Project Report

FOD Object Detection Model (MobileNet)

1. Introduction

Foreign object debris, known as FOD, is an ever-increasing concern in the aerospace industry. Causing nearly $13 billion in direct and indirect costs in the last year, the damage done by FOD presents a danger to the aircraft and the people exposed to the risk. FOD includes debris, animals, or any other foreign substance in an operating environment that could cause substantial damage. Many measures have been previously deployed to protect against FOD such as sweeping, magnetic bars, and detection cameras. Proposed detection methods include video-based detection from cameras mounted on the edge of the runway, millimeter-wave radar, and convolutional neural network object detection models. Object detection models can be used with proprietary cameras to develop a real time processing solution to observe FOD on the runway, maintenance areas, and storage hangars. The object detection model can also be used in parallel with other protection methods to decrease the risk of FOD causing damage to aircraft and people.

This report presents a newly trained FOD detection model based on a set of images that can be used in accordance with a camera pointed straight down at the pavement from a height of 1-3 meters. The intended format would be a vehicle moving across the pavement with the object detector camera attached to it, giving real-time alerts to the location of FOD. Secondarily, the camera can be posted high on a pole for a stationary view over a runway/taxiway.

The objective of this paper is to create a portable, trained object detector based in convolutional learning. This means that the neural network should be modular in nature and be importable to standard TensorFlow and PyTorch models through the Open Neural Network Exchange (ONNX) program.

Creating a network from scratch would be extremely time-consuming and open access models do exist as an acceptable substitute for general object detection. These models are pretrained on their own datasets of hundreds of thousands or even millions of images. A fast network is also preferable as the time to transfer a network to a new dataset takes hours or even days

2. Dataset

In order for the neural network to be trained, training data must be accrued and labeled by class or type of image. The FOD-UNOmaha Image Dataset is a collection of 33792 image frames taken from video and divided into 31 object classes. These classes include common FOD like bolts, soda cans, and small pieces of metal and plastic.

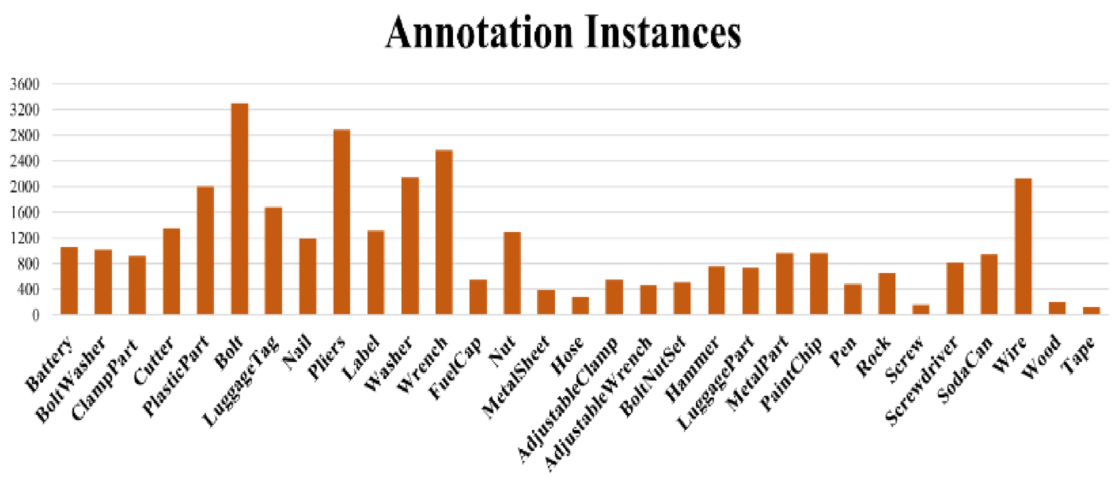
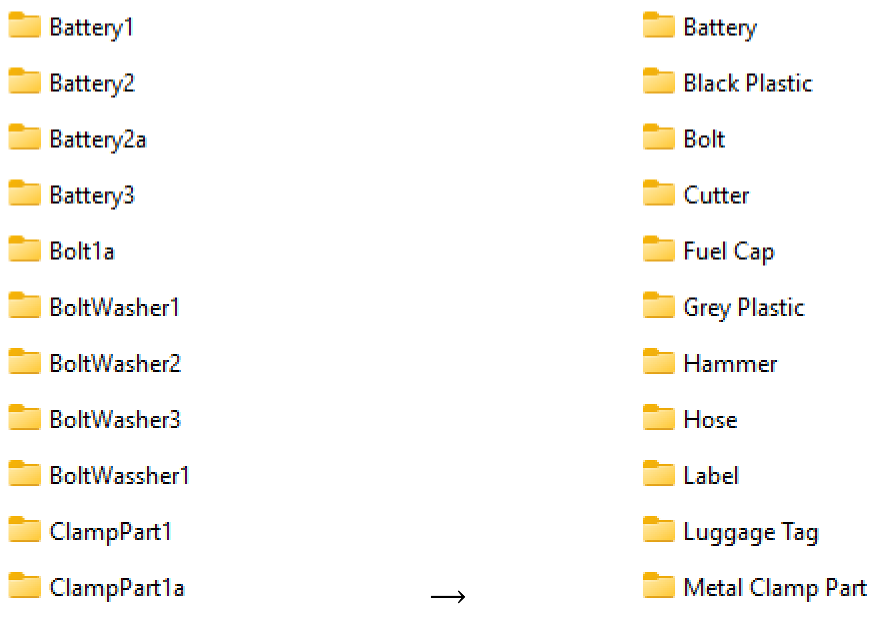


