ECSE 415 - Final Project
Detecting, Localizing, and Tracking Vehicles

Tyler Watson
260867260
tyler.watson2@mail.mcgill.ca

Anthony Porporino
260863300
anthony.porporino@mail.mcgill.ca

Felix Simard
260865674
felix.simard@mail.mcgill.ca

George Kandalaft
260866146
george.kandalaft@mail.mcgill.ca

Kyle Myers
260851765
kyle.myers@mail.mcgill.ca

April 11th, 2022

Abstract

The goal of this project is to build a computer vision pipeline that can detect, localize and track vehicles in a given input video.

1 Introduction

Over the course of several weeks, our team has brainstormed, designed, implemented, and experimented various techniques to best classify, detect, localize and track vehicles moving across different image frames. This report will highlight the methods and results for each of the listed sections of the project. All team members provided a significant and equal contribution to the project.

2 Dataset

The provided dataset was sourced from the KITTI vision benchmark [1] and contained images extracted from four separate video sequences, each at a resolution of 1242x375. There was also a text file for each video sequence which provided ground truth details about object locations and types in every frame of each video sequence. Finally, a tracking video sequence was also provided.

2.1 Initial patch extraction

The goal of the patch extraction phase was to formulate another dataset based on the original that we could use as training and testing data in our classifiers. This phase extracted certain subsections of all provided images and saved them for further use. Firstly, for each image, the bounding boxes of relevant objects (vehicles only) were parsed from the provided text files. These bounding boxes were then utilized to extract patches that contained vehicles. There was also a need for non-vehicle patches which only partially overlapped with vehicle patches. In order to achieve this, for each vehicle patch, four non-vehicle patches were extracted from the same image. Each of which had only a quarter of the vehicle on one of the four sides (top, bottom, left and right) of the patch. Thus, there was a 1:4 ratio of vehicle to non-vehicle patches. The patches were stored in four NumPy arrays based on which video sequence they were extracted from. The corresponding labels for each patch were also stored in NumPy arrays with a label of 1 for vehicle patches and a label of 0 for non-vehicle patches. An example of these patches can be found in Figure 9 in the Appendix.

2.2 Further patch extraction

It was quickly determined that this initial dataset would not suffice in order to generalize well to other datasets. We determined that more non-vehicle images were necessary due to the high false positive rate. Thus, for each image in the dataset, full non-vehicle patches were extracted. This included patches of trees, roads, buildings, and anything else that was not a vehicle. This allowed for more robustness and less false positives. The final addition made to the dataset was the extraction of more vehicle patches. For each vehicle patch, four patches were extracted with three quarters of the original vehicle patch on one of the four sides (top, bottom, left and right). This gave many more positive samples which helped our classifier determine when a vehicle was truly there. Once again, examples of these can be found in Figure 9 in the Appendix.

2.3 Dataset statistics

In total, the final dataset version contained a total of 51 123 patches. There was a fairly even split between vehicle and non-vehicle samples as demonstrated in Figure 1. A distribution of samples from each dataset can be viewed in Figure 10 in the Appendix. These ranged in various heights and width but the mean height was 70 pixels and the mean width was 110 pixels. This led to around a 1.8 mean aspect ratio as seen in Figure 2. These numbers were in accordance with our deduction upon going through the dataset that most vehicle patches were wider than they were tall.

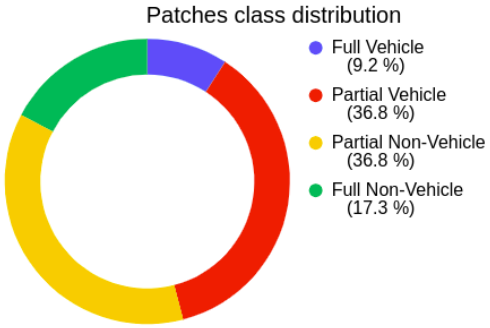

Figure 1: Class distribution of final dataset

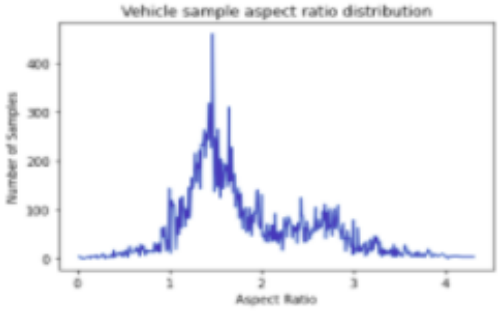

Figure 2: Aspect ratio distribution

3 Classification

The classifier is a crucial piece of this entire system. The goal of the classifier is to be able to determine whether a given image is of a vehicle or not. Our classifier initially used SIFT features descriptors but quickly changed to HoG features and observed significant improvement. Using the HoG features extracted from our image dataset, we were able to train an SVM, Random Forest and Logistic Regression model able to classify new images as vehicle or non vehicle. We also trained a deep learning model to compare with the performance of our non deep learning methods.

