following commands:

rm OpenCV-4-9-0.sh

sudo /etc/init.d/dphys-swapfile stop

sudo apt-get remove --purge dphys-swapfile

sudo rm -rf ~/opencv

sudo rm -rf ~/opencv_contrib

reboot

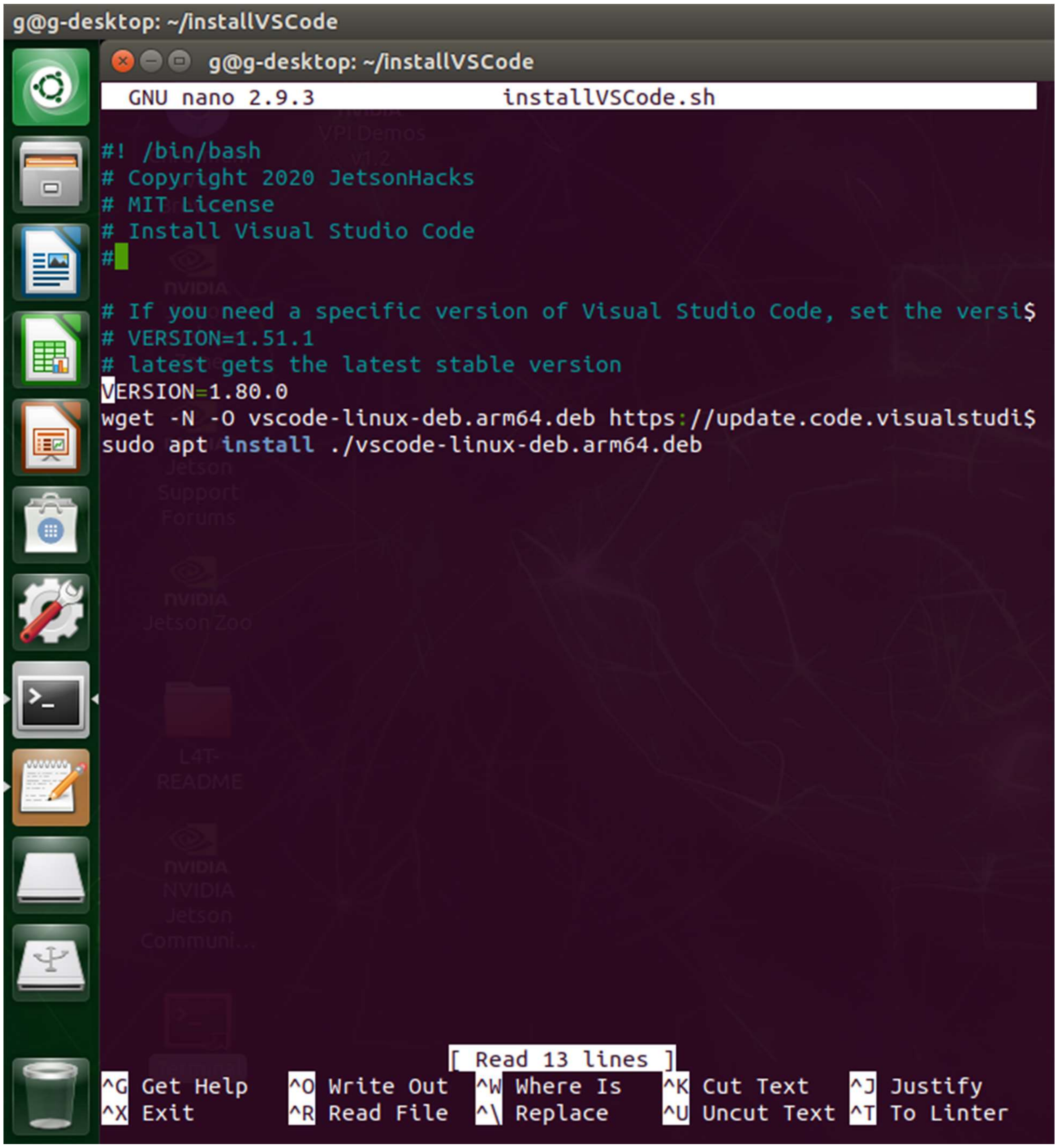
There is a bit more of a process to get the development environment installed and working. This is since Visual Studio Code is not natively supported on the Nano. To install, run the following commands:

```
git clone https://github.com/JetsonHacksNano/installVSCode.git
```

```
cd installVSCode
```

```
nano instalVSCodeWithPython.sh./installVSCodeWithPython.sh
```

Since, at the time of writing, the latest version of Visual Studio Code did not work, we needed to set the .sh file to install a known good version, which is currently 1.80.0. Figure B.9 - Visual Studio Code Setup shows what this should look like. After adding in 1.80.0, hit "Ctrl + x" and save the file. Then run the command ./installVSCodeWithPython.sh. After this, Visual Studio can be run by typing code into the terminal.



```
g@g-desktop: ~/installVSCode
  GNU nano 2.9.3                  installVSCode.sh

#! /bin/bash
# Copyright 2020 JetsonHacks
# MIT License
# Install Visual Studio Code
#

# If you need a specific version of Visual Studio Code, set the versi$
# VERSION=1.51.1
# latest gets the latest stable version
VERSION=1.80.0
wget -N -O vscode-linux-deb.arm64.deb https://update.code.visualstudi$
sudo apt install ./vscode-linux-deb.arm64.deb

                            [ Read 13 lines ]
^G Get Help    ^O Write Out   ^W Where Is    ^K Cut Text    ^J Justify
^X Exit        ^R Read File   ^\ Replace     ^U Uncut Text  ^T To Linter
```

Figure B.9

Visual Studio Code Setup

Once all the Visual Studio code and the dependencies are installed, copy the AI_Image_Solar Folder from AI_Image_Solar_pi_Jetson_Nano to /home/. Return to Visual Studio Code, Go to the explorer tab, and open the AI_Image_Solar Folder by clicking the open folder, selecting the AI_Image_Solar folder, and clicking open. Run Predict_N_Sort.py to verify the code base is working. We should see the unsorted folder is now empty, and the test images are sorted into the correct categories.

VITA

Garrick Daniel Muncie was born in Detroit, MI. He is the first of two children. He attended the Academy of Information Technology at Apex High School in Apex, North Carolina. After graduation, he attended Tennessee Technological University, where he became interested in Computer Engineering. He completed the Bachelor of Science degree in December 2016 in Computer Engineering with a minor in Mathematics. Garrick is continuing his education in engineering by pursuing a master's degree at the University of Tennessee Chattanooga.