DATA612 - Project Final Report
Traffic Sign Classification with YOLO Framework

By: Jaehyun Yoon, Renea Young, Andrew Tran

Table of Contents

Background

Object detection is a computer vision technique that identifies and locates objects in an image or video. The identification and location of objects can be done by creating bounding boxes around the objects and labeling them. Object detection can be applied in a number of ways. Crowd counting, video surveillance, face recognition, and self-driving are some of the primary ways object detection can be applied.

Traffic sign detection is one of the essential technologies in the field of assisting drivers such as Advanced driver-assistance systems (ADAS), transportation systems, and autonomous driving. The world of neural networks and autonomous vehicles is evolving fast. Apple, Volkswagen, Honda, and other companies are relying on artificial intelligence to improve car safety. Traffic sign detection is a challenging task in which the algorithms have to cope with natural and complex environments, high accuracy demands, and real-time constraints. The challenges that arise when detecting traffic signs are blur due to capturing the image from a moving vehicle, displacement of the traffic symbols, faded traffic symbols due to the effect of weather elements, and many more. Many approaches have been proposed for traffic sign detection. This study focuses on using the You Only Look Once version 3 (YOLOv3) framework to detect the German traffic signs detection benchmark (GTSDB) database.

Current State of the Art

Deep neural network methods have become the state of the art approach to traffic sign detection. Various research has rapidly increased along with applying convolutional neural networks and their modified structures. Some state-of-the-art detection methods include Faster Region-based Convolutional Neural Network (F-RCNN), Single Shot Detector (SSD), Region-based Fully Convolutional Network (R-FCN), You Only Look Once(YOLO) and many more.

F-RCNN is similar to Fast Region-based Convolutional Neural Network (F-RCNN) however, it does not use selective search. It utilizes the Region Proposal Network (RPN) and Non-Maximum Suppression. It is applied to choose the best region. RPN is basically a fully convolutional network that simultaneously predicts the object bounds as well as objectness scores at each position of the object. Shao et al. proposed several methods that improve F-RCNN detection performance. The research paper suggested the method of merging the German traffic sign detection benchmark (GTSDB) and Chinese traffic sign dataset (CTSD) databases into one larger database to increase the number of database samples. They also proposed a method that combines the features of the third, fourth, and fifth layers of VGG16.

SSD is a method for detecting objects in images using a single deep neural network. You et al. proposed a method that uses some 1x1 convolution kernels to replace some of the 3x3 convolution kernels in the baseline network and deletes some convolutional layers to reduce the calculation load of the baseline SSD network. R-FCN is a region-based detector. It is fully convolutional with almost all computation shared on the entire image. All learnable weight layers are convolutional and are designed to classify the region of interest into object categories and backgrounds. There has been recent research that implemented an improved R-FCN framework that detects traffic signs.

YOLO detects objects in real time and provides the most accurate and fastest object detection results. Recent research implemented YOLO in Tensorflow and Keras. As stated before, traffic detection is a challenging task. Wan et al. proposed a model that can detect traffic signs under severe conditions. Traffic Sign Yolo (TS-Yolo) is based on the convolutional neural network to improve the detection and recognition accuracy of traffic signs, especially under low visibility and extremely restricted vision conditions.

What is YOLO?

You Only Look Once (YOLO) is an object detector that detects objects in real time. It was developed by Joseph Redmon et al. and published in a paper in 2015. YOLO has three variations, YOLOv1, YOLO9000/YOLOv2, and YOLOv3. YOLO version 1 architecture was inspired by GoogLeNET (later calledDarknet based on VGG), and it performs downsampling of the image and produces final predictions from a tensor. Some disadvantages of YOLO version 1 include more localization error compared to Faster-CNN and the struggle of detecting close objects and small objects. YOLO version 2 architecture uses a 30-layer architecture, which consists of 19 layers from Darknet-19 and an additional 11 layers adopted for object detection purposes. YOLO version 3 provides more accurate and faster object detection results. It has 106 layers and uses Darknet-53 as its backbone.

Model Framework

YOLO v3 has the three main features compared to the previous version. First of all, the execution time for YOLO was faster than other networks, but its prediction accuracy was lower. However, YOLO v3 makes detections at three different scales of anchor boxes with 3 bounding boxes at each scale (Figure 1), which is a similar technique of Feature Pyramid Network (FPN) (Figure 2). The Feature Pyramid Network (FPN) is where you create the layers of feature map and combine all layers back using the upsampling features. If you use FPN, you can help

prediction by reflecting the feature maps at the upper level in the feature map at the lower level.In those 3 different feature maps, the first map which is the size of 13 by 13 map finds small images, the second map which is the size of 26 by 26 map finds medium-sized images, and the third map which is the size of 52 by 52 map finds large images. After each prediction feature map, it doubles the size to match with the next feature map using the upsampling layer. That's the main difference between the YOLO version 2 and the version 3.