3.1 Feature Extraction

Our group initially attempted to use SIFT feature descriptors to build our classifier. SIFT features are scale invariant and robust and therefore were thought to provide good encodings of our images. Since there are no standard number of SIFT descriptors for every image, to be able to supply SIFT features to a machine learning algorithm such as SVM, we decided to cluster the features using K-mean clustering.

Firstly, we extracted up to N SIFT features for every image in our dataset. N is simply an integer and is a hyperparameter that we later tuned. We provided all the extracted features to a K-means clustering algorithm to learn cluster centers. Using these cluster centers we are now able to encode an image as a normalized vector of length K (size of clusters). This is accomplished by extracting the SIFT features of an image and for every feature determining which cluster is closest. We add a 1 at index representing the cluster

that feature is closest to in our encoding vector. After each SIFT descriptor is processed, we normalize the vector and have our encoding.

Using the training dataset and an SVM model with RBF kernel, the accuracy of our model was around 35%. By tuning the values for N and K were able to increase this to 63%. We believe increasing the clusters size would further increase accuracy however since these results are far too low, we switched methods completely.

Using HoG features provided a significant improvement in our results. HoG features, as described in class, provide a histogram of oriented gradients. For each pixel the gradient magnitude and direction are calculated from the x and y gradients. We divide all pixels into groups of cells and use the direction to divide each pixel in the cell into bins [2]. From here we can normalize across cells and output the bin vectors for each cell as the final encoding. The important aspect of using HoG is that all images of the same size using the same HoG parameters will be encoded into vectors of the same size. This allows us to directly provide the HoG descriptors to a machine learning algorithm and only requires us to specify the resizing shape for our images.

3.2 HoG Parameter Tuning

The two HoG parameters analyzed were the image resize shape and the pixel cell size. The other parameters were kept constant. Number of bins was set to 9 and block size set to (2,2). The resize shape is used to resize all images before extracting the HoG parameters. Again this is to ensure all vectors are of the same length. **Resizing the image is the only prepossessing step** done before using HoG.

The pixel cell size hyperparamter refers to the number of pixels set as the width and height of one cell during HoG computation. This hyperparameter changes the size of the output vector and can affect the quality of the encoding. Our group used the HOGDescriptor function provided by the cv2 python package from OpenCV [3]. We noticed that the HoG descriptors from the cv2 function performed better than those from skimage HoG function. While the documentation for cv2 HOGDescriptor function is limited, we believe this is because the cv2 function interpolates the magnitude of gradients into bins instead of putting the entire magnitude into one bin.

The values tested for the resize shape were: (110, 72), (128, 64), (64, 32), (256, 124). The first value represents the width while the second represents the height. The resize shape (110, 72) was chosen because it was determined to be approximately the average width and height of all patches in our dataset. The value (128, 64) was chosen since this was the value from the original HoG paper [2]. Note that we inversed the width and height because the original paper attempted to detect people but cars are usually wider rather than taller. We doubled and halved this shape size to get 2 other possible values. The size of the image directly affects the number of features that are produced so by choosing both large and small resize shapes, we will learn the best length for the output vector.

The values tested for the pixel cell size were 8, 16 and 32. 8 is a standard value for pixel cell size but we were interested in seeing if we can increase the cell size and still have the same results. Increasing the cell size would lower the number of parameters and would therefore decrease execution time for our models.

Table 1 shows the average F1 score results on 3 fold validation for most combinations of the two hyper parameters. Certain combinations were not run due to producing too large or too small of a vector size as output. We decided to compare hyper parameters using the average F1 score because F1 score is a measure of both precision and recall. See figure 11 in the Appendix for F1 score formula. We used this metric when comparing classifier results as well. For these tests a Linear Regression model was used simply because it has a direct solution and computes very quickly. We observed that the highest F1 score is with a pixel cell size of 8 and shape of (110, 72). **These are the hyper parameters we chose to use for our classifier.** There are other valid options with similar F1 score's (all with similar vector sizes) but we decided to use these parameters because it had the highest score and (110, 72) was approximately the average width and height

of all the patches from our dataset. While we could have decreased the cell pixel size more or increased the shape further, it would have created too large of a vector size to train using an SVM.

