Traffic Object Detection

Josh Sawyer
jsawyer2@uoregon.edu

Nathan Pang
npang@uoregon.edu

Jake Follett
jfollett@uoregon.edu

Abstract

An attempt at building an R-CNN model, and a quick attempt at building a YOLO Object Detection Model for detecting pedestrians and vehicles.

Object Detection is a Computer Vision task with many solutions. In this paper we explored two of them to see what results we could get using the Penn-Fudan Pedestrian[1] dataset and a custom Vehicle Dataset from Kaggle[2].

The two models we tried re-creating were YOLO, and then RCNN using MobileNetV2 as a backbone for one iteration of RCNN to detect pedestrians, and our from scratch CNN models for both pedestrians and vehicles.

Of the two RCNN models we found that our from scratch CNN gave slightly better results than using MobileNetV2 on our pedestrian dataset, and our from scratch CNN for vehicles gets great results on images with the classes it trained on.

While the YOLO model failed to come to fruition..

1. Introduction

In the last 10 years there have been more autonomous vehicles, and smart transportation systems have shown technological improvements. The emergence of these 'smart vehicles' can be directly attributed to efficient and accurate traffic object detection models that can identify and localize various objects in real-time such as vehicles, pedestrians, and traffic signs.

Due to our limited computing power we worked on images rather than detecting the objects in a video. We mainly focused on building a Faster Region-Based Convolutional Neural Network (RCNN) model and also worked on building You Only Look Once (YOLO) model. These two are the two most popular and widely-used machine learning Models for traffic object detection.

RCNN combines the power of a CNN to classify images with region proposals. It works by first generating a set of potential object bounding box proposals using a non ML algorithm or a separate region proposal network (RPN) in the case of Faster RCNN.

These proposals are then fed into a CNN, which extracts features from the proposed regions and classifies them into object classes.

On the other hand, the YOLO model is a single-stage object detection technique that processes images in a single pass, making it significantly faster than RCNN. Compared to RCNN, YOLO divides the input image into a grid and assigns each grid cell the responsibility of predicting a fixed number of bounding boxes and class probabilities. This model allows to have real-time performance without sacrificing accuracy, making it suitable for applications requiring low latency like autonomous vehicles.[6]

2. Details of Approach

Our Original plan was to implement Faster RCNN from scratch - this means training the backbone network for the detection network and the backbone network for the Region Proposal Network (RPN) and finally setting up the head for the Faster RCNN model.

To start we did some Data Visualization to see what kind of data we were working with for the PennFudan Pedestrian Dataset that meant writing a function to draw bounding boxes according to some coordinates aligned with a given convention.

Pseudocode for converting between Bounding Box conventions:

```
//convert from (upper-left,lower-right)
//to (center, width, height)

def corner_to_center(boxes):
      x1,y1 = boxes[x1],boxes[y1]
      x2,y2 = boxes[x2],boxes[y2]
      centerx = (x1+x2)/2
      centery = (y1+y2)/2
      width = x2-x1
      height = y2-y1
```

```
      boxes = stack(centerx,
                    centery,
                    width,
                    height)
      return boxes


//convert from (center,width,height)
//to (upper-left,lower-right)

def corner_to_center(boxes):
      centerx,centery
            = boxes[centerx],
              boxes[centery]

      width,height
            = boxes[width],
              boxes[height]
      x1 = centerx - 0.5 * width
      y1 = centery - 0.5 * height
      x2 = centerx + 0.5 * width
      y2 = centery + 0.5 * height
      boxes = stack(x1,y1,x2,y2)
      return boxes
```

Pseudocode for drawing Bounding Boxes on top of images:

```
//draw bounding boxes ontop of a
//subplot with multiple axes
def bbox_to_rect(bbox,color,axs,x,y):
// using corners convention
return axs[x][y].add_patch(Rectangle(
 xy=(bbox[0],
 bbox[1]),
 width=bbox[2]-bbox[0],
 height=bbox[3]-bbox[1],
 fill=False,
 edgecolor=color,
 linewidth=2)
)
```

These functions worked for all of our datasets assuming we followed their annotation conventions the following figures are the results of these functions

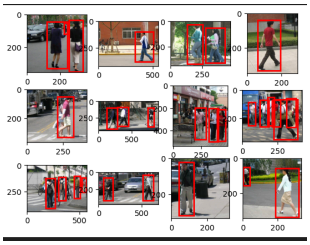

Figure 2.0 - Penn Fudan Data Vis

Figure 2.1 - Vehicle Visualization

We then trained some classifiers which we would use as our backbone for the detection network in our Faster RCNN model.