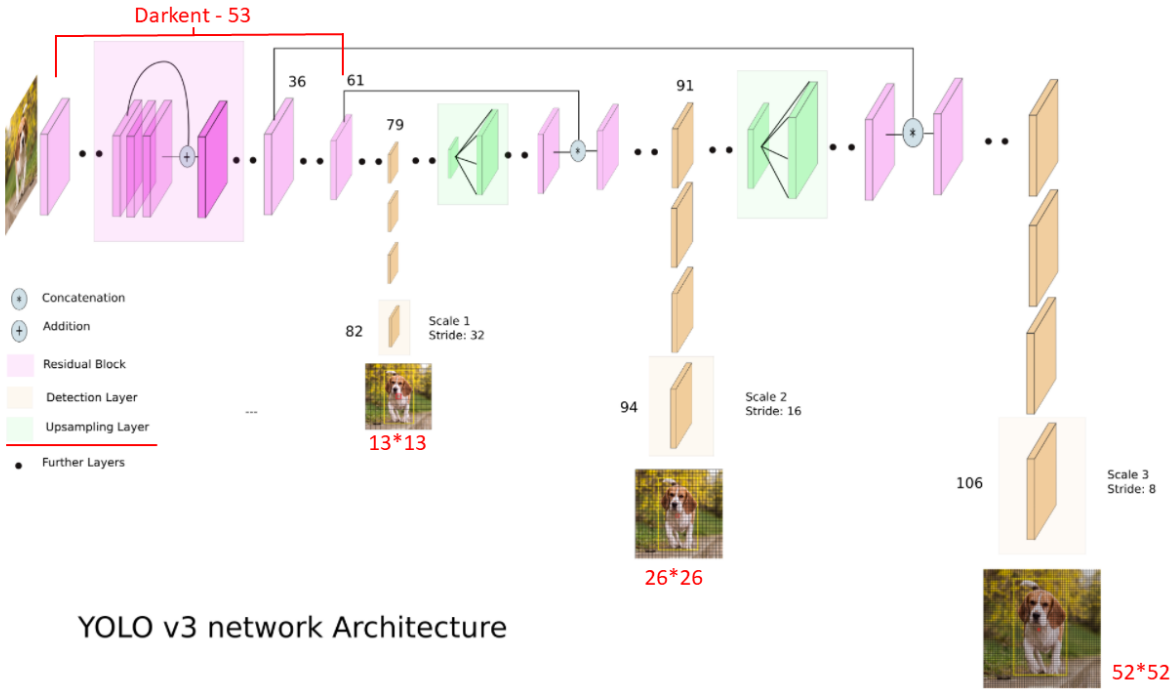

Figure 1 : YOLO v3 network Architecture

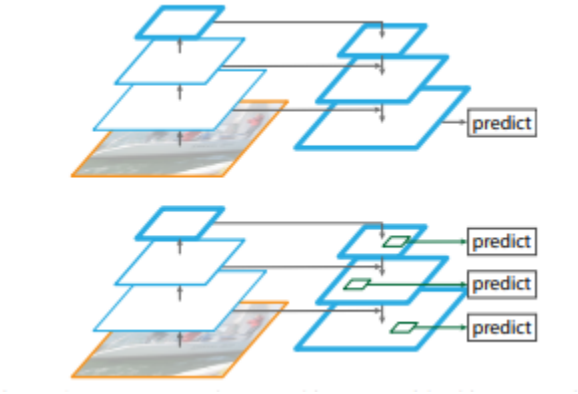

Figure 2 : Feature Pyramid Network

The tensor for each prediction feature map calculates as N(=the size of feature map) * N (B(=3)* (Box co-ordinates + Objectness score + Class scores)) (Figure 3). The box co-ordinates have four values which are the center X, the center Y, height, and width of the bounding box. The Objectness score which is the confidence score can be calculated as the Intersection of the

Union (IoU), the area of overlap over the area of union, times the probability of object. If there is no object in the bounding box, then the probability becomes zero and the objectness score is also zero. The goal of the training is where we adjust the confidence score to find the best cost function.

The second difference is the backbone network. For the backbone network, YOLO uses darknet which is better than the ResNet network. The number, 53, means that there are 53 layers including the residual blocks and the sum of 53 convolutions. According to the YOLO v3 paper, Darknet-53 has the highest measured floating-point operations per second which means it makes the network more efficient to evaluate and faster. The number, 53, means that there are 53 layers including the residual blocks and the sum of 53 convolutions. The last difference is that the YOLO v3 also performs multi-label classification which means it uses the logistic sigmoid classification, binary classification instead of the softmax. They declared in the paper that the logistic classification can have a better performance in YOLO than the softmax.