3.3 Models and Results

We trained 3 different non deep learning classifiers. These classifiers were SVM, Random Forest and Logistic Regression. Logistic Regression is a binary classifier that produces a linear hyper-plane that best fits the data. It computes very quickly since a direct solution exists to solve it. Random Forest also creates a linear decision boundary over our dataset while SVM can produce a non linear boundary if the RBF kernel is used. Since we predict our dataset will not be linearly separable we hypothesized that RBF SVM will produce the largest average F1-score.

The hyperparameters analyzed for SVM were simply the kernel function (linear or RBF) while the hyperparameter analyzed for Random Forest was the criterion (gini or entropy). As mentioned, the RBF kernel allows the SVM to create a non linear decision boundary. The criterion for decision trees measures the quality of a split. The gini criterion measures the probability of a random sample being classified correctly, while entropy criterion measures the amount of information loss.

As part of our project we also trained a deep neural network that can classify an image as a vehicle or non vehicle. We are interested to compare how well a deep learning model performs against non deep learning alternatives. The network architecture we chose to implement is the one of the well established VGG16 Convolutional Network for Classification and Detection. Figure 3 displays the layers of the VGG16 network architecture.

This network has been shown to produce strong classification results for large image datasets and we believe it will be able to solve our problem as well. As for the concrete implementation of the network, we relied on the *torchvison.models* package to provide us with the prebuilt layers of the deep learning network, while

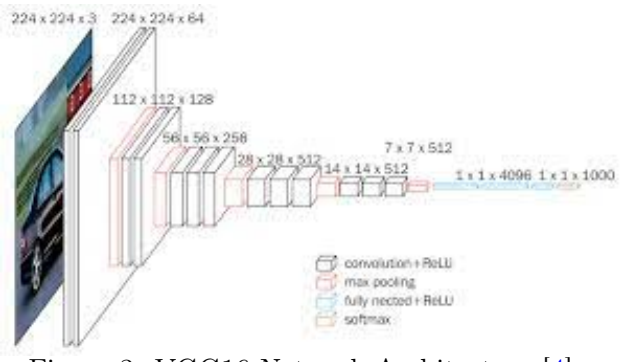

Figure 3: VGG16 Network Architecture [4]

ensuring that we set *pretrained=False* when importing the model. In terms of the dataset, we constructed a custom *VehicleDataset* through which we could feed in combinations of the images folder *0000, 0001, 0002, or 0003.*

Since the network takes long to train and benefits significantly from more data, we only trained this network once.

We set our training set to the *0000, 0001 and 0002* patches folder and the validation/test set to the *0003* folder. After training on 8 epochs, which took around 4+ hours to complete, we were left with a trained VGG16 CNN which we tested on our validation set and received an 89% test accuracy. The accuracy of this deep learning model was significantly higher than those of any non deep learning method. Table 2 shows the results obtained for each of the classification models we tried along with the deep learning test accuracy score. The abbreviation AVG refers to average, STD to standard deviation, ACC to accuracy, PRE to precision and REC to recall. As you can see, we unfortunately were not able to obtain f1-score, precision or recall for the deep learning model as we ran into some issues exporting the model (saving the trained weights) and considering the training took nearly 4 hours, we decided to keep solely the accuracy score for it. While this does limit our ability to know with certainty that the deep learning model out performs non deep learning alternatives, we are fairly sure that the accuracy is a good indication that it will. One reason is because our dataset is evenly distributed between vehicles and non vehicles and that all other models we used had only slightly lower F1 scores compared to accu-

Table 1: HoG Hyperparameter Tuning

AVG-F1	STD-F1	Width	Height	Cell Size	Vector Length
71.3%	1.39%	110	72	8	3456
65.9%	2.12%	110	72	16	540
61.9%	2.52%	110	72	32	72
67.9%	0.95%	64	32	8	756
64.9%	1.45%	64	32	16	108
70.7%	1.39%	128	64	8	3780
68.3%	1.88%	128	64	16	756
63.7%	2.84%	128	64	32	108
70.8%	1.85%	256	124	16	3240
64.8%	0.52%	256	124	32	504

racy scores.

We report that as hypothesized the SVM with RBF kernel produces the highest 3 fold cross validation average F1 score. Allowing the SVM to find a non linear boundary significantly increases the F1 score. The criterion parameter for decision tree did not seem to make much differences as all metrics for Random Forest using gini or entropy criterion were very similar. Logistic regression had similar results to the linear SVM.

