Final Report

Road Sign Detection

Shane Lafollette

PROBLEM:

Automation is advancing in our daily lives. We have apps and tools to take care of tasks that we can hand off to technology. It is also advancing in every job in the economy. It helps speed up development, manufacturing, quality assessment, and helps free up time for people to be productive in other areas.

Several new automations in our lives have come through the use of computer vision and using algorithms to help computers learn to decipher their surroundings. Much of the new technology on new vehicles use computer vision that is trained to interpret the surroundings they view. From this information, the computer will decide and perform actions to aid the driver or will alert the driver to what it may interpret. To take this technology further, several companies have created self-driving cars which take the human completely out of the equation. These vehicles are equipped with several cameras that interpret the vehicles' surroundings just as a human would, viewing the road, traffic, median, edge lines, signs, crosswalks, pedestrians, traffic lights, and hazards.

One of the items that this tech must decipher is road signs and traffic lights. When driving, the vehicle needs to interpret signs that identify stops, speed limits, crosswalks, merging lanes, traffic lights, school zones and more. Not only does the vehicle need to do this, but it needs to process this information accurately and quickly to keep up with the ever changing situations on the road.

The goal of our project is to develop a neural network that processes images, identifies traffic sign objects, and classifies the object into the correct class while also predicting a bounding box that encompasses the object detected. While our model may not be cutting edge, it will give us an idea of how these models work and how good a base model is. We will try several models, seeing what gives us the best accuracy with our data. Some models used will be model structures that have already been tried and tested, known to deliver great results, and will just need to be trained on the dataset.

Data:

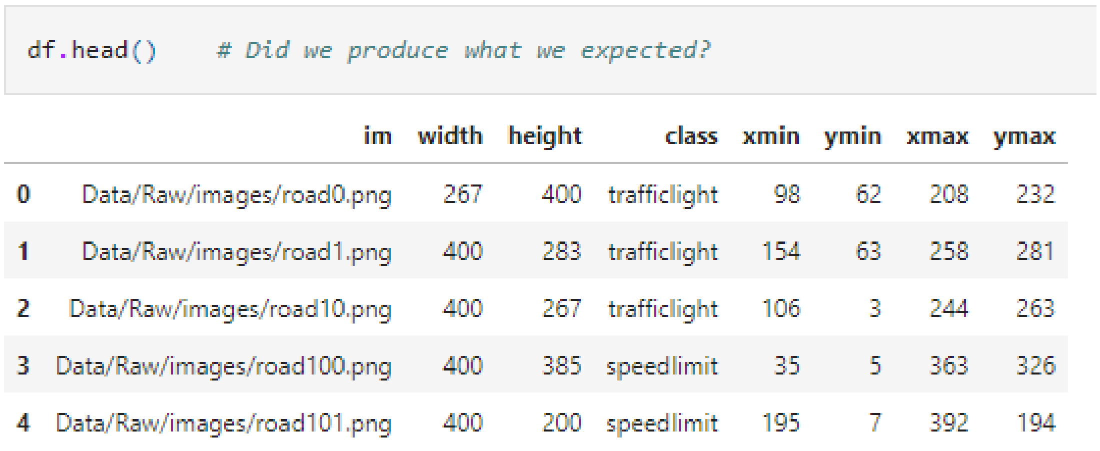
The dataset being used, we have acquired from Kaggle.

Link to data: https://www.kaggle.com/datasets/andrewmvd/road-sign-detection

This Dataset is 877 images of road signs and traffic lights. There are four classes that we will be classifying these images as: Speed limit, Cross walk, Stop, and Traffic light. These images show the signs and lights from different angles, distances, and in different lighting, just as they would be seen while driving. This should help to train the model to be accurate in varying conditions. The raw images are color images in a RGBA format. They vary in height and width, which may have to be transformed for the model used. This data has already been labeled and has bounding boxes around the objects we would like our model to detect and predict. This should save us the time and steps of having to label and create the bounding boxes ourselves. The label and bounding box data is saved in an .xml file that identifies which image it belongs to as well as class label, and coordinates for the true bounding boxes.

EDA:

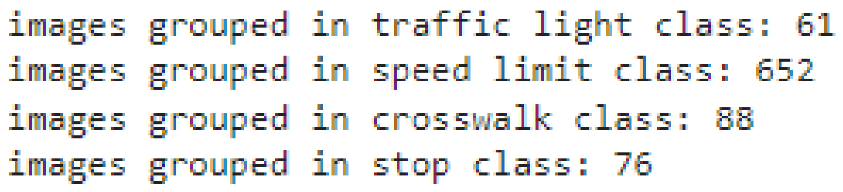
We read the image paths and xml files into a pandas dataframe to get a look at the data and make find any observable issues that we may see. After reading the information into the dataframe, we have 877 rows, each row being a picture and its feature descriptions, and eight columns, consisting of image path, image height, image width, image class, and the minimum and maximum coordinates of the bounding box in the image.



```
df.head()     # Did we produce what we expected?
```

	im	width	height	class	xmin	ymin	xmax	ymax
0	Data/Raw/images/road0.png	267	400	trafficlight	98	62	208	232
1	Data/Raw/images/road1.png	400	283	trafficlight	154	63	258	281
2	Data/Raw/images/road10.png	400	267	trafficlight	106	3	244	263
3	Data/Raw/images/road100.png	400	385	speedlimit	35	5	363	326
4	Data/Raw/images/road101.png	400	200	speedlimit	195	7	392	194

For training we would like several images of each class to make sure we have good training for the model. After getting this count, it is obvious that we have an uneven number of class samples. The speed limit class makes up the majority of the data with 652 images, or 74% of the data. The other three classes are closer in sample size, with 88 cross walk images, 76 stop images, and 61 traffic light images.