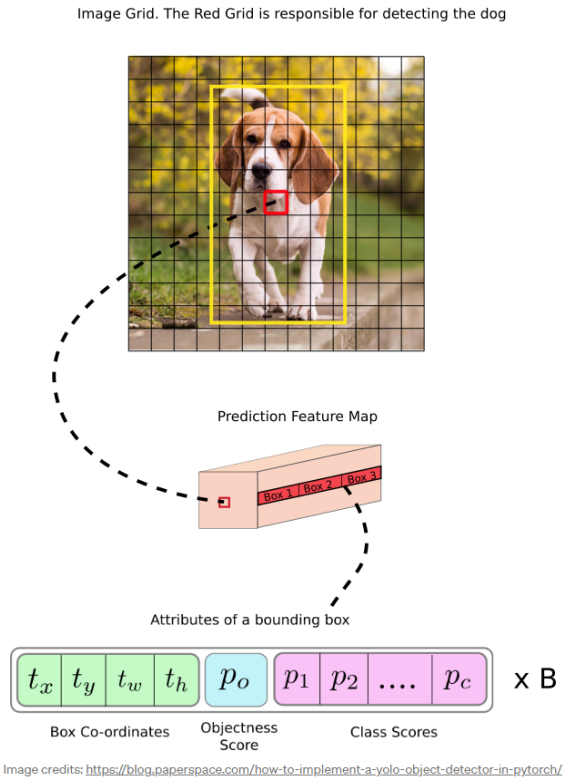

Figure 3 : Prediction Feature Map

The Data

For our project of traffic sign detection, we are using data sets compiled by the Institute Fur Neuroinformatik based in Germany. The data sets used in this study are specifically provided by the group for the purpose of solving computer vision and machine learning problems. The two data sets they provide are the German Traffic Sign Detection Benchmark Set ("GTSDB") and the German Traffic Sign Recognition Benchmark ("GTSRB") set. The GTSDB set is composed of 900 images of landscape photos, each of which includes some aspect of traffic signs in different scenes to be used for solving object-detection problems. The GTSRB is composed of over 50,000 images of zoomed-in traffic signs to be used for solving classification and recognition problems. The traffic signs within both of these sets contribute to a total of 43 different traffic sign classes.

The GTSDB data set was provided in a folder of about 2.9GB in size, with each image in a .ppm format containing the image details in a text format. The data in the .ppm file represents the images in their original dimensions, therefore, the data will need to be reformatted before running the model. Each image is also accompanied by a separate .txt file that contains the label details for the class of each traffic sign in the image, as well as the coordinates of where the traffic signs are located within the image for training and testing purposes.

The GTSRB data set was provided in a .pickle file of about 1.3GB in size containing a binary encoding of the image pixel details as well as their labels corresponding to the 43 classes. This .pickle file also includes a preset 83%-training, 4%-validation, and 13%-test split on the data that we can utilize for testing our model. The .pickle file holds preprocessed data of the images where they have been normalized and standardized so the pixel values are within the range of 0 to 255.

The images below display examples of the images within each data set:

Figure 4: Example of a landscape image from the GTSDB data set

Figure 5: Examples of various zoomed-in traffic sign images from the GTSRB data set

Implementation

YOLO Implementations

Preparing the Data

The YOLO requires the user to provide the input in the YOLO format, which is in the order of class id, the center X, the center Y, and height, and width for each image. The dataset provides a Roi of X and a Roi of Y for each object. We need to calculate the center X, the center Y, height, and width based on the provided Roi. We customize the configuration file for our model in which we need to set the number of batches, the max batch, number of classes, and filters. For the max batch, Darknet recommends setting the max batch as the number of classes times 2000. Since we have limited GPU and CPU to train our weights, it wasn't feasible to train the weights with all 43 classes in our local machine. We decide and divide into 4 different classes which are "Danger", "Prohibited", "Mandatory", and "other". With only the 4 classes of weights, it took us 6 to 8 hours to get the weights in the cloud machine. Darknet provides a function to save the best weight file and the backup files for every 1000 epochs in case of getting an error and being stopped during the training.

