Object Detection for Automated Doorstep Package Classification

1. Introduction

There is a long-standing business case in the United States for investing in technologies that allow consumers to keep tabs on their homes when they are away. Within the last five years, the home security company SimpliSafe and the smart doorbell company Ring have both surpassed a $1 billion valuation and combined serve over a million customers.[1,2] Specifically, the explosion of ecommerce and corresponding expansion of package theft present a number of potential uses for computer vision models that specialize in detecting packages and deliveries.[3]

This paper outlines an automated doorstep package detection system that uses deep learning to identify multiple classes of packages in real time. Our team generated an object detection dataset and augmented that data to become robust to adverse weather, motion, and lighting conditions. We trained a YOLOv5 object detection model in AWS and deployed that model on a Kubernetes cluster on the edge. We then used the model to classify packages in a live webcam feed and broadcast the results using a MQTT broker.

2. Data

Preliminary investigation found no publicly available, multi-class package datasets that were labeled for object detection use cases. Rather, we identified several small image datasets that, combined with some images generated by our own team, could be augmented and labeled to meet the requirements of this project.

2.1 Data Sources

This project relies on data from the sources listed below:

- **Roboflow Packages Dataset:** This dataset consists of 26 raw images of packages on doorsteps that have been augmented with the Roboflow toolkit to generate 250 images labeled with a single package class in YOLO format.
- **Abhijeet Bhatikar Repo:** This dataset consists of over 500 images of packages on doorsteps in a variety of positions, angles, and lighting conditions labeled with a single package class in YOLO format.
- **Self-Generated:** Our team supplemented the two preexisting datasets with a number of package images of our own creation in the test environment in which this project will eventually be deployed. Reference the project Github for detailed instructions on how this data was generated.

Fig 1. Left to Right: Roboflow, Abhijeet, Self-Generated

In total, the combination of these three data sources yielded 865 unique images to be used for training the model.

2.2 Annotation

As mentioned previously, the two pre-existing datasets used in this project were originally annotated in YOLOv5 format with a single package class. While this was helpful in calculating a baseline model accuracy up front, our concept is predicated on differentiating multiple classes of packages from one another. Naturally, the images that we generated ourselves had no annotation whatsoever.

Our team annotated 865 original images in order to make the dataset suitable for our specific use case. We used the open source annotation tool Makesense.ai to generate bounding boxes according to the class operationalization described in the table below.

Table 1. Operationalization of Package Class Annotations

Class	Operationalization
`box`	Any rectangular cardboard package that holds its own shape rather than assuming the approximate shape of the object inside.
`plastic_bag`	Any soft plastic wrapping that does not hold its own shape and roughly assumes the shape of its contents
`envelope`	Any flat, rectangular packaging made of thick paper or thin cardboard with a single defined opening that is only designed to transport paper, thin books, or other flat objects.

2.3 Augmentation

The dataset was augmented with a simple python script using the albumentations package in order to make the model more resistant to diverse lighting, motion, and weather conditions. The repository file data-aug.py reads all image files in a training directory, augments them with one of three transformation functions, and saves them to a new directory with a new corresponding annotation file in YOLOv5 format. For this project, we used data augmentation to generate a 10x increase in the number of example images in our training dataset.

Table 2. Pixel-Level Transformations Used by Category

Category	**Transformation**
Lighting	RGBShift RandomBrightnessContrast
Motion	MotionBlur
Weather	RandomFog RandomRain RandomSnow RandomSunFlare

It is valuable to note that, while spatial transformations are both available and useful for increasing model accuracy, challenges associated with generating new bounding boxes lead our team to remove this functionality from consideration. Nonetheless, spatial transformations present a great opportunity to improve the model in the future. Some examples of augmented images are displayed below.

Fig 2. Examples of training images with rain, sun flare, and RGB transforms

3. Model Development and Validation

The next step is model development and validation. We built our baseline model on the unaugmented datasets and then used the predictions to measure the baseline's performance—this would then become what we compared any other machine learning algorithm against. We selected a network architecture by testing performance of default configurations on the unaugmented datasets and thereafter fine-tuned the best performing model to optimize its performance. After configuration selection, we trained our model on the full dataset with augmentation. Then, the performance of our fully trained model is evaluated on a testing images set. If it doesn't perform well, we need to go back to the train and validate steps and conduct hyperparameter optimization or tuning. Otherwise, we capture the accuracy and the model is ready for the real-time package classification development on the NX edge, using a Kubernetes cluster.