Figure 1. This chart shows the occurrences of each image in the 31 object classes.

Attributes are also given to each image on top of the label. These attributes consist of information regarding the time of day and the dry/wet considerations. This is important because weather will have a visual impact on the outcome of the detection process.

3. Data Wrangling

The dataset needed to be modified to fit the data object TensorFlow needed for preprocessing. The folder structure needed to be flattened to only one layer instead of the multiple subdirectories present within the structure. This means that the categories needed to be combined for general classification. This works well in our favor as this means different types of each object can be generalized combating overfitting.



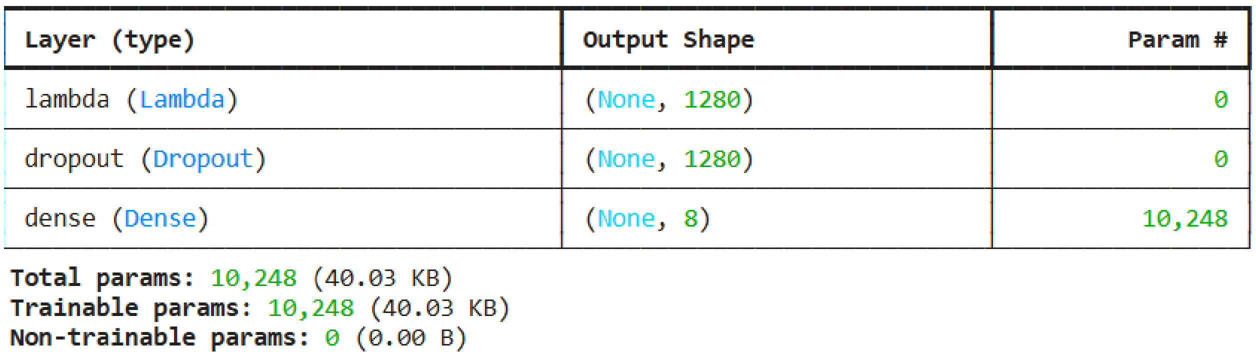
Battery1	Battery
Battery2	Black Plastic
Battery2a	Bolt
Battery3	Cutter
Bolt1a	Fuel Cap
BoltWasher1	Grey Plastic
BoltWasher2	Hammer
BoltWasher3	Hose
BoltWassher1	Label
ClampPart1	Luggage Tag
ClampPart1a	Metal Clamp Part

$\longrightarrow$

4. Preprocessing

Before training the dataset, there are a few choices to be made regarding the parameters. The training/validation split will be set to 80/20, the batch size will be set to 32 before updating the tuning parameters, the label mode will be set to categorical (meaning the labels will be set to the names of the folders for the images), and data augmentation will be conducted.

Since the idea for the model is to be deployed on an airfield, the images must be augmented to represent the variable height at which these cameras might be at. One way to do this is to modify the images by rotating, translating, and zooming in and out in random ways. This extorts the value out of our dataset even further.