This called for making another dataset as we were going to use the PennFudanPed[1] and Vehicle dataset[2] mostly for the complete RCNN and we came to the conclusion that training our backbone network on what is essentially our validation set would be a bad idea.

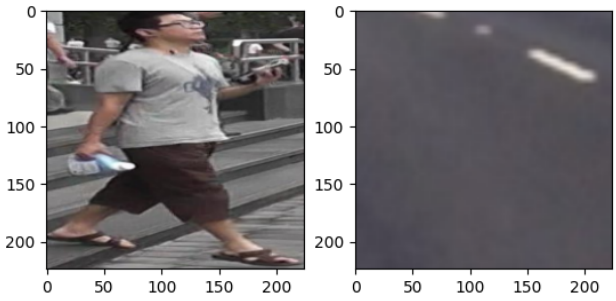

Figure 2.2 - Augmented dataset for backbone pretraining

The next step in our plan was training a CNN to be the backbone for our Region Proposal Network. This went smoothly as we had already set up training datasets by this point and had trained CNNs (See Section 3.1 and 4.1).

The last step would be to put it all together in the Faster RCNN architecture. This was the part of the project that proved to be far too time consuming by week 8 which caused us to pivot to using torchvision's FasterRCNN architecture[9] and simply slotting in our backbones rather than writing a Faster RCNN model and a training function for it.

2.1 Setbacks and Solutions:

Most of our time in the beginning was spent wrangling our Datasets and finding out how to load them into our models such that it would be able to train on bounding box data.

This would prove to be the most difficult step and the most time consuming aside from training the models. Adding insult to injury we found a custom dataset class for the PennFudanPed Dataset from a pytorch finetuning tutorial[4] which we went on to use extensively. For our Vehicle Dataset we also found a tutorial[5] on how to load in its data halfway into creating our own.

The main problem we ran into was implementing the multistage training which Faster RCNN requires. Specifically training the Detection network and RPN at the same time.

Other problems we had while trying to implement the Faster RCNN architecture was the Region of Interest Pooling (ROI Pooling) creating shape errors.

All of this combined led us to our decision to abandon creating the RCNN architecture from scratch and move to fine tuning the torchvision model with our own backbone and comparing it to a pre-trained backbone.

3. RCNN Results

There are two parts to ensuring we got good results with detecting objects. We split this up into training our own CNN to extract feature maps and then after training the CNN, we stripped the linear layer and used the convolutional layer of it as the backbone to the RCNN model.

We trained two different models, one for detecting people, and the other for detecting vehicles in traffic and both models got great results in the type of images we wanted to be able to detect objects in.

3.1 People CNN Results

The CNN model we trained was trained on a custom dataset using the "non-vehicle" images from the Vehicle Detection Image Set[7] to help the model distinguish between what is and isn't a person. The images of people was gathered from the PRW (Person Re-identification in the Wild) Dataset[8]. Once processed and cropped down, we had a total of 2100 random images, and 2057 images for people.

Training for this model didn't take long as we only had to train it for 10 epochs to get 98% accuracy on our test data. Due to the lack of complexity with this problem, not much was needed to be done to get this result. With our previous experience working with multi-class classification, doing a single classification with a CNN model made getting good results fairly easy. Training accuracy, compared with validation accuracy can be seen in figure 3.0.