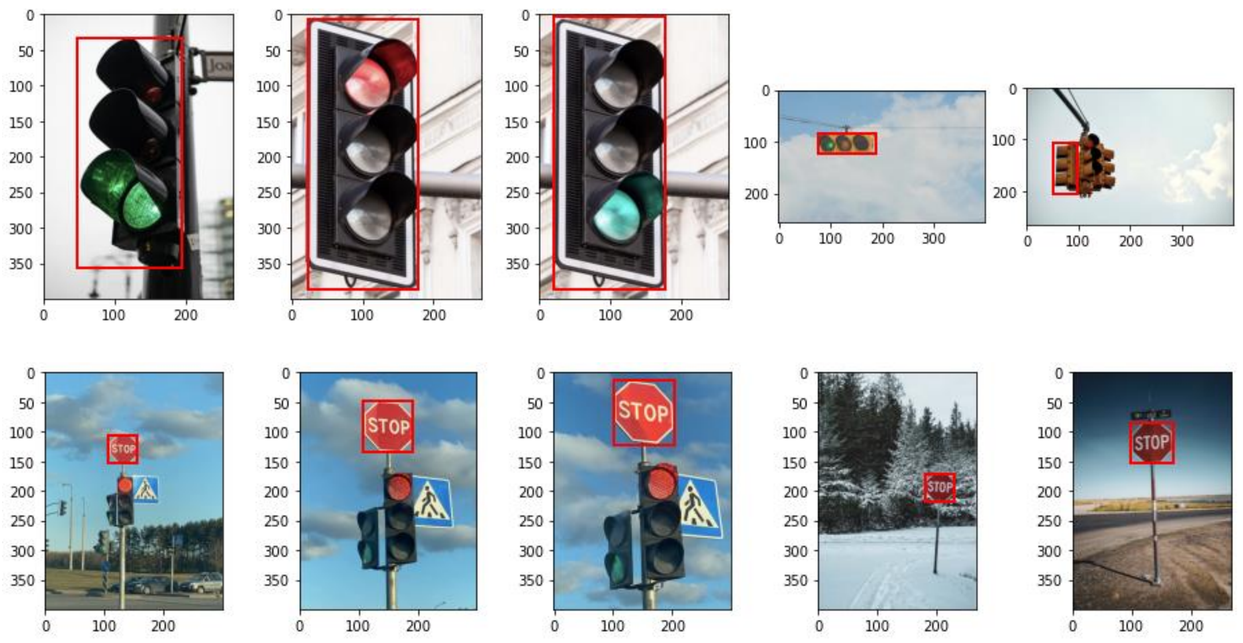


```
images grouped in traffic light class: 61
images grouped in speed limit class: 652
images grouped in crosswalk class: 88
images grouped in stop class: 76
```

While having a dataset with equal classes may seem like the best idea, I think we also must consider that the speed limit class will be the class that actually has the most changes from sign to sign. As the stop and cross walk signs should generally be the same likeness from image to image, the numbers on the speed limit signs will vary. The traffic light may also have some degree a variation depending on which signal is being given by traffic light. These differences may be used by the model or may not, to what extent will vary by the training of the neural network. My initial thought is that the model will use the silhouette of the object by edge detection to make most of its predictions, with little weight being given to the pixels within the object. This is due to the fact that each of the classes we are identifying have distinct shapes. If we wanted to take our model a step further in the future and train it to be able to identify the actual speed limit for the area and make the distinction between a red light and green light, these pixels within the detected object would become much more important for our model. Let’s look at several of our images to see some examples of the shapes and colors.

As can be seen through just a couple images, the objects being detected are in various locations with some having little background noise (few objects and hard lines in the image) and others being images taken from city streets with dozens of objects in the image. The images also have a great range of object size and shape with the varying distances and angles.

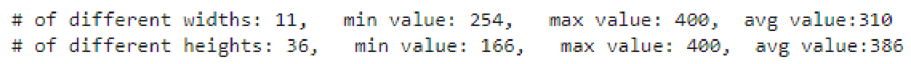
We would also like to visualize the true bounding boxes to make sure that the correct data was pulled from the .xml files and correctly labeled within our dataframe.



It looks like our bounding boxes seem to be in the correct areas. The bounding box should draw a line around the object as precise as possible while encompassing the whole object. A couple things that stick out when looking at some of the images and the bounding boxes.

1. Several pictures show multiple classes in the image but only have bounding boxes around the labeled class. This will affect the model accuracy as it may predict the label and bounding box for the wrong object in the image.
2. With images having multiple objects of multiple labels, as in the real world, the model should be able to detect and predict multiple objects with multiple bounding boxes in the image.

It has also been noticed that the images are of varying sizes. This may have to be changed and all the images resized to be input into the model. Some resizing we may do before pushing data to model, while in other instances, you may want to add padding to images to fit a certain input size. Lets take a look at the size values and hopefully gain some insight into what size we may transform the images into and how those changes may affect the image in regards to aspect ratio and distortion.



```
# of different widths: 11,   min value: 254,   max value: 400,  avg value:310
# of different heights: 36,   min value: 166,   max value: 400,  avg value:386
```

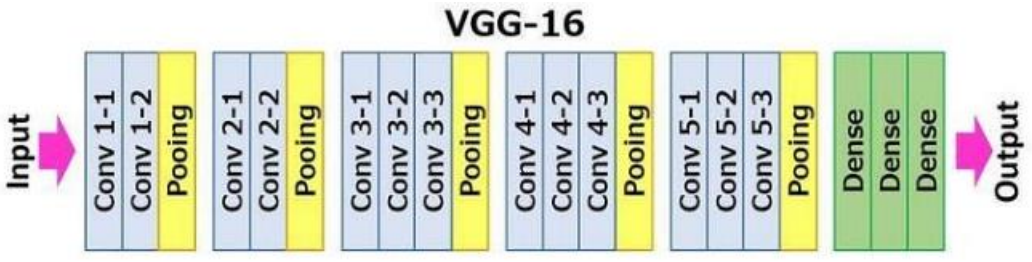
The width of the images has fewer unique values with an average value of 310 pixels. The height of images has a greater number of unique values, but most seem to be nearer to the max size of 400 pixels due to the average height being 386 pixels. If we resize the images, they may need to be looked over to see how this affects them.