Layer (type)	Output Shape	Param #
lambda (Lambda)	(None, 1280)	0
dropout (Dropout)	(None, 1280)	0
dense (Dense)	(None, 8)	10,248

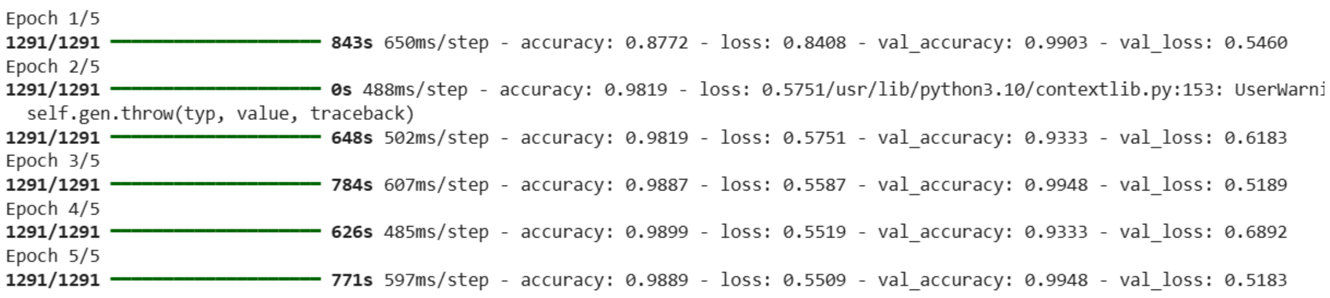
```
Total params: 10,248 (40.03 KB)
Trainable params: 10,248 (40.03 KB)
Non-trainable params: 0 (0.00 B)
```

Figure 2. This chart shows the topology of the network. Three layers are present with the MobileNet model contained entirely within the lambda layer.

Our specialized three-layer model has been created. The lambda layer contains the entire MobileNet framework which includes all its layers. The dropout and dense layer are added after the lambda layer. We can compile the model and see the initial output.

5. Model and Results

--

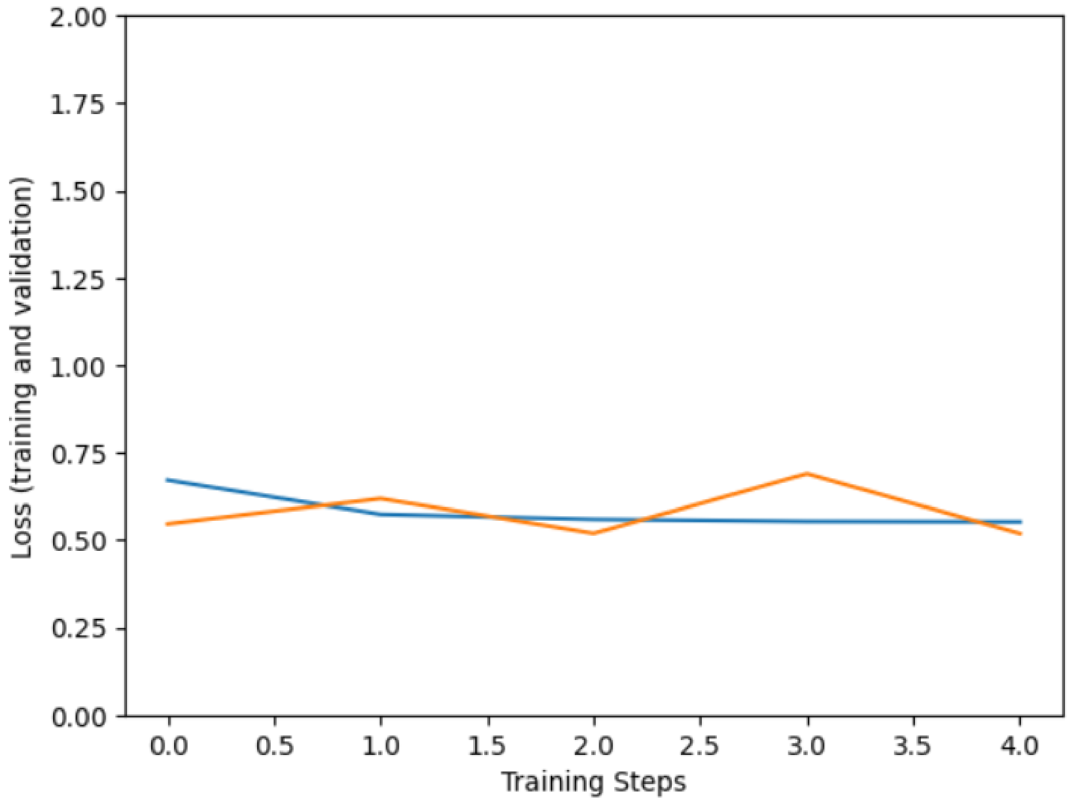


```
Epoch 1/5
1291/1291 ━━━━━━━━━━━━━━━━━━━━ 843s 650ms/step - accuracy: 0.8772 - loss: 0.8408 - val_accuracy: 0.9903 - val_loss: 0.5460
Epoch 2/5
1291/1291 ━━━━━━━━━━━━━━━━━━━━ 0s 488ms/step - accuracy: 0.9819 - loss: 0.5751/usr/lib/python3.10/contextlib.py:153: UserWarni
  self.gen.throw(typ, value, traceback)
1291/1291 ━━━━━━━━━━━━━━━━━━━━ 648s 502ms/step - accuracy: 0.9819 - loss: 0.5751 - val_accuracy: 0.9333 - val_loss: 0.6183
Epoch 3/5
1291/1291 ━━━━━━━━━━━━━━━━━━━━ 784s 607ms/step - accuracy: 0.9887 - loss: 0.5587 - val_accuracy: 0.9948 - val_loss: 0.5189
Epoch 4/5
1291/1291 ━━━━━━━━━━━━━━━━━━━━ 626s 485ms/step - accuracy: 0.9899 - loss: 0.5519 - val_accuracy: 0.9333 - val_loss: 0.6892
Epoch 5/5
1291/1291 ━━━━━━━━━━━━━━━━━━━━ 771s 597ms/step - accuracy: 0.9889 - loss: 0.5509 - val_accuracy: 0.9948 - val_loss: 0.5183
```