Building the Model and Predicting with the Model

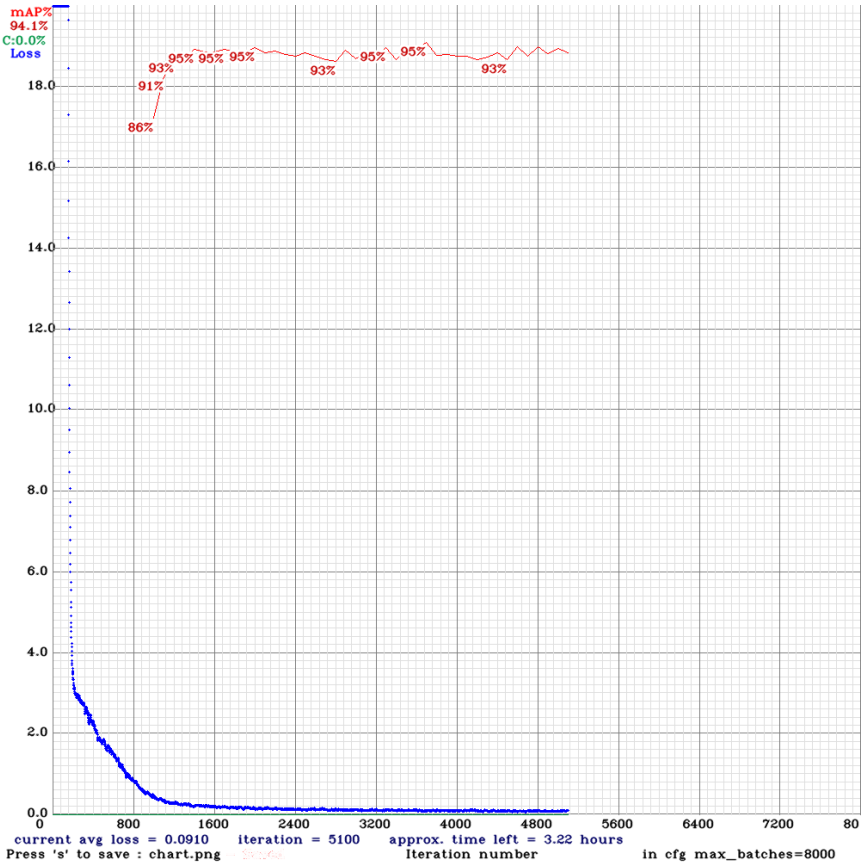

Figure 6 : The loss value and the mAP value for our weights

The graph shows how the loss value and the mean average precision (mAP) changed as we train the weights. As you can see in the Figure 6, the result of loss value dropped right about the 400 epochs and the mean average precision showing at about the 1000 epochs. Since we have max batches as the 8000 (4 classes *2000), it will run until 8000 epochs. We decided to stop at the 5000 epochs instead of all the way to 8000 epochs to prevent overfitting. The best mean average precision (mAP) was about 95.35 percent and the loss value got to close to zero which was 0.000010.

Classification CNN Model Implementation

Preparing the Data

For classification, we used the which we downloaded in a .pickle file of binary encoding. To prepare the data for training the classification model, we needed to import the data into a usable array format by importing the “pickle” package and using its “load” function with and specifying the “encoding” parameter to reformat the data. Then, we must pull out the predetermined training, validation, and test data split designed in the source data. We must also

encode the target variables into categorical labels set to the desired 43 classes to fit with our classification model. Lastly, we need to redimension the image data so that the channels-dimension (for the three color channels) are the last dimension of the input data. After completing all of these steps, the data is now ready to be fed into the classification model for training.

Building the Model

In building the image classification model, we first started out with using the popular AlexNet architecture that is known for being able to efficiently and accurately predict labels of image data, such as the well-known MNIST handwritten-digits and flower-iris data sets. AlexNet is known for having eight main layers: five convolutional layers and three fully connected layers. Despite the architecture's strong reputation, we found that AlexNet yielded poor accuracy on the test data set, sitting at around 55% accuracy. Interestingly, in further tests, we gradually found that removing layers from AlexNet started to improve accuracy. After testing multiple architectures, we decided on implementing an eight-layer architecture with only two convolutional layers, two max pooling layers, and two fully connected layers. This architecture looked to yield the best training and testing accuracy on this data set. The full architecture for our convolutional classification model can be seen in the image below:

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_7 (Conv2D)            (None, 32, 32, 32)        2432
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 16, 16, 96)        76896
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 8, 8, 96)          0
_________________________________________________________________
flatten_4 (Flatten)          (None, 6144)              0
_________________________________________________________________
dense_9 (Dense)              (None, 1000)              6145000
_________________________________________________________________
dropout_5 (Dropout)          (None, 1000)              0
_________________________________________________________________
dense_10 (Dense)             (None, 43)                43043
=================================================================
Total params: 6,267,371
Trainable params: 6,267,371
Non-trainable params: 0
_________________________________________________________________
```

Figure 7: Visualization of the network layers of the CNN classification model

After deciding on a single architecture, we then trained this on the GTSRB dataset with the preset training, validation, and test split of 83%, 4%, and 13%, respectively. With this final architecture, we also tested different batch sizes and epoch parameters to find what combination maximized testing accuracy on the GTSRB data set. In these tests, we tested batch sizes of 5, 10, 50, and 100, and tested epochs of 3, 5, 10, and 15. In each of these tests, we tracked the total time needed to train the model, the training accuracy, the validation accuracy, and the test accuracy for comparison purposes. The metrics found from each test can be seen in the table below:

Batch Size	Epochs	Time (seconds)	Training Accuracy	Validation Accuracy	Test Accuracy
5	3	1,246	95.36%	89.05%	89.02%
5	5	2,615	97.05%	92.36%	91.20%
5	10	5,957	97.76%	91.41%	91.07%
5	15	8,084	98.36%	95.16%	91.11%
10	3	775	96.69%	92.18%	90.77%
10	5	1,526	97.96%	91.11%	90.31%
10	10	3,441	98.47%	91.95%	92.09%
10	15	6,036	98.90%	94.24%	93.27%
50	3	420	98.43%	93.02%	92.47%
50	5	776	99.14%	93.85%	92.67%
50	10	1,327	99.40%	95.12%	94.20%
50	15	1,997	99.54%	94.92%	94.11%
100	3	295	99.78%	94.20%	94.34%
100	5	456	99.80%	95.12%	93.94%
100	10	964	99.80%	95.08%	93.67%
100	15	1,296	99.83%	95.99%	94.74%

Figure 8: Chart displaying the training metrics gathered from experiments on batch size and epochs of CNN model

In this table, the highlighted rows display the test of each batch size that resulted in the highest prediction accuracy on the test set. We can see that with each batch size, increasing the number of epochs gradually improved the test accuracy. We can also see that increasing the overall batch size reduced the training time significantly while also improving the test accuracy. From these tests, we then decided to use the classification architecture with a batch size of 100 and epoch-value of 15 where the training time took approximately 1,296 seconds and yielded a prediction accuracy on the test set of 94.74%. A graphical visualization of the testing accuracy across batch sizes and epochs can be seen in the line graph below:

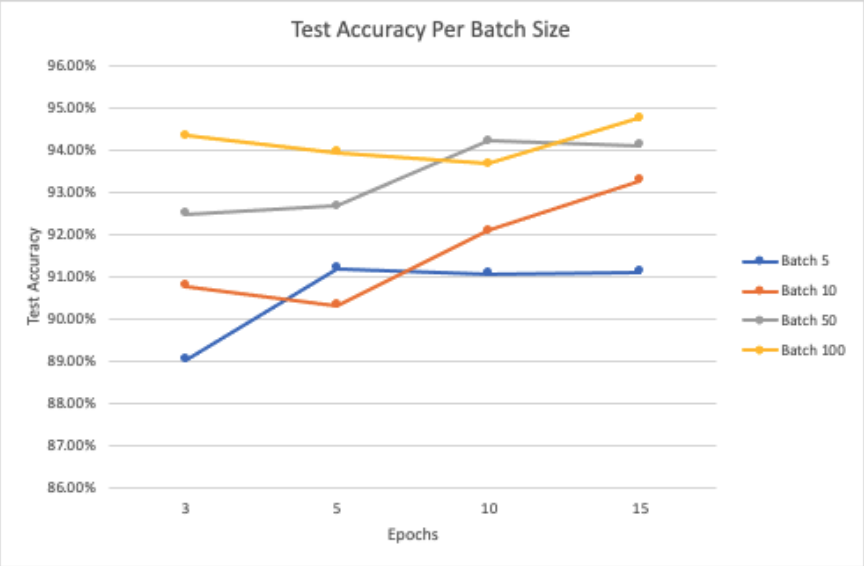

Figure 9: Chart comparing the test accuracies of various combinations of batch sizes and epochs on CNN model training

Predicting with the Model

With a final model in place, we then are ready to predict the class-labels of traffic signs in other images. By first utilizing the YOLO model on the prediction image, we are left with the coordinates of bounding boxes identifying the objects of interest within the image, in this case, traffic signs. With those bounding-box coordinates, we can then slice the original image into separate, smaller images that only contain the traffic signs. With these sliced images, we can then feed them into our classification model as they are similar in design and format to the training images in the GTSRB set.