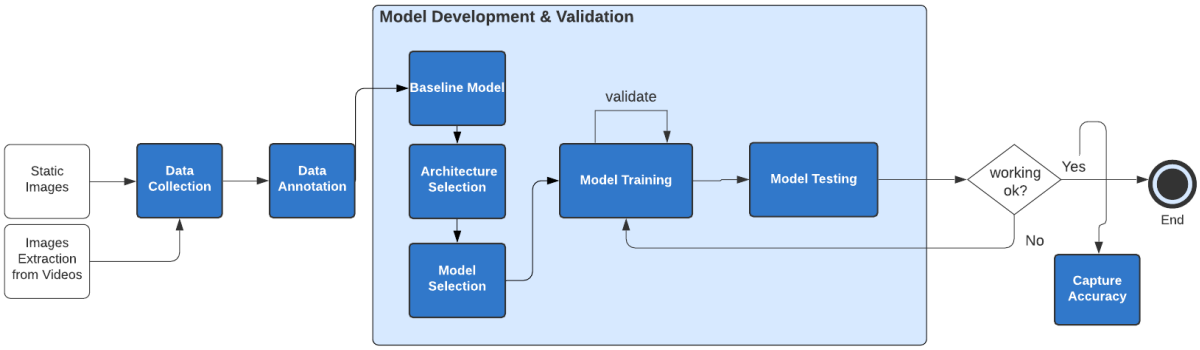

Fig 3. Model development & validation flow diagram

3.1 Architecture Selection

While YOLOv5 is dominant among object detection architectures (as of this writing), we wanted to ensure that there are no particulars of the package detection problem that render an alternate architecture particularly well-suited to this purpose. Broadly speaking, we wanted to test lightweight variants of the Single Shot Detector (SSD), Regions with CNN features (R-CNN), and You Only Look Once (YOLO) approaches. In order to quickly test both the practicality of use and predictive ability of these systems, we adapted default configurations of the MobileNetSSDv2[4], FasterRCNN[5], and YOLOv5[6] network architectures as outlined by Roboflow (see references) and trained them using our unaugmented dataset, which were each able to train on the provided data within 30 minutes. The mean average precision (mAP) metrics associated with each architecture at the 0.5 and 0.5:0.95 IOU levels are outlined in the table below:

Table 3. Comparison of Architectures Trained on Unaugmented Dataset

Architecture	mAP@0.5	mAP@0.5:0.95
MobileNetSSD	0.5214	0.8338
Faster-RCNN	0.5305	0.8391
YOLOv5	0.6059	0.9515

The maximal performance of YOLOv5 across both mAP metrics validated the common understanding of YOLOv5's dominance in object detection, even when applied to the package detection use case. As such, we proceeded in training the model with the YOLOv5 architecture.

3.2 Configuration Selection

The next step was to determine the ideal YOLOv5 configuration for the purpose of package detection. YOLOv5 includes 4 separate network configurations by default, named by their size—yolov5s (small), yolov5m (medium), yolov5l (large), and yolov5x (extra large), increasing in both performance and runtime in that order:[7]

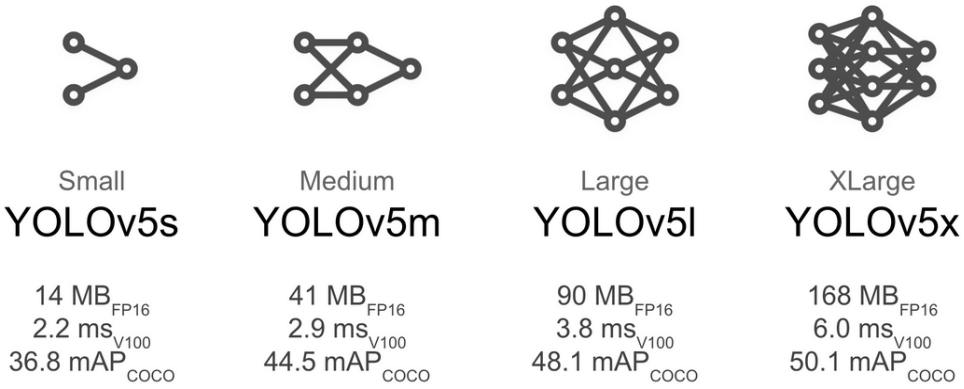

Fig 4. Visual comparison of the YOLOv5 network configurations[7]