--

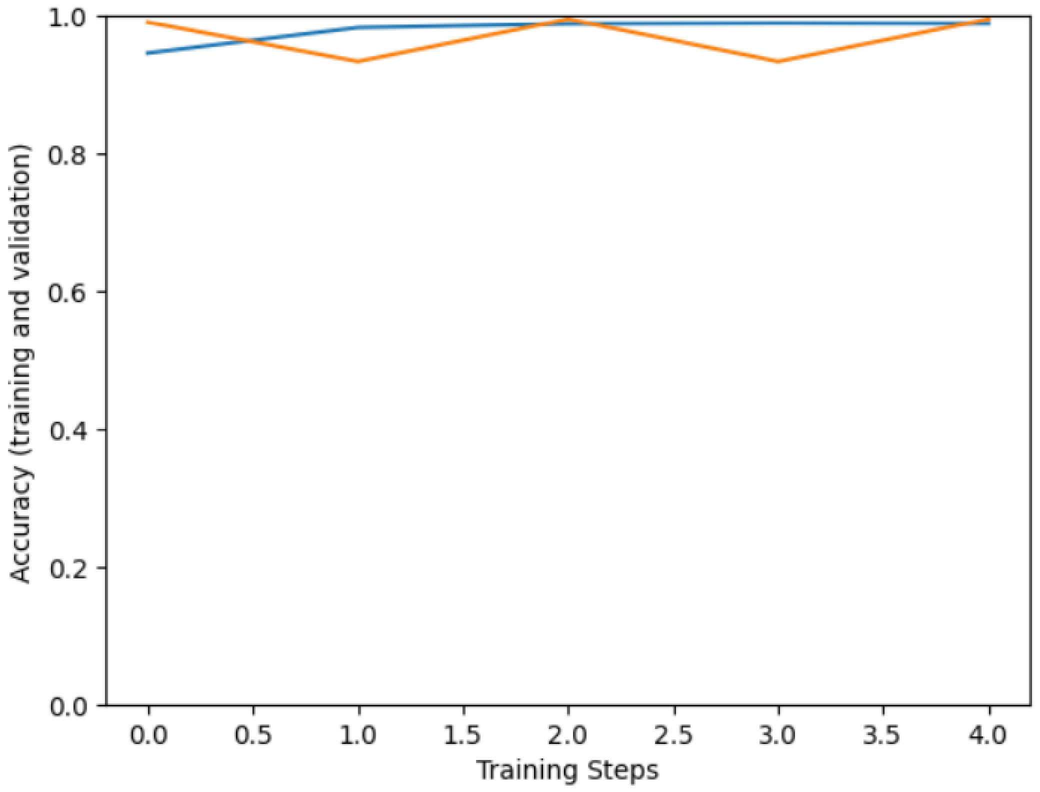
Figure 3. The history of the training of the model is shown. The final accuracy score is shown during epoch 5, the final epoch, along with the other metrics.