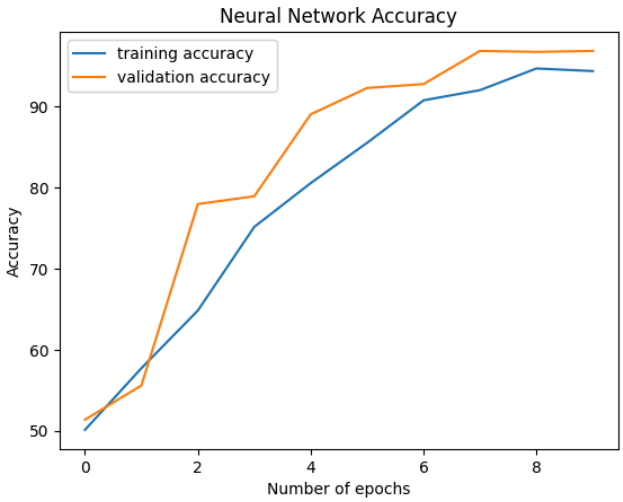

Figure 3.0

3.2 Vehicle CNN Results

The CNN was trained on the Traffic Vehicles Object Detection dataset[2] which contained 278 test images, 738 train images, and 185 validation images. Due to the images having bounding boxes in them, the number of training images turned out to be around 3000.

The final model we ended up with was quite efficient and was able to train up to 90% accuracy on the test data after only around 20 epochs (see Figure 3.1 for the accuracy and loss graph during training).

The train and validation curves in accuracy begin to separate after around epoch 6, where we can see the model begins to overfit on the training set. Despite this, validation accuracy continues to increase until around epoch 10 where it finds its local minimum.

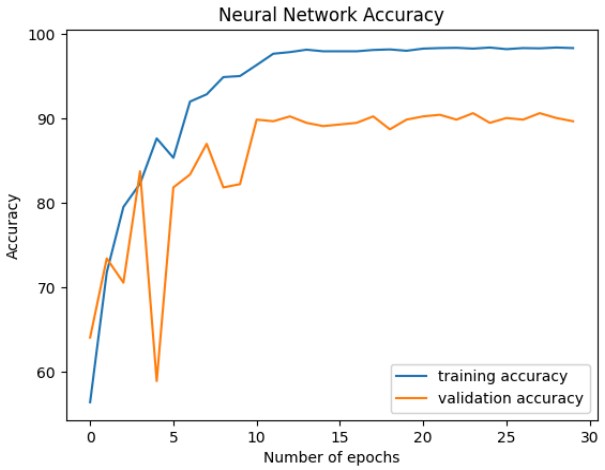

Figure 3.1 - Shows the accuracy of the network on the training and validation set. Validation begins to level out at around epoch 10.

3.3 Pedestrian/People Detection Results

We used two backbones to compare what results we got - our CNN we trained on pedestrians, and then a different CNN model, MobileNetV2[10] (referred to as MN2 for the remainder of the paper).

The results we got were great, with most pedestrians standing or walking getting in the 97% confidence range. Something important worth mentioning is that our model is also able to draw more bounding boxes (pick up more people) than were indicated in the annotation files. An example of this can be seen in figure 3.2 where the image on the left shows the bounding boxes given in the dataset, and the image on the right shows what our model picked up on.

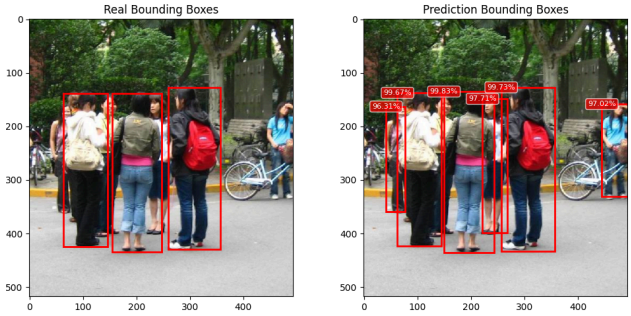

Figure 3.2

One of our main goals was to be able to pick up pedestrians in the street, so when given random images from the internet the model does a great job of getting high confidence bounding boxes of pedestrians walking or standing. A good example of this can be seen in figure 3.3 where all people close in the image are picked up with 98% confidence or higher.