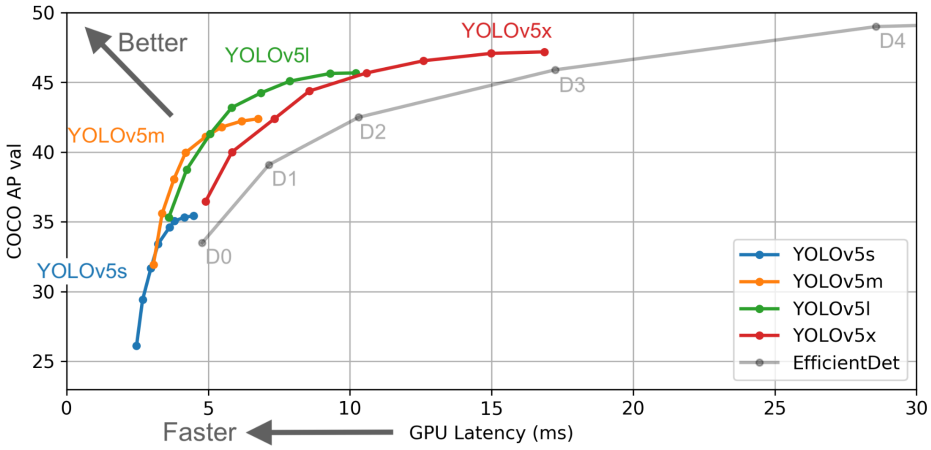

Fig 5. Performance of the 4 YOLOv5 network configurations against EfficientDet[7]

Given the time considerations involved with training the different configurations on the full, augmented dataset, we decided to first train the least and most complex variants for 100 epochs each to determine the increase in mAP associated with the larger network. The outcomes of this training are provided below:

Table 4. Comparison of YOLOv5 Network Configurations Trained on Augmented Dataset

Configuration	Train Time (100 epochs)	mAP@0.5	mAP@0.5:0.95
yolov5s	01:22:17	0.694	0.558
yolov5x	07:20:30	0.688	0.551

The lightest model (yolov5s) unexpectedly performed slightly better than the heaviest model, (yolov5x), despite requiring upwards of 5x more time to train 100 epochs. As such, we decided that it would be best to proceed with yolov5s, given the reduced training time and the expected reduction in inference time as a result.

3.3 Predictions

The predictions provided by the trained yolov5s model were relatively accurate. As evidenced in the images below, the model met several key criteria for success, namely:

- Packages are not detected in images that do not contain them
- Packages positioned differently from those used in training
- Multiple overlapping bounding boxes
- Images taken in darkness
- Packages with colors not present in training data

Fig 6. Bounding boxes detected by trained yolov5s model.

Overall, this proved to be sufficiently capable of detecting packages in different settings, so we used this model in the live detection setting.

4. NX Architecture

For this project, a lightweight containerized application pipeline has been built with components running on the edge, the Jetson NX, and is used for running inference with a YOLOv5 model on images, videos, or streams. The application has been written in a modular/cloud native way so that it could be run on any edge device or hub.

In addition, the edge application has been deployed using Kubernetes (K3s) on the Jetson NX. The application is able to detect packages in a video stream coming from the USB camera. For the **`package detector`** component, the application scans the video frames coming from the connected USB camera for objects. When one or more objects are detected in the frame, the application scans them and publishes them to the broker. The edge application uses "MQTT" as the messaging fabric. Here NX is acting as a hub that has a **`package broker`** installed, and the **`package detector`** sends its messages to this package broker first. There is another local listener component, called **`package logger`** that receives these messages from the local broker, and just outputs to its log (standard out) that it has received a package or multiple packages.