The final accuracy score is highlighted as 98.89%, which is very good especially after only five epochs. Let's graph the loss.



--

Figure 4. The blue line is the prediction on the training set and the orange line is the validation loss prediction. There seems to be a peak at the third epoch before going down again.



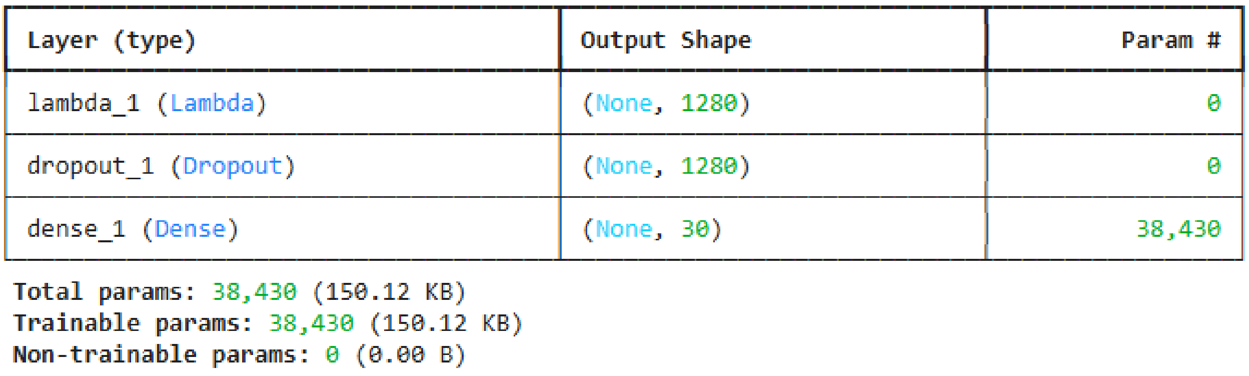
--

Figure. 5. This is the accuracy. The blue line is the actual and the orange line is the accuracy on the validation set. Both near perfect.

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 134ms/step
True label: Metal
Predicted label: Metal
```

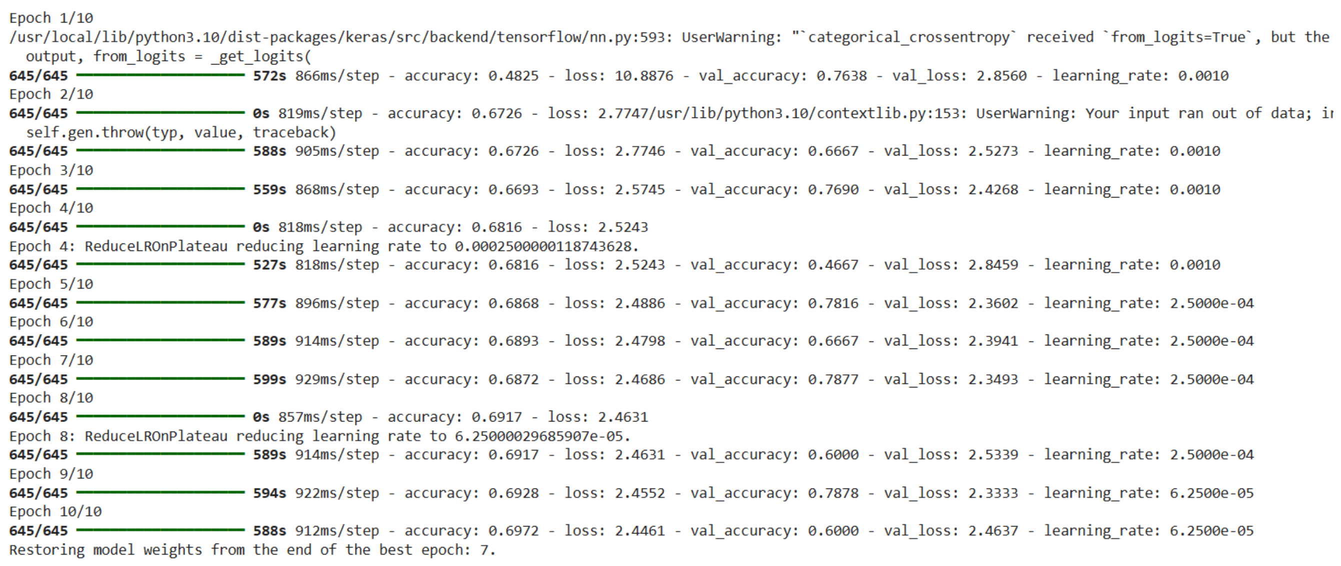
--

Out of an abundance of caution for overfitting and possible not quality data, I will introduce more random data transformation in the preprocessing before compiling the model. Let's also run ten epochs to consider a more comprehensive image collection.



Layer (type)	Output Shape	Param #
lambda_1 (Lambda)	(None, 1280)	0
dropout_1 (Dropout)	(None, 1280)	0
dense_1 (Dense)	(None, 30)	38,430