Modeling:

VGG-16 Model

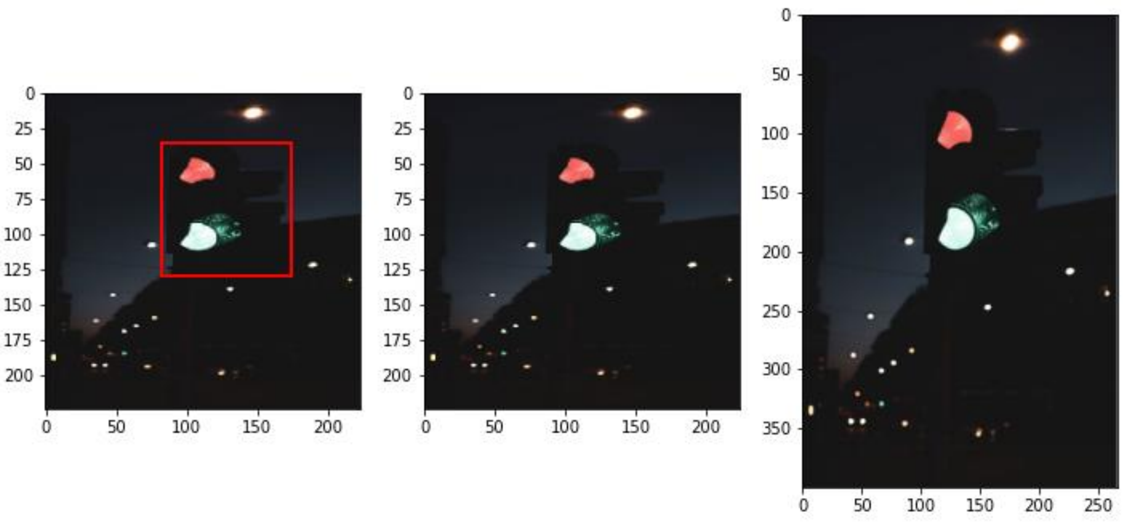
For image classification, a neural network is the preferred machine learning model. These models are given an input of the image as an array of pixel values. The model will then pass this image through several “layers” that will apply any number of filters (calculations of different weights) on the data to create a feature map of the image. These feature maps are then passed to the next layer. These layers will use activation functions that help the model find the important features. After each batch, these filters are adjusted to make more accurate predictions. There are also pooling and Dropout layers that help to keep the number of parameters from growing too large and helps to keep models from overfitting the training data. There are tons of changes that can be made to the model, the training, and evaluation parameters that will affect your model accuracy.

The dataset being used is relatively small compared to all the possible images and variations of signs and traffic lights it may see. With a percentage used for testing, only about 750 images will be available for training. With all the changes that can be made on these models, I decided to start with a CNN (Convolutional Neural Network) architecture that had already been created and tested with good accuracy. The VGG-16 CNN model is a common model used for image classification. It has 13 convolutional layers with Max Pooling layers after 2 or 3 of the convoluted layers. The last 3 layers are dense and can be dropped when importing the model from TensorFlow. These layers can be replaced by your own custom layers to better fit your own data.



When importing this model from TensorFlow, it can also come pre-trained on ImageNet data which includes over 14 million images and 1000 classes. Using this pre-trained model is known as transfer learning and can help when our own dataset is small. These pre-trained layers help speed up the learning of the model due to the weights starting out near the optimal point. It is possible to update the training on any number of these layers or freeze them as they come.

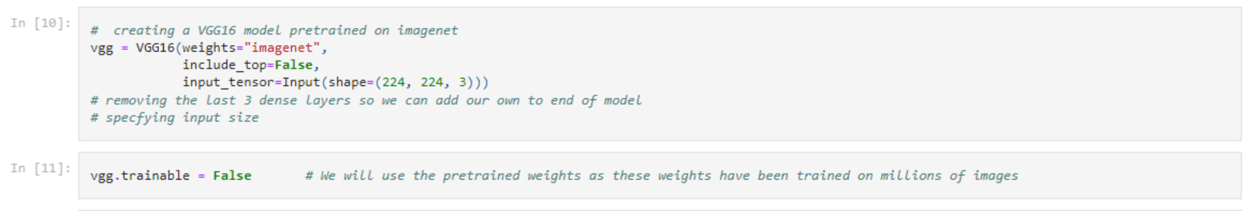
The VGG-16 was pre-trained on images of pixel size 224 x 224 x 3 and expects images of this size as the input. After importing the dataframe we created during EDA, the images and bounding boxes were resized to take a look and see if there was too much distortion for the model to produce good accuracy.



The images above show one image that has been resized from its original size on the right. While the image does have distortion, it was thought the pretrained weights would make up for this with better accuracy.