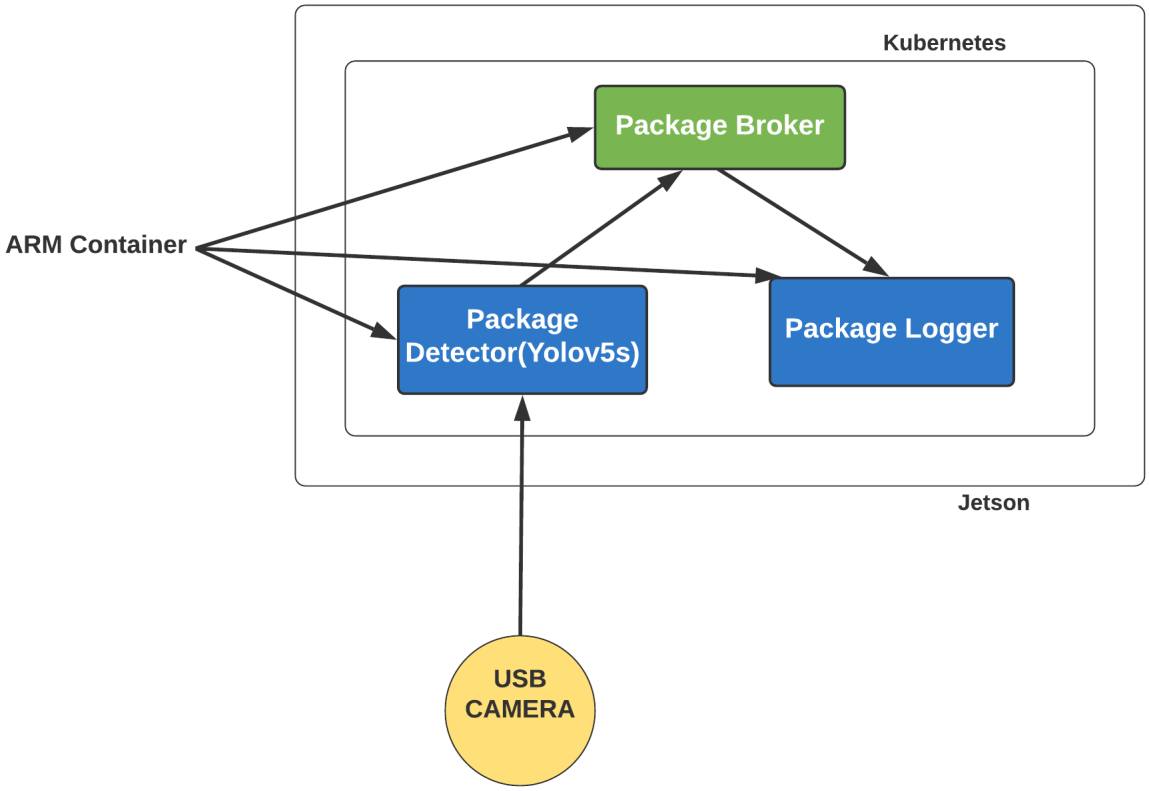

Fig 7. Architecture of Docker containers

As seen in the above architecture, there are 3 containers on NX.

The `**Package Detector**` takes images from the USB camera and detects the object as per the class defined in the data.yaml file. It then publishes the detected images that are listened to by the `**Package Logger**` through the broker topic on `**Package Broker**`.

Package Detector - This container includes:

- yolov5s.pt
- package-detector.py
- Dockerfile
- Package-detector-deployment.yaml

Package Broker - This container includes:

- Dockerfile
- package-broker-deployment.yaml
- package-broker-service.yaml (created a service, to access the broker from outside Kubernetes and then from inside it)

Package Logger - This container includes:

- Dockerfile
- package-logger-deployment.yaml
- package-logger.py

All the steps to replicate NX step is available at: NX Implementation

5. Results:

Real-time prediction on a webcam feed connected to the NX works as expected, with the following output displaying the model detecting a plastic bag, envelope, and box in one frame:

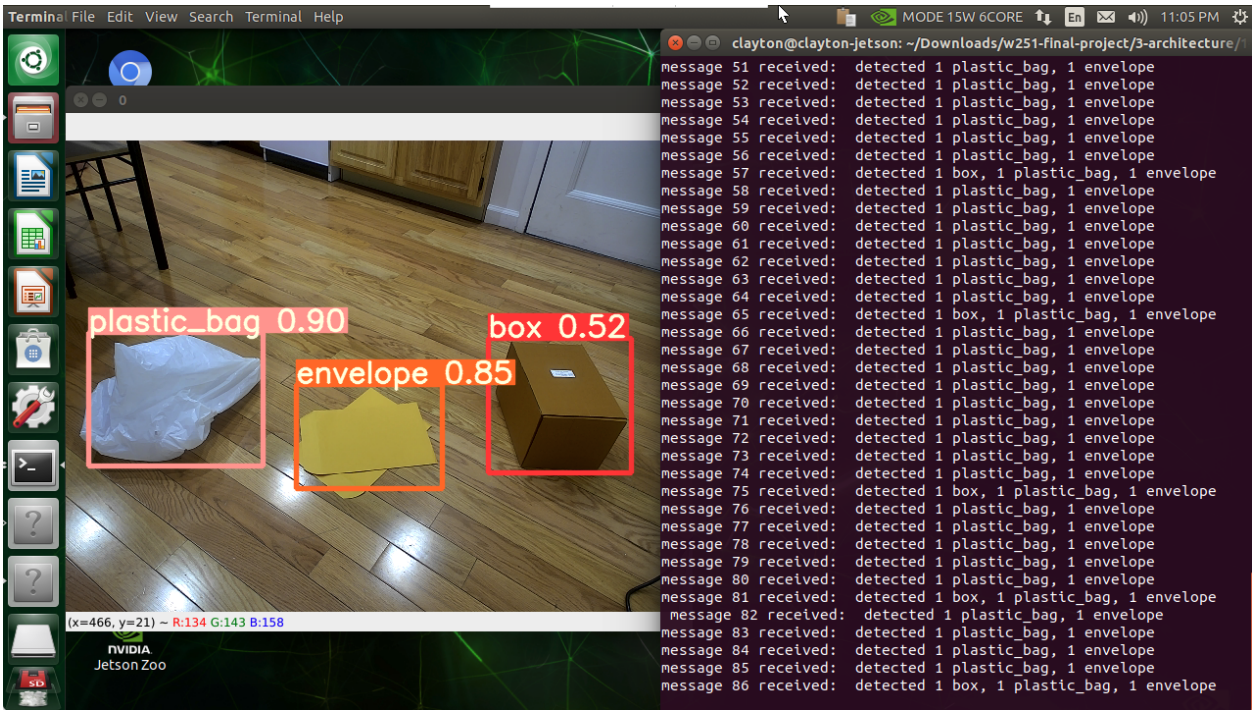

Fig 8. Bounding boxes detected in real-time on the NX using the trained yolov5s model.

6. Conclusion

The automated doorstep package detection system we built that could identify multiple classes of packages in real time with average accuracy above 70%. After comparing with different architectures and configurations, Yolov5s model gave the best performance. We deployed it on NX edge device to make the whole system interactable and applicable.

During the development, we faced a challenge in envelope annotation. There were many envelopes in a stack, which made the annotation really hard unless the envelopes were at angle. Some logo (eg. Macy's star) on the box also led to confusion to be classified as envelopes. The possible solution is to use polygon annotation, which would aleve some of the issues with stacked

envelopes, and likely improve classification over all. For the future enhancements, we can extend our system to porch pirate prevention and delivery/pick up service support. Also, we used pixel level augmentations in this project to simulate bad weather, weird lighting, etc. But spatial augmentations - horizontal flip, warping, etc - could make our model more robust. These are some good trys for further exploration.

7. Works Cited

[1]F. Lardinois: *Hellman & Friedman deal values SimpliSafe at $1B*. Published June 29, 2018. Retrieved from https://techcrunch.com/2018/06/29/hellman-friedman-acquires-controlling-interest-in-simplisafe/.

[2]L. Stevens, D. MacMillan: *Amazon buys smart-doorbell maker Ring in deal valued at more than $1 billion*. Published February 28, 2018. Retrieved from https://www.marketwatch.com/story/amazon-buys-smart-doorbell-maker-ring-in-deal-valued-at-more-than-1-billion-2018-02-27.

[3]R. Laycock, C. Choi: *Porch pirates statistics*. Published July 31, 2021. Retrieved from https://www.finder.com/porch-pirates-statistics.

[4]J. Nelson: *Training a TensorFlow MobileNet Object Detection Model with a Custom Dataset*. Published February 9, 2020. Retrieved from https://blog.roboflow.com/training-a-tensorflow-object-detection-model-with-a-custom-dataset/.

[5]J. Nelson: *Training a TensorFlow Faster R-CNN Object Detection Model on Your Own Dataset*. Published March 11, 2020. Retrieved from https://blog.roboflow.com/training-a-tensorflow-faster-r-cnn-object-detection-model-on-your-own-dataset/.

[6]J. Nelson, J. Solawetz: *How to Train YOLOv5 On a Custom Dataset*. Published June 10, 2020. Retrieved from https://blog.roboflow.com/how-to-train-yolov5-on-a-custom-dataset/.

[7]G. Jocher: *yolov5*. Published August 1, 2021. Retrieved from https://github.com/ultralytics/yolov5.