```
Total params: 38,430 (150.12 KB)
Trainable params: 38,430 (150.12 KB)
Non-trainable params: 0 (0.00 B)
```

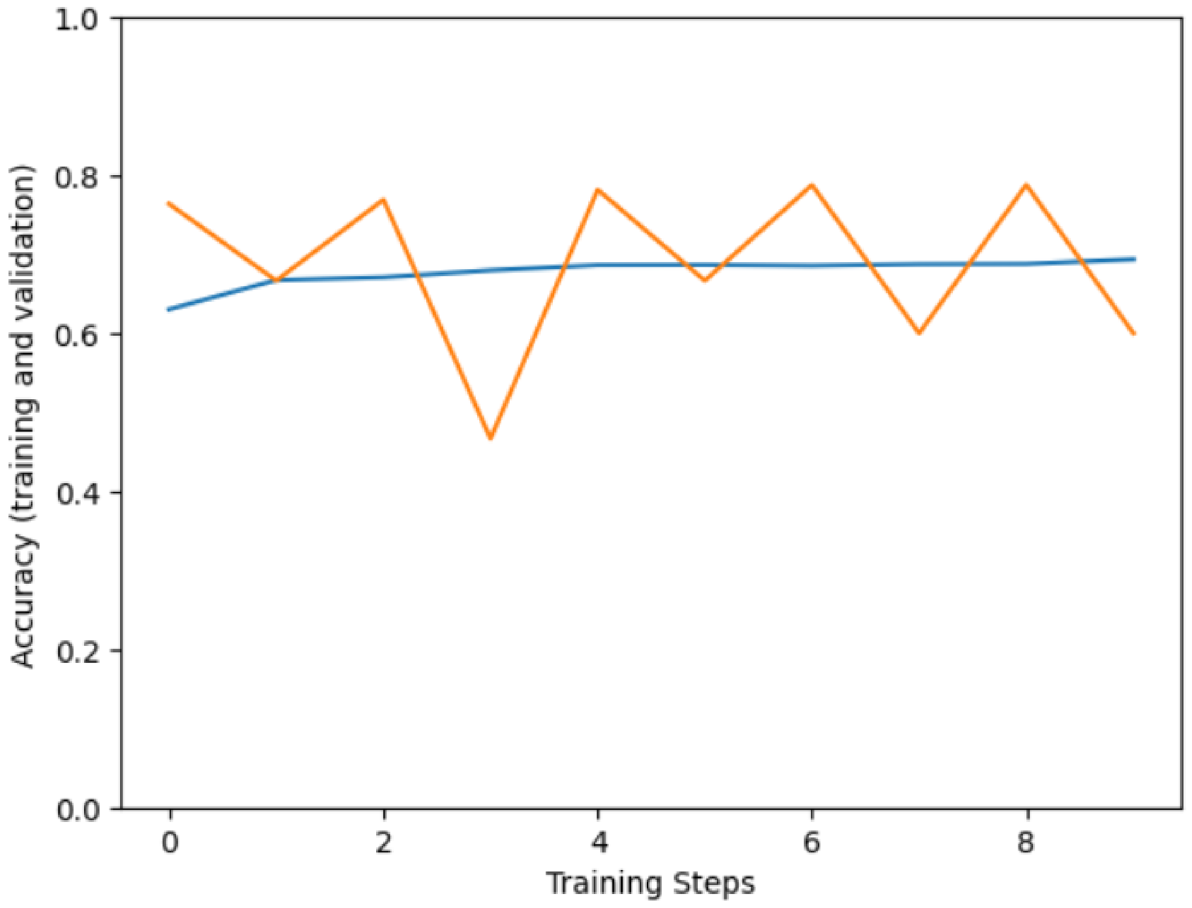