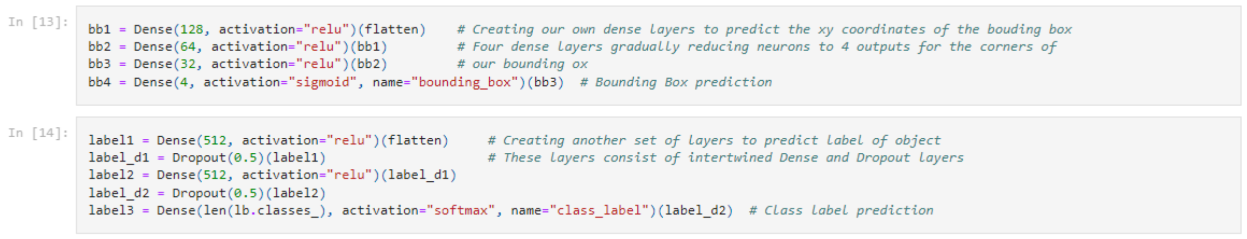
After the resizing of images and bounding boxes, the pixel data, that ranges between 0 (black) and 255 (white), was normalized to be between 0 and 1 to help minimize the computation complexity. The normalized image data, label data, and bounding box coordinates were all converted to a NumPy array. The data was then split into training, validation, and testing sets. 15% of all data was used for testing which calculated to 132 images. This left 745 images for training and validation.

I then implemented the VGG-16 model using the ImageNet weights, dropping the dense layers so that we can add our own. We also froze the VGG-16 model layers so no custom training will happen for these initial layers.



```
In [10]:  #  creating a VGG16 model pretrained on imagenet
          vgg = VGG16(weights="imagenet",
                      include_top=False,
                      input_tensor=Input(shape=(224, 224, 3)))
          # removing the last 3 dense layers so we can add our own to end of model
          # specfying input size

In [11]:  vgg.trainable = False       # We will use the pretrained weights as these weights have been trained on millions of images
```

These pre-trained layers will output feature maps that we will flatten and pass to custom created dense layers along with some dropout layers to predict the class label and predict the bounding box location.

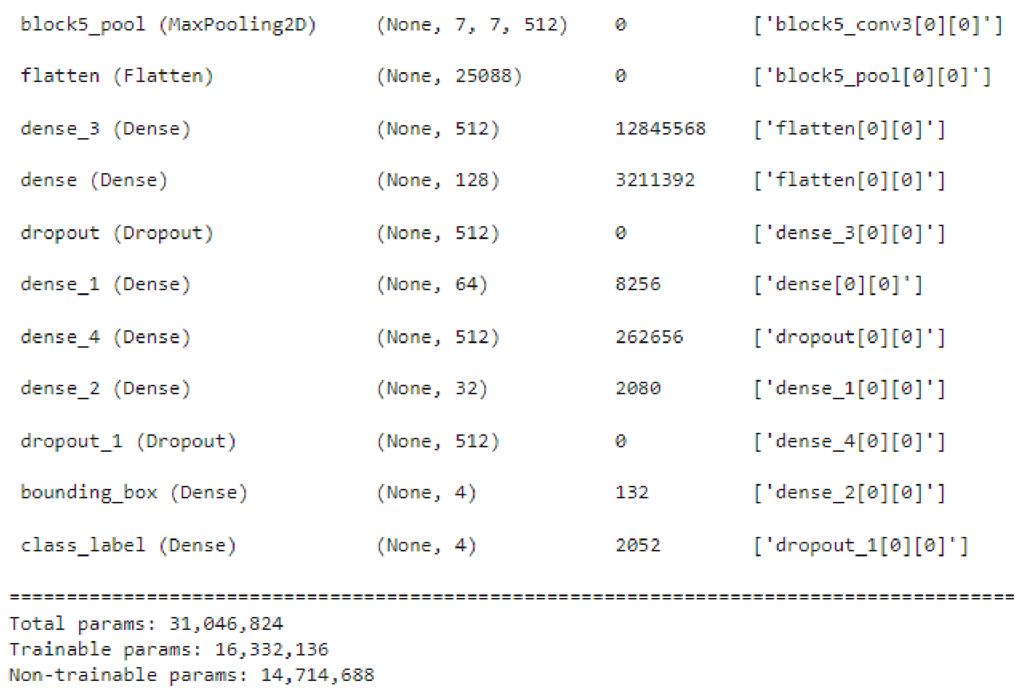


```
In [13]:  bb1 = Dense(128, activation="relu")(flatten)    # Creating our own dense layers to predict the xy coordinates of the bouding box
          bb2 = Dense(64, activation="relu")(bb1)         # Four dense layers gradually reducing neurons to 4 outputs for the corners of
          bb3 = Dense(32, activation="relu")(bb2)         # our bounding ox
          bb4 = Dense(4, activation="sigmoid", name="bounding_box")(bb3)  # Bounding Box prediction

In [14]:  label1 = Dense(512, activation="relu")(flatten)     # Creating another set of layers to predict label of object
          label_d1 = Dropout(0.5)(label1)                    # These layers consist of intertwined Dense and Dropout layers
          label2 = Dense(512, activation="relu")(label_d1)
          label_d2 = Dropout(0.5)(label2)
          label3 = Dense(len(lb.classes_), activation="softmax", name="class_label")(label_d2)  # Class label prediction
```

I created four dense layers to learn and predict the XY coordinates of the bounding box. The initial VGG model output a flattened array of 25,088 parameters that we will feed to the first bounding box dense layer that will have 128 filters. This will bring the number of parameters to over 3 million. Values will be narrowed down when going through the other three dense layers to 132 parameters that the model will use to make a prediction on the bounding box. The same process will happen to predict our class label but the data will run through other dense layers that will contain 4 times as many parameters after the first dense layer. Dropout layers have been added between the dense class label layers that will dropout (ignore random parameters) half the parameters so to not overfit. The model will be using categorical cross entropy for label accuracy and mean squared error for bounding box location accuracy.