To feed these images as inputs into the classification model, we must convert the sliced image from the output of the YOLO model into the proper format for the classification model. For this, we redimension the image to a 32x32 format with standardized values between 0 and 255. Lastly, we then transpose the dimensions of the image data so that the dimension for the image-channels is the last dimension. This reformatted data now allows us to input the image into the model for prediction, where the outputs will be the predicted traffic-sign class of the traffic-sign image. We can then combine the predicted label found from the classification model with the bounding box coordinates found in the YOLO model to visualize the predicted results from the image.

Results

To test and visualize the effectiveness of the combination of the two models' predictions, we inputted two different, landscape images for traffic sign recognition and classification. The two sets of two images below display the predicted bounding boxes from the YOLO model and the predicted class-labels from the classification model overlaid onto the two original images. The top image in each set shows the full image with the overlaid prediction, while the bottom image in each set is a zoomed-in version to better read the predicted labels.

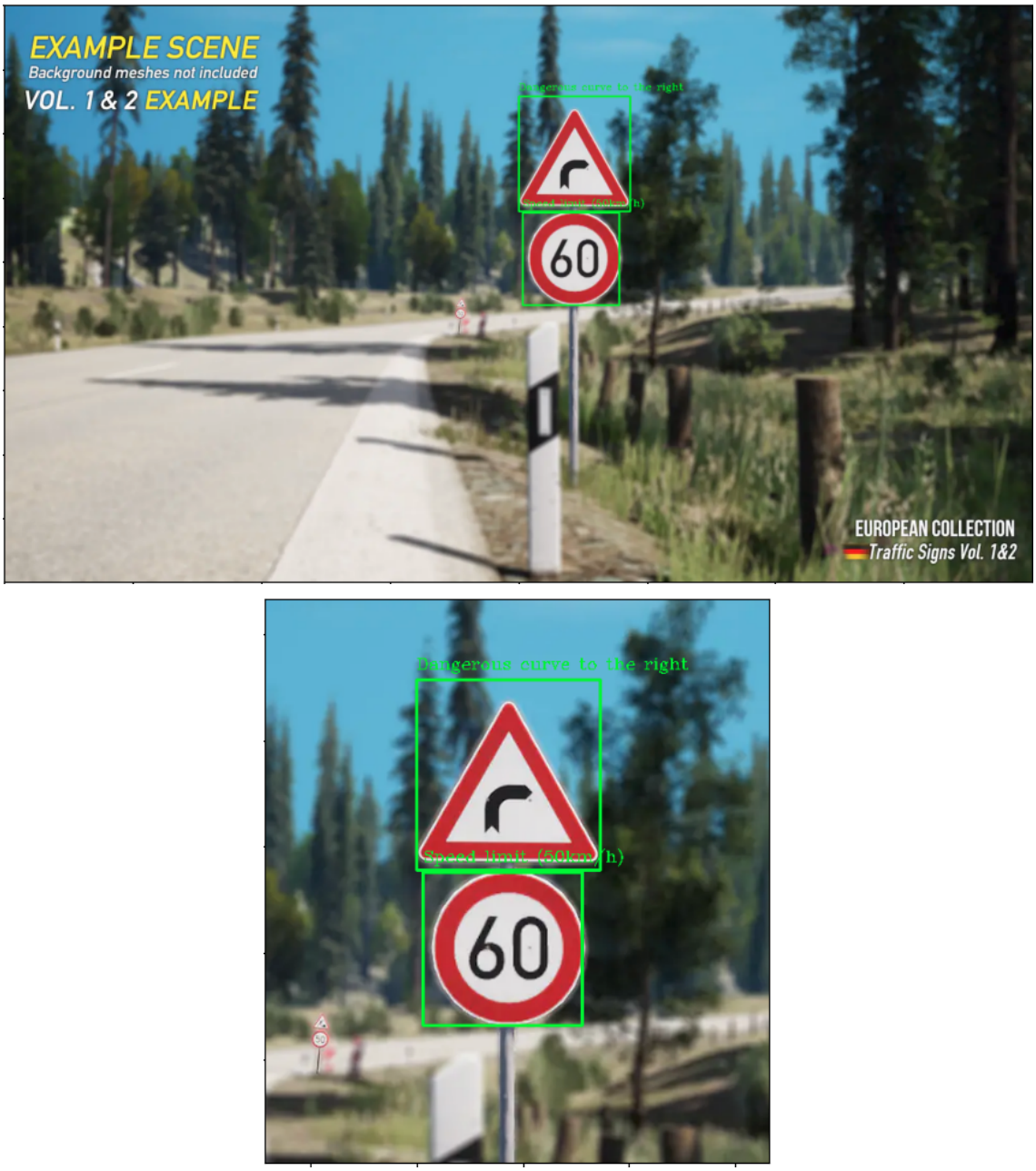

Figure 10: Visualization of prediction outputs of YOLO and CNN classification models

Figure 11: Visualization of prediction outputs of YOLO and CNN classification models

In the first set of predicted images, we can see that the YOLO model accurately identified the locations of the two traffic signs by seeing that the green bounding boxes accurately encapsulate each sign. The classification model, however, showed impressive yet slightly less accurate predictions of the sign-classes as seen in the labels associated with each image. The classification model looks to have classified the top sign correctly as the sign for “dangerous curve to the right” but has incorrectly classified the bottom sign as “speed limit (50 km/h)” when the sign is actually representing a speed limit of 60 km/h. Despite the incorrect classification of the bottom sign, the model was still able to identify that the sign is a speed limit sign which is very close to the accurate speed of 60 km/h. In analyzing this result, the number “50” and the number “60” can subjectively be considered very similar in shape and size, thus it may be understandable for the model to have confused the predicted classes.

In the second set of predicted images, we can see that the YOLO model accurately identified the traffic signs in the image and the classification model accurately classified the identified traffic signs into the correct classes. The images show that the bounding boxes are correctly encapsulating the three traffic signs in the image and the classification model has correctly identified the classes of those signs as “keep right,” “yield, ” and “roundabout mandatory.” Though all three of these signs look to have been correctly identified and labeled, we can also notice that the YOLO model did not identify the “pedestrian crossing” sign to the far right of the image.

Though both the YOLO model and the classification models performed impressively on these two test images, there is still room for our models to grow in terms of accuracy. To further improve the performance of the YOLO model, we could improve the quality or increase the size of the training data for the YOLO model. With better quality data and a larger sample size of data, the model could potentially be trained to better find traffic signs that would otherwise have been missed during prediction. To better improve the accuracy of the CNN classification model, we could include augmented images as part of the training data to train the model on a wider range of scenarios that it may encounter when scanning through test-image files. Augmentation can include things such as image rotation, change in image size, change in image brightness, and others. We could also test the model’s accuracy by increasing the number of epochs to see if the model can learn more with more iterations, though this may come at the cost of increased training time.

Novelty