```
Epoch 1/10
/usr/local/lib/python3.10/dist-packages/keras/src/backend/tensorflow/nn.py:593: UserWarning: "`categorical_crossentropy` received `from_logits=True`, but the
  output, from_logits = _get_logits(
645/645 ━━━━━━━━━━━━━━━━━━━━ 572s 866ms/step - accuracy: 0.4825 - loss: 10.8876 - val_accuracy: 0.7638 - val_loss: 2.8560 - learning_rate: 0.0010
Epoch 2/10
645/645 ━━━━━━━━━━━━━━━━━━━━ 0s 819ms/step - accuracy: 0.6726 - loss: 2.7747/usr/lib/python3.10/contextlib.py:153: UserWarning: Your input ran out of data; i
  self.gen.throw(typ, value, traceback)
645/645 ━━━━━━━━━━━━━━━━━━━━ 588s 905ms/step - accuracy: 0.6726 - loss: 2.7746 - val_accuracy: 0.6667 - val_loss: 2.5273 - learning_rate: 0.0010
Epoch 3/10
645/645 ━━━━━━━━━━━━━━━━━━━━ 559s 868ms/step - accuracy: 0.6693 - loss: 2.5745 - val_accuracy: 0.7690 - val_loss: 2.4268 - learning_rate: 0.0010
Epoch 4/10
645/645 ━━━━━━━━━━━━━━━━━━━━ 0s 818ms/step - accuracy: 0.6816 - loss: 2.5243
Epoch 4: ReduceLROnPlateau reducing learning rate to 0.0002500000118743628.
645/645 ━━━━━━━━━━━━━━━━━━━━ 527s 818ms/step - accuracy: 0.6816 - loss: 2.5243 - val_accuracy: 0.4667 - val_loss: 2.8459 - learning_rate: 0.0010
Epoch 5/10
645/645 ━━━━━━━━━━━━━━━━━━━━ 577s 896ms/step - accuracy: 0.6868 - loss: 2.4886 - val_accuracy: 0.7816 - val_loss: 2.3602 - learning_rate: 2.5000e-04
Epoch 6/10
645/645 ━━━━━━━━━━━━━━━━━━━━ 589s 914ms/step - accuracy: 0.6893 - loss: 2.4798 - val_accuracy: 0.6667 - val_loss: 2.3941 - learning_rate: 2.5000e-04
Epoch 7/10
645/645 ━━━━━━━━━━━━━━━━━━━━ 599s 929ms/step - accuracy: 0.6872 - loss: 2.4686 - val_accuracy: 0.7877 - val_loss: 2.3493 - learning_rate: 2.5000e-04
Epoch 8/10
645/645 ━━━━━━━━━━━━━━━━━━━━ 0s 857ms/step - accuracy: 0.6917 - loss: 2.4631
Epoch 8: ReduceLROnPlateau reducing learning rate to 6.25000029685907e-05.
645/645 ━━━━━━━━━━━━━━━━━━━━ 589s 914ms/step - accuracy: 0.6917 - loss: 2.4631 - val_accuracy: 0.6000 - val_loss: 2.5339 - learning_rate: 2.5000e-04
Epoch 9/10
645/645 ━━━━━━━━━━━━━━━━━━━━ 594s 922ms/step - accuracy: 0.6928 - loss: 2.4552 - val_accuracy: 0.7878 - val_loss: 2.3333 - learning_rate: 6.2500e-05
Epoch 10/10
645/645 ━━━━━━━━━━━━━━━━━━━━ 588s 912ms/step - accuracy: 0.6972 - loss: 2.4461 - val_accuracy: 0.6000 - val_loss: 2.4637 - learning_rate: 6.2500e-05
Restoring model weights from the end of the best epoch: 7.
```

--

Figure 7. History is shown for the updated compilation. Scores are much lower overall this time around. The best score is at the end of epoch seven. This could mean an absolute max or a local max.

Let’s take a peek at the graphs for the scores.



--

Figure 9. The accuracy score is shown. Blue line is the training score while the orange line is the validation score. As shown, the validation scores don't converge yet which suggests that more training time is needed to reach an absolute minimum

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 5s 5s/step
True label: Cutter
Predicted label: Cutter
```