In all of our results shown, the accuracy proved to be a good measure of the classifier however there are many examples where this would not be the case. Our group understood early that precision and recall are much more important metrics to measure the strength of a classifier. If 80% of images in our dataset were not of a vehicle our classifier could achieve a high accuracy by labelling most images as non vehicle. This would not indicate that it is a good classifier since it still cannot accurately predict a vehicle. The recall metric for this example however would be fairly low indicating that there were a lot of false negatives. This is one example why accuracy is not always a good measure for a classifier. As mentioned, our group decided to compare our classifiers using the F1 score which weighs both precision and recall evenly and provides a more accurate representation of our classifiers ability to generalize to new data.

3.4 Examples

Figure 4 and 5 show examples of sample images with ground truth labels as well as our classifier predictions.

4 Detection and Localization

The goal of the localization phase is to identify and locate where vehicles were. Numerous ways can help us find the location of an object within an image. After trying different approaches, we decided to use a sliding window technique to loop over the pixels of an image. We then pass each window to our classifier to identify if it is a vehicle or not. We then draw a rectangle surrounding the specified vehicle. The challenging part is merging overlapping boxes. We needed an efficient algorithm that could help us combine overlapping boxes, ideally eliminating all the false positive and false negative rectangles our classifier has identified. Therefore, we implemented non-maximum suppression technique to group rectangles together. We also compared this function's performance with OpenCV's group rectangle function.

4.1 Description of Dataset

See section 2 to view our dataset statistics.

4.2 Description of Detection and Localization Method

The sliding window we built is concise and accurate. Our goal is to capture as many rectangles as possible when passing through the pixels of an image. Therefore, we decided to implement a sliding window that takes an array of different window sizes as an input. We then loop over this

Table 2: Average 3-fold validation results

Model	AVG-F1	STD-F1	AVG-ACC	STD-ACC	AVG-PRE	STD-PRE	AVG-REC	STD-REC
LogReg	71.3%	1.39%	75.6%	1.52%	76.5%	2.50%	67.0%	3.44%
SVM-rbf	80.9%	1.18%	83.5%	1.06%	85.6%	2.65%	76.9%	4.49%
SVM-linear	69.8%	0.98%	72.5%	1.20%	75.8%	2.47%	64.6%	4.56%
RF-gini	75.5%	2.07%	78.7%	1.66%	79.2%	3.99%	72.7%	6.31%
RF-entropy	75.3%	1.68%	78.6%	1.29%	79.5%	4.16%	72.0%	5.94%
Deep Learning	NA	NA	89.0	NA	NA	NA	NA	NA

Figure 4: Predictions examples for *vehicle* label

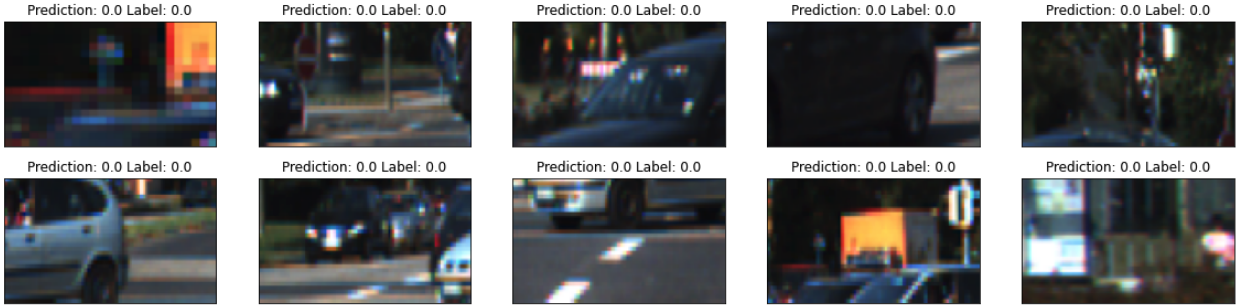

Figure 5: Predictions examples for *non-vehicle* label

array and capture all the rectangles for each size in the array. We also added an overlapping coefficient that allows rectangles to overlap. On average, we found that an overlapping coefficient of 85% gave the best result. We also implemented an image pyramid technique. The image pyramid allows us to use only one sliding window size, but we resized the image multiple times to capture as many windows as possible. This method was more computationally expensive and did not provide any observable improvements. After generating all possible windows, we then pass them to our classifier to label them. We then receive the labelled windows and we group them based on coordinates. To group the images, we implemented a function that checks if two windows are overlapping or not. Finally, we pass our grouped images to our NMS (non maximum suppression) function which combines overlapping windows into one. The NMS function takes each group and extracts x, y, w, and h into separate arrays. Then it returns a single window that consists of min(x), min(y), max(x) + w[x.index(max(x)) - min(x), and max(y) + h[y.index(max(y))] - min(y). We detected that this window gives the smallest false negative percentage as it produces the biggest window that combines all overlapped rectangles.