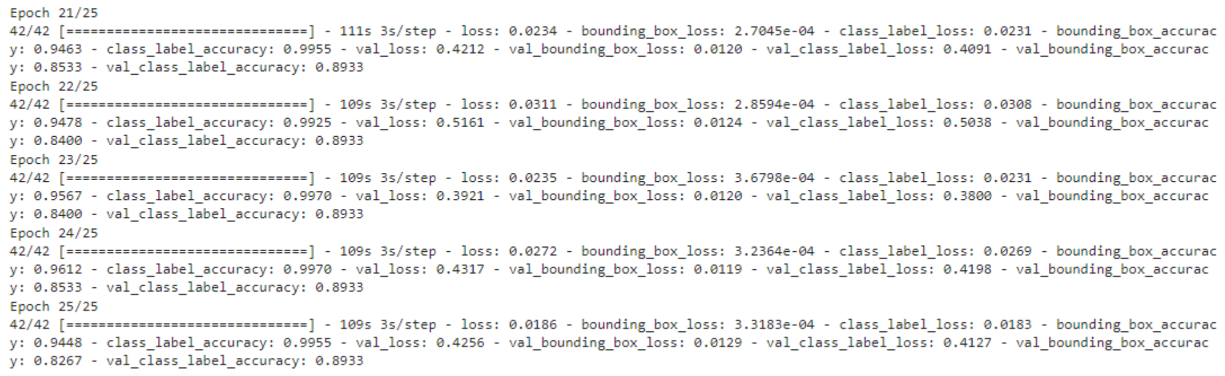
Before compiling model, an optimizer, learning rate, batch size and number of epochs must be decided upon. We will use the ADAM optimizer and a learning rate of 0.001 to start with. Batch size is the number of images trained on before updating the model weights. The batch size will be set to 16. This mean there will be 42 batches for every epoch. Epochs are the number of times that the model will go over the data for training. We will set our epochs at 50 and see what the accuracy and loss metrics look like. After initializing the model, we can get a model summary with a list of layers, layer shape, and number of params it contains.

```
Model: "model"
________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
========================================================================================
 input_1 (InputLayer)           [(None, 224, 224, 3  0           []
                                )]

 block1_conv1 (Conv2D)          (None, 224, 224, 64  1792        ['input_1[0][0]']
                                )

 block1_conv2 (Conv2D)          (None, 224, 224, 64  36928       ['block1_conv1[0][0]']
                                )

 block1_pool (MaxPooling2D)     (None, 112, 112, 64  0           ['block1_conv2[0][0]']
                                )

 block2_conv1 (Conv2D)          (None, 112, 112, 12  73856       ['block1_pool[0][0]']
                                8)

 block2_conv2 (Conv2D)          (None, 112, 112, 12  147584      ['block2_conv1[0][0]']
                                8)

 block2_pool (MaxPooling2D)     (None, 56, 56, 128)  0           ['block2_conv2[0][0]']

 block3_conv1 (Conv2D)          (None, 56, 56, 256)  295168      ['block2_pool[0][0]']

 block3_conv2 (Conv2D)          (None, 56, 56, 256)  590080      ['block3_conv1[0][0]']

 block3_conv3 (Conv2D)          (None, 56, 56, 256)  590080      ['block3_conv2[0][0]']

 block3_pool (MaxPooling2D)     (None, 28, 28, 256)  0           ['block3_conv3[0][0]']

 block4_conv1 (Conv2D)          (None, 28, 28, 512)  1180160     ['block3_pool[0][0]']

 block4_conv2 (Conv2D)          (None, 28, 28, 512)  2359808     ['block4_conv1[0][0]']

 block4_conv3 (Conv2D)          (None, 28, 28, 512)  2359808     ['block4_conv2[0][0]']

 block4_pool (MaxPooling2D)     (None, 14, 14, 512)  0           ['block4_conv3[0][0]']

 block5_conv1 (Conv2D)          (None, 14, 14, 512)  2359808     ['block4_pool[0][0]']

 block5_conv2 (Conv2D)          (None, 14, 14, 512)  2359808     ['block5_conv1[0][0]']

 block5_conv3 (Conv2D)          (None, 14, 14, 512)  2359808     ['block5_conv2[0][0]']
```



```
 block5_pool (MaxPooling2D)     (None, 7, 7, 512)    0           ['block5_conv3[0][0]']

 flatten (Flatten)              (None, 25088)        0           ['block5_pool[0][0]']

 dense_3 (Dense)                (None, 512)          12845568    ['flatten[0][0]']

 dense (Dense)                  (None, 128)          3211392     ['flatten[0][0]']

 dropout (Dropout)              (None, 512)          0           ['dense_3[0][0]']

 dense_1 (Dense)                (None, 64)           8256        ['dense[0][0]']

 dense_4 (Dense)                (None, 512)          262656      ['dropout[0][0]']

 dense_2 (Dense)                (None, 32)           2080        ['dense_1[0][0]']

 dropout_1 (Dropout)            (None, 512)          0           ['dense_4[0][0]']

 bounding_box (Dense)           (None, 4)            132         ['dense_2[0][0]']

 class_label (Dense)            (None, 4)            2052        ['dropout_1[0][0]']

==================================================================================================
Total params: 31,046,824
Trainable params: 16,332,136
Non-trainable params: 14,714,688
```