6. Conclusion

In this paper, we developed a computer vision model based on the MobileNet base and trained it on the FOD-A-UNOmaha dataset. Our best metrics for the updated model include a loss score of 2.34 and an accuracy score of 69.72%.

Longer training times can be utilized to achieve improved scores and further hyperparameter tuning can be done for lower loss. The random data augmentation is essential to this dataset in order to achieve the effect of a fielded object detector 5-10 meters on a pole. For robustness, the model can be potentially trained to take in video data instead of still frames. This would allow for real time data analysis and collection.

To test the model on FOD points, we can simulate the model on some datapoints using a .kml file from Google Earth. A scenario can be set up to test the model and its feasibility on an airstrip.

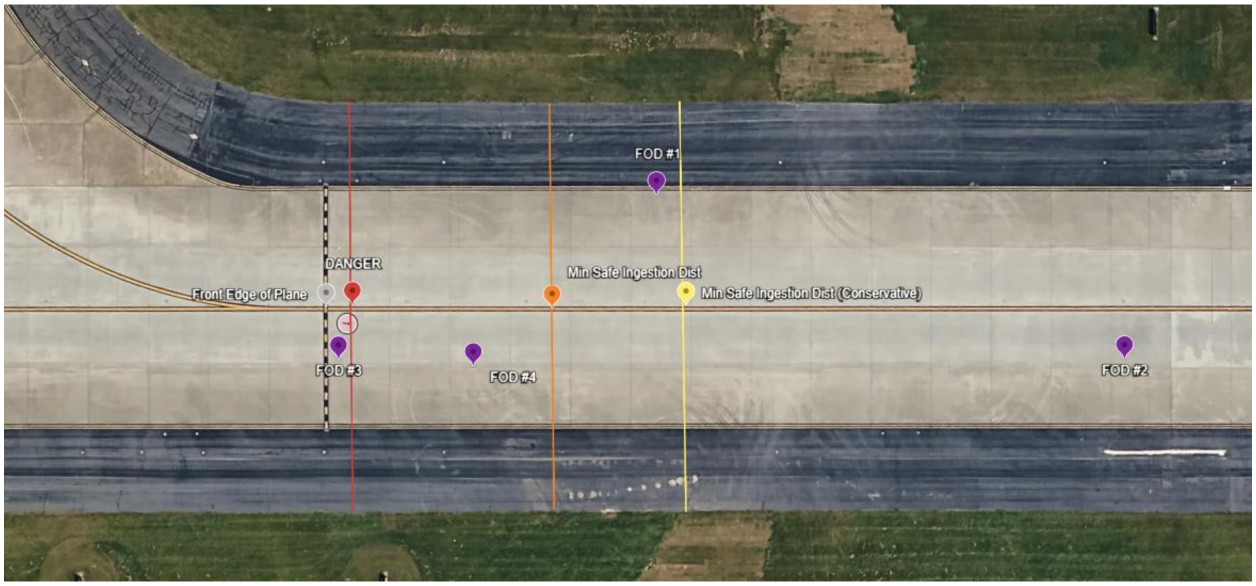


Figure 10. Points can be added to represent different data on a map. The purple points represent FOD. The red line represents a severity of very high risk, orange represents a high risk, and yellow represents a medium risk with anything further away from the aircraft being low risk.

A .kml file is basically a .xml file whose labels can be easily accessed. The Euclidean distance between points of interest and the aircraft can be obtained and calculated. A model can be combined with this program to develop a severity score. With this in mind, this program can be downloaded as a script on a IoT camera sensor for real time compilation of the data. Conceptually, an air traffic controller would receive the severity score and make quick adjustments by communicating with aircrew and emergency services to avoid a catastrophic failure.