# Preliminary Results

Lauren Spee

**Problem statement**:
Home security application that uses video input from a camera and alerts the user of any suspicious objects detected in the footage. Suspicious objects will be part of a predetermined list (animals, persons, cars). Though due to hardware limitations, I will be displaying my model on stock security footage from youtube.

**Data Preprocessing:**
I decided to use the Pascal VOC dataset, specifically the sets from 2007 and 2012, as the COCO dataset was too massive and difficult to work with with limited computing power. VOC is also compatible with YOLOv5 and is significantly smaller. YOLOv5 includes preprocessing, which resizes the images (to 416 in my case for speed of training), normalizes them, and augments them.

**Machine learning model:**
I changed my model to YOLOv5 as it's lightweight and easier to train - thus better for beginners, which I am, and better for limited computing power.

YOLOv5 requires downloading of torch, matplotlib, numpy, opencv, pandas, scipy, and various other libraries that it depends on to train.
YOLOv5 has 270 layers (majority of which are convolutional or c3 layers) with a (416, 416, 3) input vector representing the image size I chose and the 3 colour channels. As with all single-layer object detection algorithms, it has a backbone, a pre-trained network which is used to extract rich feature representation from images, followed by a neck that extracts feature pyramids, and then finally a head which performs the final operations, including adding anchor boxes to the feature maps and final rendering.

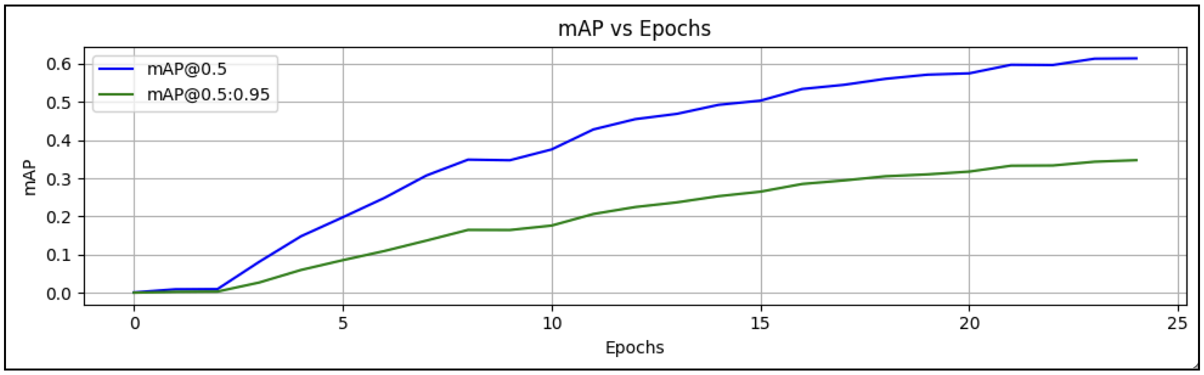
*Splits*:
With the combined VOC 2007 and 2012 data I used, I implemented a roughly 70% training-30% validation split. This provides enough data to train the model, while keeping enough aside to estimate its performance, tune hyperparameters, and prevent overfitting.
*Regularization*:
YOLOv5 implements some regularization techniques, including data augmentation (creating variations in the training data, improves robustness), label smoothing (reduces confidence, improves generalization), batch normalization (normalizes the inputs of each layer to have a mean of 0 and variance of 1, adds noise in the activations), weight decay (penalizes the model by adding the sum of the squared weights to the loss function, prevents overfitting).
*Hyperparameters/Optimization*:
When training, I downsize the images to 416 in order to speed up training time, this is also a reasonable decision since security cameras generally don't record very high-quality video, so my model doesn't need

to train on high-quality images. Additionally, I used a batch size of 64 and cached the images in order to speed up training time.
My model has currently been trained on 25 epochs, which was the balance I found between letting my computer run forever and getting metrics worth extrapolating over.