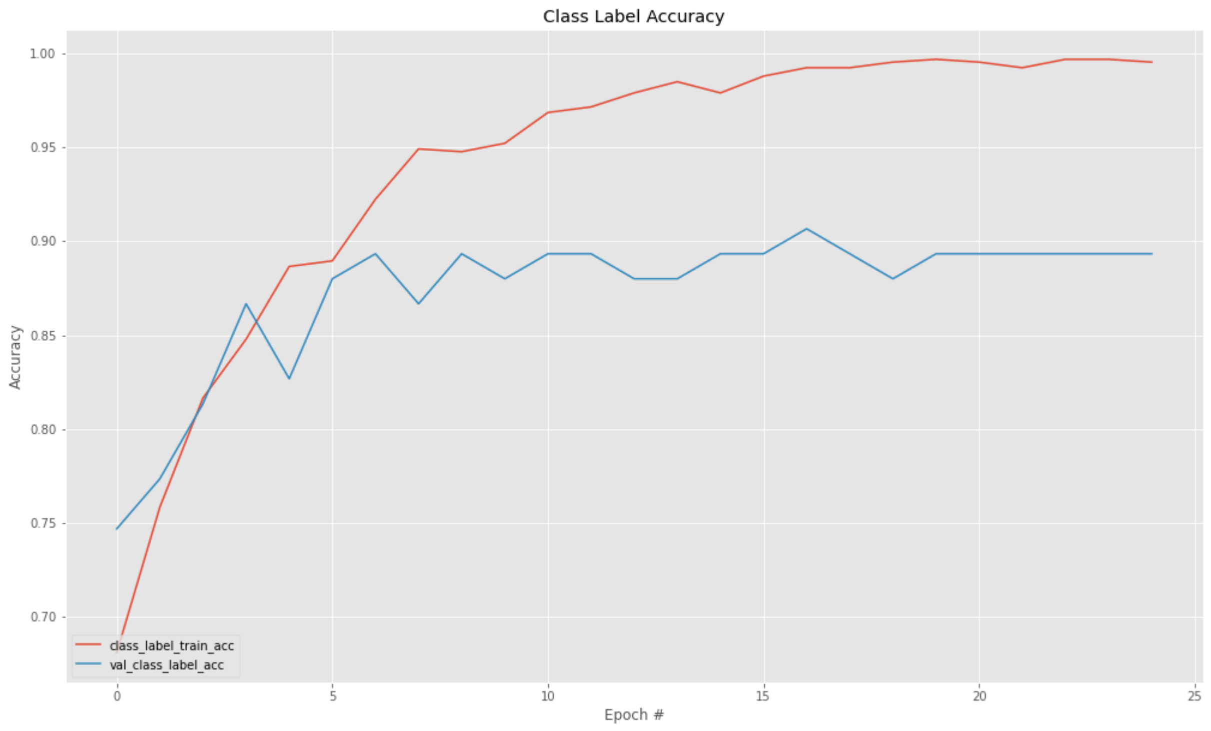
As shown above, the model has over 31 million values from each image as it creates feature maps and uses these to find the important features that help with good prediction accuracy. The class_label and bounding_box layers will output the model prediction for each image. The next step is to train the model and see how it performs on the training data.