As an alternative to our NMS function, we used the OpenCV group rectangle function [3]. This function merges overlapping rectangles into one rectangle.

4.3 Results

Table 3 below shows the average IoU across all images for each validation dataset. The classifier used for localization is the SVM RBF classi-

fier trained on the 2 datasets not being tested. The second column specifies which non maximum suppression algorithm is used, either the one designed by our group (Group10) or the group rectangle function from OpenCV [3].

Table 3: Localization Validation IoU

Test Dataset	NMS method	IoU
0000	Group10	12.66%
0001	Group10	26.49%
0002	Group10	9.96%
0000	OpenCV	8.67%
0001	OpenCV	16.00%
0002	OpenCV	9.85%

The average IoU is fairly low for all test datasets and non maximum suppression algorithm. We believe the main reason for this is the errors from our classifier. The different classifiers used for the validation tests had an average F1 score of 81%, but because we are calling the classifier potentially hundreds of times during the sliding window phase, the errors it makes accumulate. This results in many false positives and false negatives. We observed that our NMS function performed better than the group rectangles function from OpenCV. We believe the reason is because our NMS algorithm is generous with box sizes and will create the largest possible merged bounding box. This ensures the intersection between our predicted and the true bounding boxes be as large as possible.

4.4 Examples

Figure 6 found below, as well as Figures 12, 13, 14 and 15, found in the Appendix, show 5 different examples of images with the bounding boxes our localizer predicts on the left and ground truth bounding boxes on the right. For each image we include the next image as well to illustrate that, while our localizer can perform very well in some cases like in Figure 6, it is still inconsistent even with extremely similar, almost identical images.

As we can see our localizer does decently but still has many false positives.

5 Tracking

Tracking is a crucial part of the pipeline of any successful project that aims to detect and localize in a video sequence. It would be far too computationally expensive and unnecessary to run the detector and localizer in each and every frame of a video sequence in order to update the current position of each vehicle. This becomes especially true if the video sequence is comprised of numerous frames. This is where tracking can help significantly. Tracking uses information from a previous frame to determine where the object should be in the current frame. Three tracking techniques were tested in this project: Farneback's dense optical flow, AdaBoost tracking and KCF tracking. Each of these comes with advantages and drawbacks that will be discussed in the following sections.

5.1 Optical flow tracking

The first tracking technique that was implemented was Farneback's dense optical flow tracking. An optical flow representation of a frame takes in information from the previous and current frame in order to decipher how each pixel moves between frames. It attempts to match pixels between frames using their intensity values and, in a dense optical flow implementation, computes the displacement vector for each matching pixel pair. In order to determine how an individually identified bounding box is moving, the algorithm computes the mean displacement vector in the bounding box area and then displaces the bounding box coordinates by exactly this amount.

One such example of the optical flow representation can be seen in Figure 7 below. The camera is moving forwards and so this immobile car moves towards the bottom-left of the image as a result of the camera movement. The green lines representing the motion vectors of a subset of pixels model exactly that movement.

An obvious drawback to this method is that it assumes brightness consistency between frames, which may not always be the case if there is a change in lighting or if a pixel becomes occluded by another object that appears in the second frame. Another drawback is that this method does not allow for the bounding box to change shape to account for, as an example, a car that