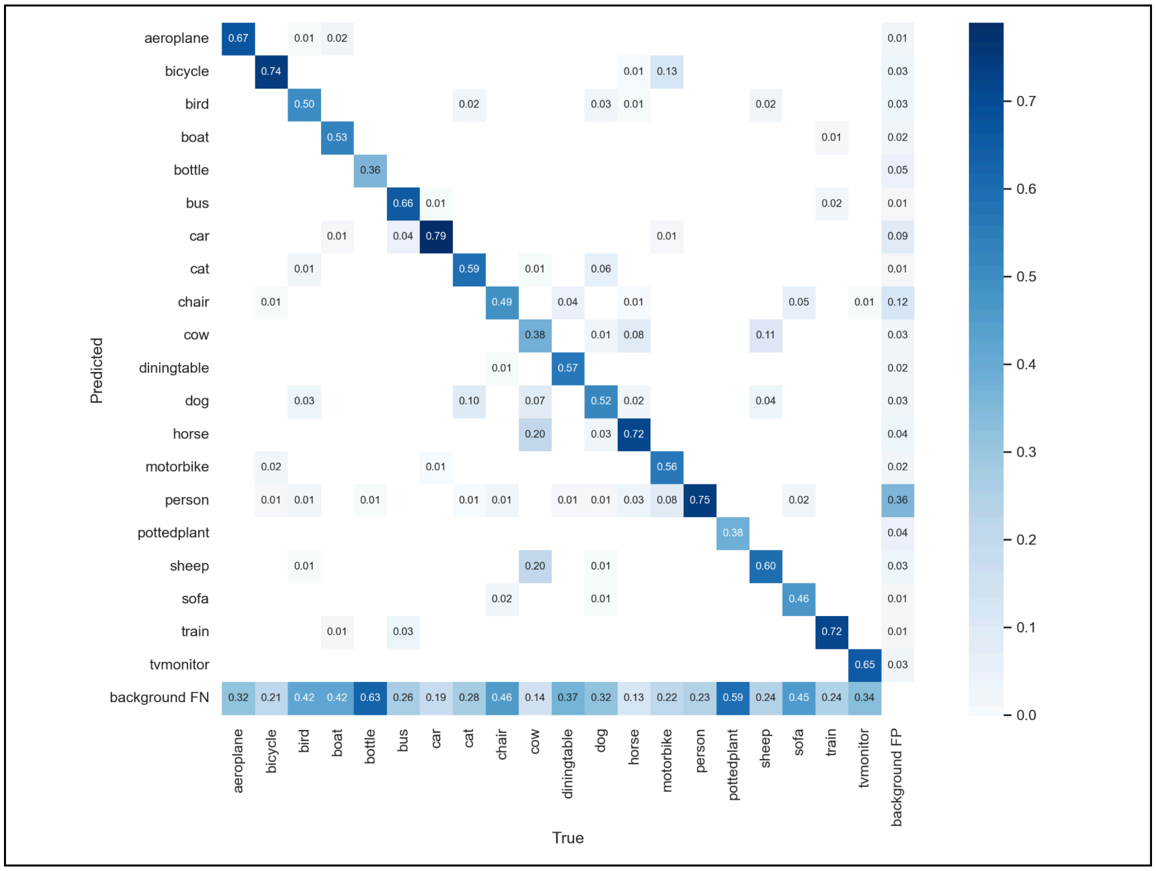
*Validation methods:*
The metrics that YOLOv5 uses include Mean Average Precision (mAP), which measures the average precision across different recall thresholds, precision, which is the ratio of true positives to total positives, and recall, which is the ratio of true positives to total true elements - the latter of which are also displayed in a confusion matrix. After each epoch in the training cycle, the model is assessed, then at the end I ran a validation script to test it on the validation set.

*Challenges:*
My biggest challenge was dealing with the data. Downloading it took forever, and I had to find something compatible with the model as I didn't have the time or the manpower to edit the annotation format for an entire dataset. The second-biggest challenge is the training time. Google Colab only lets the user use the GPU for a certain amount of time and would unpredictably end its runtime, deleting anything saved in the process. So I managed to get CUDA and cuDNN on my laptop and was able to train my model there using my Nvidia GPU, but it still took a very long time.

**Preliminary results**:



In the training data, it's seen that with each epoch, the Mean Average Precision improves, which means that the model is learning. Normally you'd see a plateau followed by a decrease which would show overfitting, but that does not occur here, most likely due to the limited amount of epochs which served as an "early stopping" strategy. It's also good news that since the value improves with each epoch, underfitting is not happening either. However, the mAP values are not high enough, which indicates I should train my data on more epochs, and if possible, more data.



After the training, a validation run on the validation data was run, which resulted in this confusion matrix. As shown in the matrix, the diagonal cells, which represent the correct prediction was made, are significantly more saturated than any of the other cells, which is good. Some of the values, however, barely exceed 30%, and there is some scattering of visibly saturated cells that are not on the diagonal. Considering I only trained the data over 25 epochs on an arguably more limited dataset, improvement would probably come with more training and a larger dataset.

Overall, I'm satisfied with how the model performed given the circumstances, but I plan on more, better training before deploying it.

**Next steps**:
If I'm lucky, I won't have to alter my model. I didn't strongly cater my training for my specific task, which might become a problem in the future, but seeing as I'm looking for an object detection model, I'm hoping it's not too far off. For my next steps, I will be implementing the functionality of detecting objects in security camera footage, so the results from that will tell me if I have to go back to the drawing board in terms of model training.