Overall, our novelty in improving the current state of the art of object detection with the YOLO framework is by improving the speed of training to more quickly and accurately identify and classify traffic signs in images. The YOLO framework is always being improved for increased speed and accuracy of object detection and class prediction, training speed will always be a risk when the number of output classes increases, as more filters are needed in each of the over 100 network layers to account for the increased number of classes. Because of this risk of slower training, many YOLO models often begin with pretrained weights to save on training time. In our project, we trained the YOLO model on a more general 4 classes instead of the more specific 43 classes of traffic signs. Training the YOLO model on 4 classes alone took our system almost 8 hours to train. Training on the 43 classes theoretically could linearly increase the required training time and would therefore require days to train the model to predict on all 43 classes. Our implementation utilized the best aspects of both the YOLO model and CNN models to reduce training time while accurately classifying images into one of 43 classes. We found that the bounding boxes generated by the YOLO model were the more important output to our project rather than the class predictions. The boxes identified the desired objects for us within the

greater image which we could then feed into our much quicker, yet also effective, 8-layer classification model. Training our classification model required less than 25 minutes to train on our system, leading to a total maximum training time of 8.5 hours to train our entire model-structure. Though in this case the prediction wasn't completely accurate, a difference of total training time between a few hours with our models to a few days with a full YOLO model on all 43 classes allows for other improvements to be added to improve model accuracy. Between the difference in training times, many different improvements could be added to improve our model's accuracy while still coming in at a faster total training time than what would be required by a full YOLO model on all 43 classes.

How to Run the Python Script

The main Python script for our project can be found in the "Python Code and Related Data" folder and is titled **"MSML612_Project_Main Code.py"**. The Python script can be run directly from the folder and should automatically pull the required source data which is also included in the same folder. When opening and running the code in Visual Studio Code, pressing "Run" from the top menu and "Start Debugging" should take the user through the entire script. Various image plots may pop-up on the screen during the runtime and pause the runtime, but runtime will continue when the plots are closed. The final output of the script will be a pop-up of a test image with predicted bounding boxes and class-labels layered on top of the image.

Source data required to run the script which are included in the folder are:

- File for labels of the 4 classes for the YOLO model - **"classes.model"**
- File for the weights for each layer associated with the YOLO model - **"yolov3_ts_train_5000.weights"**
- File containing the architecture and layers to run the YOLO model - **"traffic-sign-yolo.cfg"**
- File for the source data for the CNN classification model - **"data2.pickle"**
- File containing the 43 class-labels for the CNN classification model - **"label_names.csv"**
- File used to run the final prediction experiment on both models - **"00001.jpg"**

The following Python packages must be installed on the environment to run the script:

- CV2
- Pandas
- Pickle
- Matplotlib
- Keras

The file for pretrained weights for the YOLO model from our training of the YOLO model is already included on in the folder with the source data, but the section below details how to run the Jupyter Notebook used to run the training of the YOLO model to create the pretrained weights.

The Jupyter Notebook to train the YOLO model to identify optimal model-weights can be found in the "Python Code and Related Data" folder and in the "MSML612_TrafficYOLO_Weight Training Files" subfolder. The notebook is then titled **"MSML612_TrafficYOLO_weight.ipynb"**. For our model weights, we had to use the cloud machine and the Google Colab notebook because of the limitation of GPU and CPU of our machine. The link is in the .ipynb file and you can download our datasets and train the weights if you need to. The only thing you need to do is change the path for the configuration file to where you save the traffic-sign-yolo.cfg file. Once you do that, you can run other cells for converting data into YOLO formats, data preparation, and the pre-train model from the darknet Github. The final and the best weight file is the yolov3_ts_train_5000.weights.