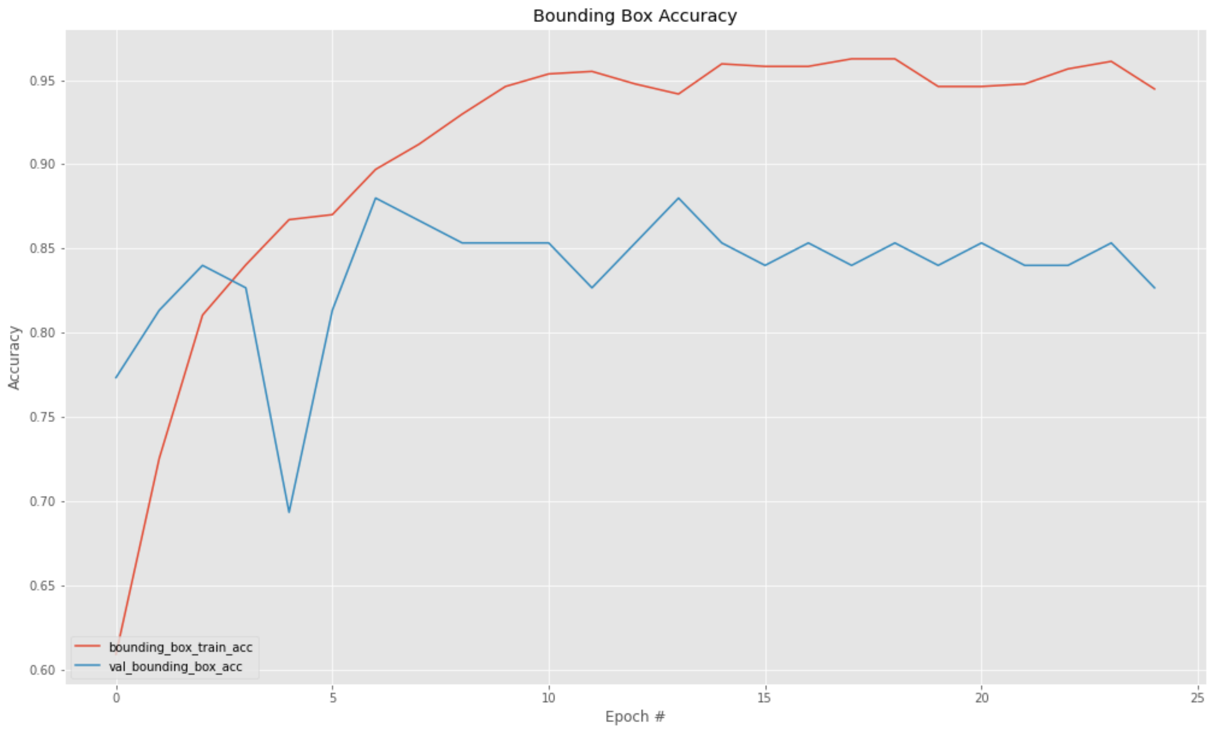


```
Epoch 21/25
42/42 [==============================] - 111s 3s/step - loss: 0.0234 - bounding_box_loss: 2.7045e-04 - class_label_loss: 0.0231 - bounding_box_accurac
y: 0.9463 - class_label_accuracy: 0.9955 - val_loss: 0.4212 - val_bounding_box_loss: 0.0120 - val_class_label_loss: 0.4091 - val_bounding_box_accurac
y: 0.8533 - val_class_label_accuracy: 0.8933
Epoch 22/25
42/42 [==============================] - 109s 3s/step - loss: 0.0311 - bounding_box_loss: 2.8594e-04 - class_label_loss: 0.0308 - bounding_box_accurac
y: 0.9478 - class_label_accuracy: 0.9925 - val_loss: 0.5161 - val_bounding_box_loss: 0.0124 - val_class_label_loss: 0.5038 - val_bounding_box_accurac
y: 0.8400 - val_class_label_accuracy: 0.8933
Epoch 23/25
42/42 [==============================] - 109s 3s/step - loss: 0.0235 - bounding_box_loss: 3.6798e-04 - class_label_loss: 0.0231 - bounding_box_accurac
y: 0.9567 - class_label_accuracy: 0.9970 - val_loss: 0.3921 - val_bounding_box_loss: 0.0120 - val_class_label_loss: 0.3800 - val_bounding_box_accurac
y: 0.8400 - val_class_label_accuracy: 0.8933
Epoch 24/25
42/42 [==============================] - 109s 3s/step - loss: 0.0272 - bounding_box_loss: 3.2364e-04 - class_label_loss: 0.0269 - bounding_box_accurac
y: 0.9612 - class_label_accuracy: 0.9970 - val_loss: 0.4317 - val_bounding_box_loss: 0.0119 - val_class_label_loss: 0.4198 - val_bounding_box_accurac
y: 0.8533 - val_class_label_accuracy: 0.8933
Epoch 25/25
42/42 [==============================] - 109s 3s/step - loss: 0.0186 - bounding_box_loss: 3.3183e-04 - class_label_loss: 0.0183 - bounding_box_accurac
y: 0.9448 - class_label_accuracy: 0.9955 - val_loss: 0.4256 - val_bounding_box_loss: 0.0129 - val_class_label_loss: 0.4127 - val_bounding_box_accurac
y: 0.8267 - val_class_label_accuracy: 0.8933
```

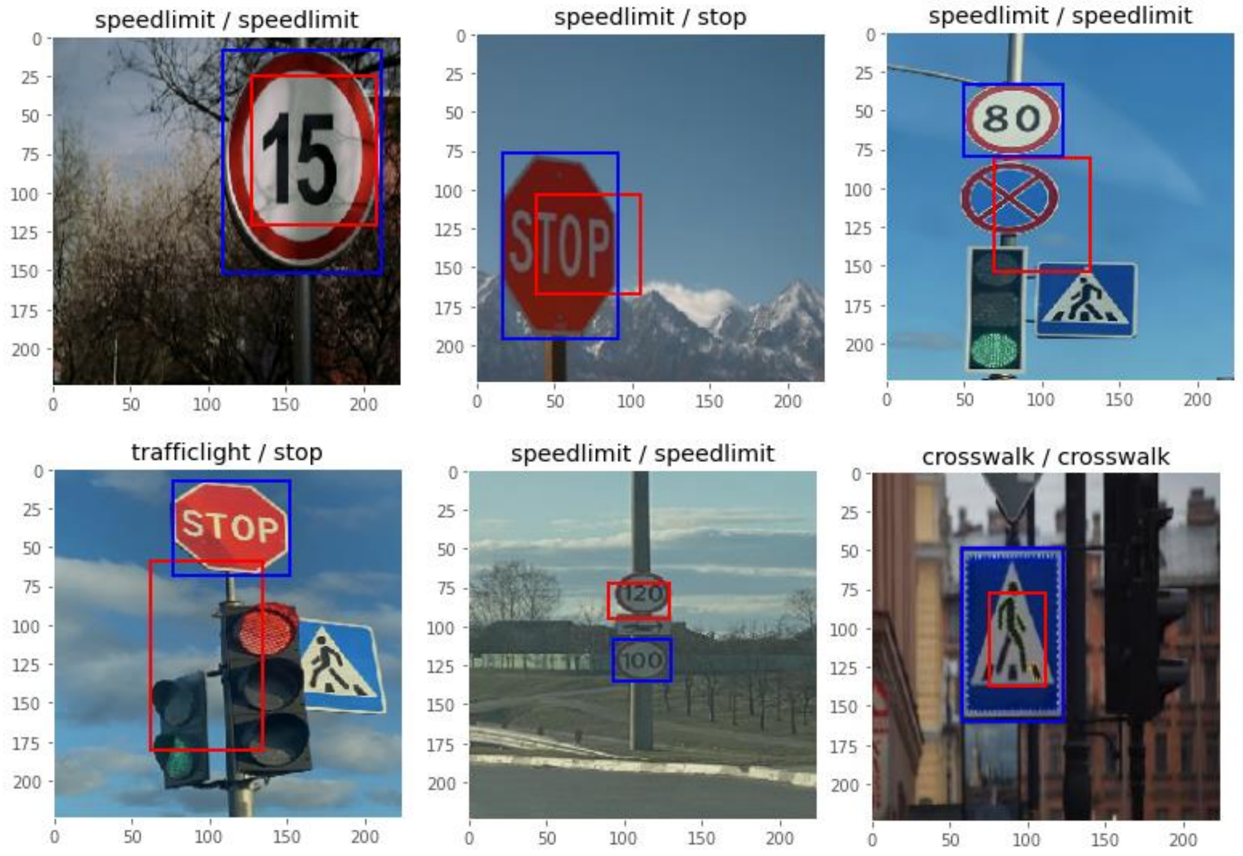
The figure above shows the last 5 epochs and their metrics on the data. After running this model several times, the learning rate was adjusted between 0.1 and 0.00001 to find a good rate where the model would not have too much or too little momentum and get stuck at a local minima or jump over the global minima of the data. The batch size was also changed between 12 and 32 to see how it would affect the metrics. After running, the best metrics after training was attained using a model with a learning rate of 0.0001 and batch size of 16. After 25 epochs, there was not a significant change in training accuracy and the validation accuracy started to drop, potentially giving a clue that training was overfitting the data.