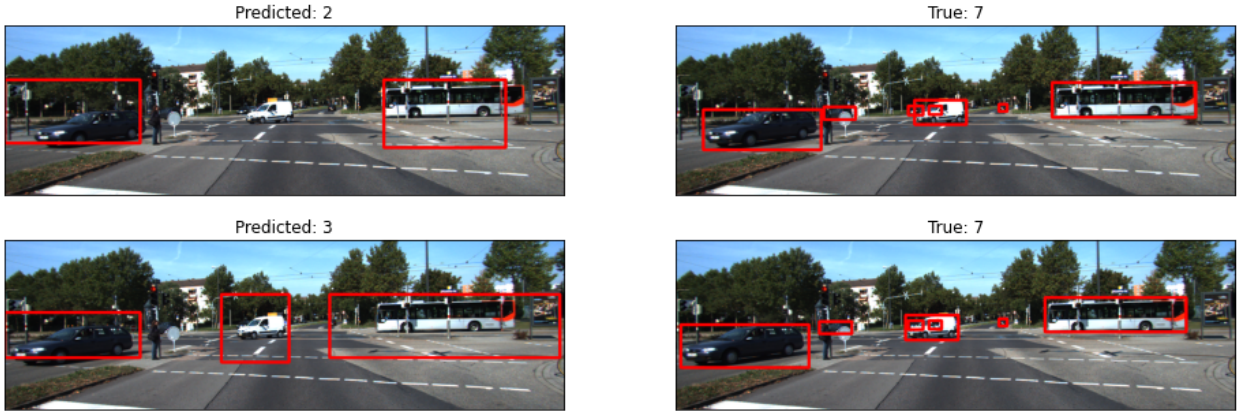

Figure 6: Localize Example 1

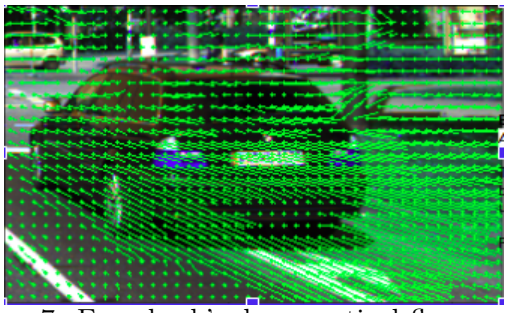
Figure 7: Farneback's dense optical flow motion vectors on a subsection of a video tracking frame

turns where before we saw the side and now we see the rear. However, dense optical flow is quite useful because in most video sequences, there is usually only a small change in both position and lighting and thus optical flow should be able to handle this. It is also quite simple to compute the motion vector of each pixel which allows for quick computation.

5.2 AdaBoost and KCF Tracking Implementation

There were two different tracking techniques used as our alternatives. The first was the boosting algorithm provided by the cv2 library. The algorithm is based on the AdaBoost algorithm, and is quick at segmenting objects from their respective background. By using the background as negative sample, the boosting tracker is able to classify the given features against those around it [5]. Upon qualitative analysis of the tracking performed on validation set 0000, there was a clear problem for when the lighting in the video sequences changes drastically. When the van enters the light from the adjacent street, the track completely looses sight of it due to the difficulty of differentiating the background as negative samples. The next cv2 algorithm that was used is the KCF tracker. This tracker uses circulant matrix properties to speed up tracking by decreasing the time taken to run the Fast Fourier transform algorithm [6]. The KCF tracker does much better at recovery after the obstruction by the cyclist in the 0000 data set. It also recognizes the van after the light difference change. Overall this method performs much better than the Boosting tracker.

5.3 Tracking results

In order to test the accuracy of both tracking techniques, 3-fold validation was performed on the three datasets (0000, 0001, and 0002) by using the associated model for each validation dataset to perform initial localization. The tracking techniques were then compared against the ground truth bounding boxes to produce mean IoU results for each validation dataset. To improve the initially low performance of simply localizing the first image and then performing tracking from there, we experimented with localizing every few images to update any errors that were propagated by the tracking technique. The frequency at which we performed the localization is called the step size.

Figure 8 shows that the optical flow technique was better than KCF tracking only for smaller step sizes. As soon as re-localization frequency was lowered, KCF began outperforming optical

flow. However, both algorithms had peak IoU's of 0.13, 0.26, and 0.10 on the 0000, 0001, and 0002 datasets respectively. Unsurprisingly, when there was no relocalization (step size of 999), KCF outperformed optical flow by 4%.

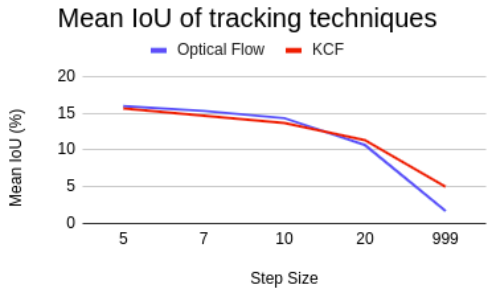

Figure 8: Mean IoU results for both tracking techniques with various step sizes

Also, as seen in Figure 8, a lower step size improved the tracker's performance which indicates that the localization is very useful in helping the tracker stay on track. However, as a compromise between efficiency and accuracy, a step size of around 7-10 was chosen for our final tracker. This is the step size that is used to track the provided tracking video sequence.