Note: We created a script to read in a video file, run predictions on the frames of the video file, and visualize the predictions but the script was created to run in a Google Collab environment and does not seem to run properly on an offline environment. We have kept this portion of the script in a commented-out section at the end of the main Python script, but for the purposes of this project we are focusing on predicting on single images. The inputs and outputs of this section of the script when run on a Google Collab environment can be seen in the folder with the source data as **"traffic-sign-video.avi"** as the input data, and **"YOLO_traffic_output.mp4"** as the output data.

Contributions

Andrew - Designed and implemented the training and testing of the CNN classification model. Implemented the slicing of the overall predicting image based on the output bounding boxes of the YOLO model to then feed into the CNN model to predict the class of the identified traffic signs in the image. Subsequently contributed to the write-up related to the implementation of the classification model, results, and novelty of our project, along the presentation of the classification model in the PowerPoint slides.

Jae - Developed functions to create and convert the required information into YOLO format and optimizing the configuration file to fit in our datasets and weights. Designed and implemented the training weights after the data preparation and provided all information about our YOLO v3 framework and architecture in the report and the PowerPoint slides.

Renea - Researched related papers on object detection, YOLO framework and state of the art. Contributed to the final report and presentation slides. Worked on the background, current state of the art, and what is YOLO sections.

References

- Bochkovskiy, Alexey, et al. “YOLOv4: Optimal Speed and Accuracy of Object Detection.” *ArXiv*, 23 Apr. 2020, arxiv.org/pdf/2004.10934.pdf.
- Tianxiaomo. “PyTorch, ONNX And TENSORRT Implementation Of YOLOv4.” GitHub, github.com/Tianxiaomo/pytorch-YOLOv4.
- Gupta, Mehul. “All about YOLO Object Detection and Its 3 Versions (Paper Summary and Codes!!).” *Medium*, Data Science in Your Pocket, 20 Apr. 2020, medium.com/data-science-in-your-pocket/all-about-yolo-object-detection-and-its-3-versions-paper-summary-and-codes-2742d24f56e.
- Real-Time Computer Vision. “Welcome to the INI Benchmark Website.” German Traffic Sign Benchmarks, 16 Sep. 2010, https://benchmark.ini.rub.de/.
- V. Meel, "YOLOv3: Real-Time Object Detection Algorithm (What's New?) | viso.ai." *Computer Vision Application Platform | viso.ai.* Viso.ai, 25 Feb 2021. Web. 28 Jul 2021. <http://viso.ai/deep-learning/yolov3-overview/>.
- J. Redmon and A. Farhadi, “YOLOv3: An incremental improvement,” Volume 4, *arXiv*, 2018.
- Brownlee, Jason. “How to Perform Object Detection with YOLOv3 in Keras.” Machine Learning Mastery, 27, May 2019, https://machinelearningmastery.com/how-to-perform-object-detection-with-yolov3-in-keras/.
- You, Shuai, et al. “Traffic Sign Detection Method Based on Improved SSD.” *Information*, vol. 11, no. 10, 9 Oct. 2020, p. 1-16., doi:10.3390/info11100475.
- Shao, Faming, et al. “Improved Faster r-Cnn Traffic Sign Detection Based on a Second Region of Interest and Highly Possible Regions Proposal Network.” *Sensors*, vol. 19, no. 10, 17 May 2019, pp. 1–28., doi:10.3390/s19102288.
- Wan, Haifeng, et al. “A Novel Neural Network Model for Traffic Sign Detection and Recognition under Extreme Conditions.” *Journal of Sensors*, vol. 2021, 10 July 2021, pp. 1–16., doi:10.1155/2021/9984787.
- Kathuria, A. (2018, April 29). *What's new IN YOLO V3?* Medium. https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b.
- Kathuria, A. (2019, December 2). *Tutorial on implementing YOLO v3 from scratch In pytorch*. Paperspace Blog.

https://blog.paperspace.com/how-to-implement-a-yolo-object-detector-in-pytorch/.

- Dollar, P., Lin, T.-Y., Girshick, R., Hariharan, B., & Belongie, S. (2016, December 9). *Feature Pyramid Networks for Object Detection*. https://arxiv.org/pdf/1612.03144.pdf.