The plots below shows the accuracy we acquired in the model on the training and validation set. The class label accuracy hovered a bit below 90% after 10 epochs on our validation data, while the bounding box accuracy stayed around 85% after 15 epochs. The model was able to find a good fit to the training data, matching labels and boxes at over 95%.





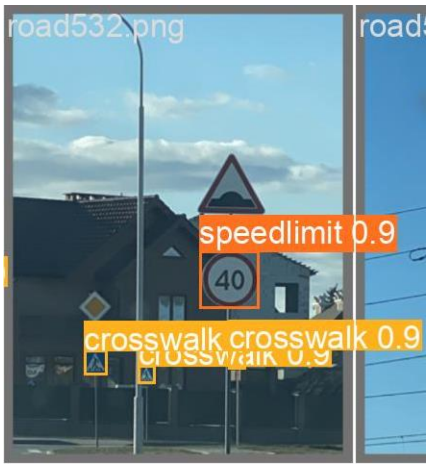
After training, the model is ready to predict on our testing data. Of the 132 testing images, the model predicted 112 correctly or 83% accuracy. A little lower than the validation stats but not far off considering small dataset being tested. The predicted bounding boxes and labels are plotted on some of the testing images below. The predicted box is red, while the ground truth is blue and the predicted label is first label in the title with the ground truth to the right of that.

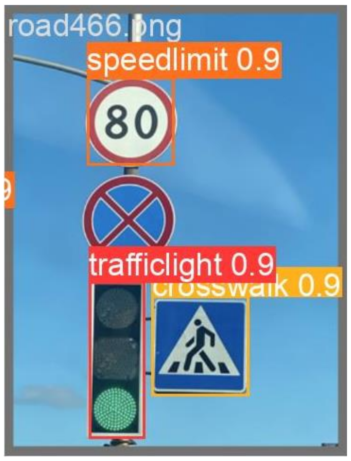


As we see from the images above, most of our mislabeling and bounding box inaccuracy may come from several sign objects of different classes in the same image. We could solve this with multiple bounding boxes in the images and a label for each box. This will require some work on the dataset we have but we may find quicker fixes and better accuracy from another neural network architecture that already exists. While the VGG-16 model does very well without retraining any pre-trained weights, I'd like to try other models first before working on the dataset and spending hours tuning this model.

YOLOv5 Model

The next CNN architecture looked at will be the YOLOv5 Model. This is an object detection model that can be trained with bounding boxes around the objects we are looking for and the class labels we are predicting. It then will break the images into grids and apply filters to seek out objects in the image. The model then puts a bounding box around the objects it has predicted to have a high probability of being one of the objects it has trained to find. Along with the box, the model will list the predicted class and class probability above the box. After training this model, we can have it detect on images and it will locate any number of objects in the image and show them with a bounding box if the probability is over the given threshold that the user applied. This can fix the dataset we have with multiple objects of differing classes in the same image and is exponentially better because we can detect every object of interest within the image. As we train this model, it will develop its weights and prediction power using a metric of mAP (mean average precision) which uses IoU (intersection over union) metrics and precision/recall scores to formulate the mAP scores for the model. The model also uses mosaic augmentation that stitches the training images together to learn to identify objects that may not be fully pictured and predicting objects at a smaller scale. After training and running the yolo model on the test data, images are shown below of what the model found. Any of the sign or light objects detected that had a probability over 0.4 was labeled.





As can be seen, this model is much better for what we are wanting and is more accurate as well, the mAP score of the model being 96% after 56 epochs.

Summary:

Both models worked well that we implemented. Neural networks do well to interpret and predict object detection, just be careful not to overfit with your training data. The VGG-16 model we created did well without much tuning. The YOLO model seems to be what performance we are looking for with the code ready to perform multiple bounding boxes within an image while giving better bounding box locations using IoU and trains rather well with limited data. The creators of the YOLO architecture, Ultralytics have a python package that you will be able to use and fine tune this model and newer models (currently on YOLOv8) that should help in the processing, training, and testing of images.

The implementation of these models has given some good experience in how easy they are to train and how well they predict upon images with small training sets. These model architectures may also be manipulated with layers, different kinds of pooling, different dropout rates, different number of filters and a ton of other parameters that can be adjusted for optimal performance on a specific dataset. When starting from scratch, the models may at least give you a target accuracy of what can already be attained with current models. Good neural network architectures built for one set or type of data will probably need to be adjusted or manipulated to find its best accuracy when training to solve a new data problem.