The final tracking videos both show tracking that is quite accurate. The localization every 7 frames helps the optical flow tracker especially when the bus turns the corner and is partially occluded by the lamp post. From a qualitative standpoint, the bus is tracked quite well throughout the video sequence.

6 Conclusion

To conclude, this project was definitely a compelling opportunity for us to apply various concepts and tools covered throughout the semester. From data exploration techniques, to pre-processing, to feature extraction, model tuning and training, image partitioning for detection and localization all the way to sophisticated tracking methods. Our team worked very well together, efficiently and fairly dividing all the required tasks to complete the project. We are also extremely satisfied with how we delivered our final demo presentation.

6.1 Key Takeaways

Overall, each step of the project highlighted above clearly show that classifying, detecting, localizing and tracking moving vehicles across frames is not a straightforward task. We believe we managed to extract meaningful and satisfactory results. We now understand the power of leveraging HoG to extract distinguishable features for training our classifier. In terms of the data set, we also realized the importance of appropriately balancing our data set of patches with an even number of vehicles, non-vehicles and partial vehicle patches (labelled as non-vehicles). Finally, we learned the importance of applying detection and localization on different image aspect ratios to maximize the chances of identifying vehicles of varying sizes in a given frame.

6.2 Future Investigation

As potential future works for our project, we think it would be noteworthy to investigate in greater depths the interleaving of our classification model with the detection and localization module. As noted earlier, we believe using sliding window is a valid approach but it seems that the classification errors add up throughout each frame which causes the detection and localization to perform poorly at times. Additionally, as mentioned in the previous section, since detecting and localizing on different aspect ratios is crucial to have a robust implementation, we would like to enhance our localization module to use the output of both *image pyramids* and sliding window technique. The combination of both techniques might have the potential to lower the cumulative errors of our classifier throughout localization. Lastly, while we implemented an initial version of a deep learning classifier based on the *VGG16* network architecture, we would be interested in experimenting with other network architecture from the *torchvision.models* library such as the *GoogleNet*, *AlexNet* or *ResNet*.

References

[1] [Online]. Available: http://www.cvlibs.net/datasets/kitti/.

[2] N. Dalal and B. Triggs, *Histograms of oriented gradients for human detection*, 2005. DOI: 10.1109/CVPR.2005.177.

[3] G. Bradski, "The OpenCV Library", *Dr. Dobb's Journal of Software Tools*, 2000.

[4] *Vgg16 - convolutional network for classification and detection*, Feb. 2021. [Online]. Available: https://neurohive.io/en/popular-networks/vgg16/.

[5] H. Grabner, M. Grabner, and H. Bischof, "Real-time tracking via on-line boosting", English, in *Proceedings of the British Machine Vision Conference*, 2006 British Machine Vision Conference : BMVC 2006 ; Conference date: 04-09-2006 Through 07-09-2006, vol. Volume 1, ., 2006, pp. 47–56.

[6] J. F. Henriques, R. Caseiro, P. Martins, and J. Batista, "Exploiting the circulant structure of tracking-by-detection with kernels", in *Computer Vision – ECCV 2012*, A. Fitzgibbon, S. Lazebnik, P. Perona, Y. Sato, and C. Schmid, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 702–715, ISBN: 978-3-642-33765-9.

A Appendix

Figure 9: Examples of each class in the final dataset patches

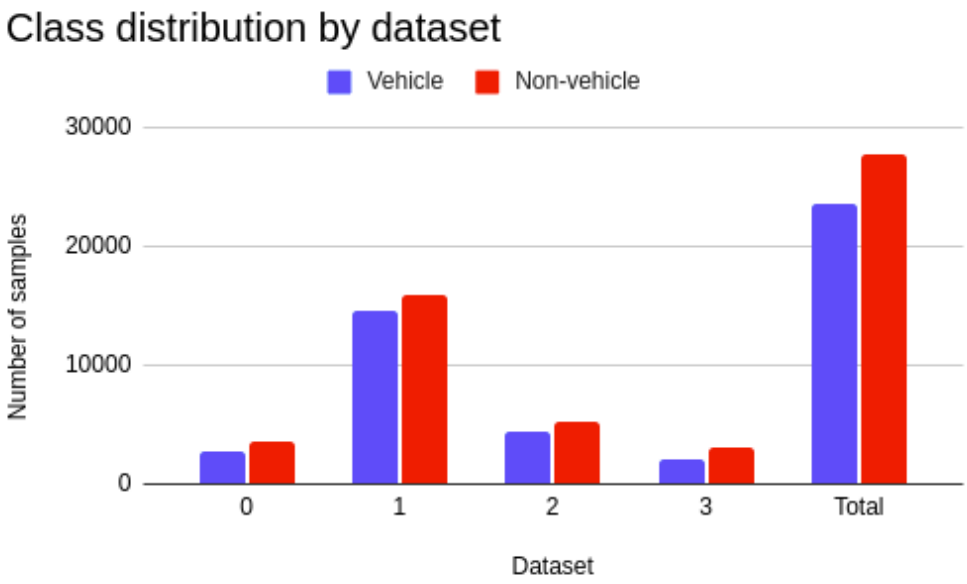

Figure 10: Class distribution of samples in each dataset

$$F1\ score\ =\ 2\ *\ (Precision\ *\ Recall)\ /\ (Precision\ +\ Recall)$$

Figure 11: F1 score formula

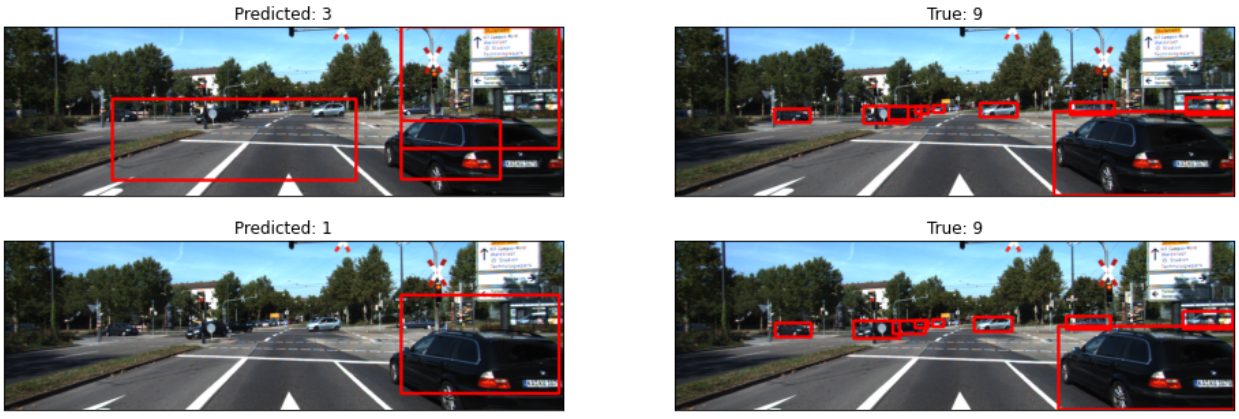

Figure 12: Localize Example 2

Figure 13: Localize Example 3

Figure 14: Localize Example 4

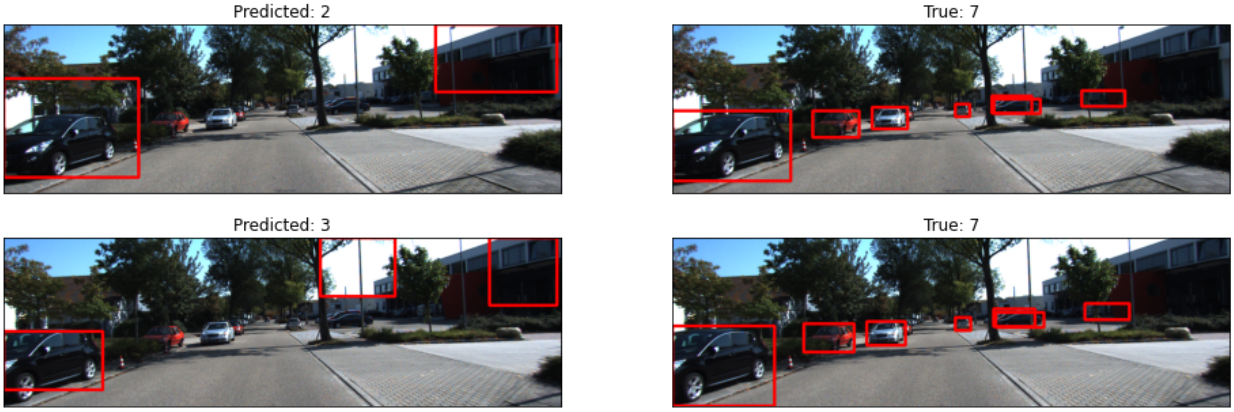

Figure 15: Localize Example 5