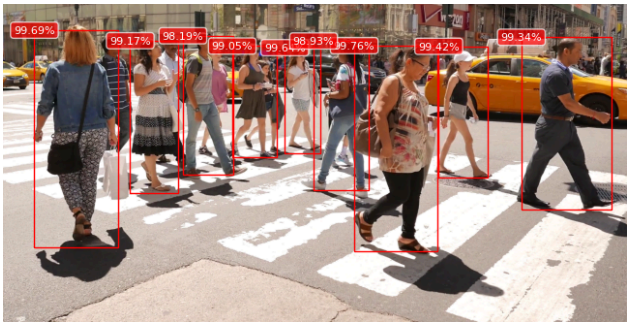

Figure 3.3

The results we got were pretty good for both, but what's important to note is that for the most part we got better results with our own CNN than with MN2 as the backbone. One instance of this can be seen in figure 3.4 where with MN2 as the backbone, a statue is picked up with a bounding box with 22% confidence (left), while with our CNN as the backbone it picks up this same statue but with only 13% confidence (right).

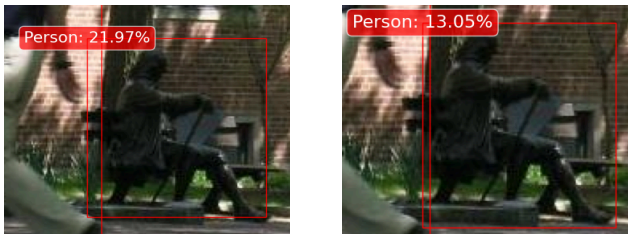

Figure 3.4

To double check that our model was actually doing a good job of ignoring "statues" and not just sitting people, we checked the results with an actual person sitting down and got good results, however, the confidence is significantly lower compared to people standing up or walking. This can be seen in figure 3.5 where MN2 as the backbone draws the box with 95.04% confidence (left), while our model as a backbone (right) draws it with 94.66% confidence.

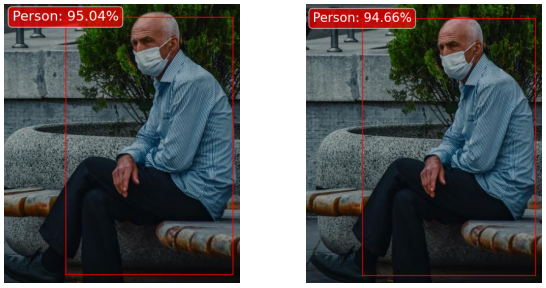

Figure 3.5

3.4 Vehicle Detection Results

Once we got our CNN trained, getting 90% accuracy on the test data, we were ready to use it as a backbone for the FasterRCNN model. The same dataset was used to train this model to help improve training to learn similar features. The results we got were consistent with the bounding boxes given in the annotation files for the dataset, and in some images, we got bounding boxes that not even that annotation files had labeled. An example of this can be seen in figure 3.6.

What's interesting about what our model does in figure 3.6 is that it's able to predict that the blurred image of the car is actually a car. The confidence is 90% which is pretty high, and this is a good sign that our model isn't overfitting in training.

Figure 3.6 - Top image shows the original image, and the bottom image shows our model's prediction.

Looking at our model's predictions on random traffic images off the internet we get very good results, even in crowded traffic as seen in figure 3.7. However, due to the dataset that the CNN was trained on, some classes (such as the license plates, trucks, and automobiles) aren't detected since they're more unique to a different region of the world, and aren't generalized to traffic in the US.

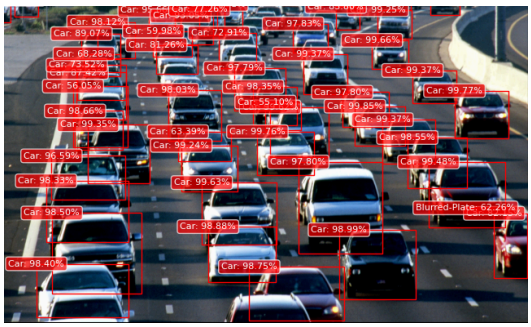

Figure 3.7

Something worth mentioning is that the model has a tough time on overhead images of traffic since it didn't see many of these types of images in training (see Figure 3.8).

We can also see in figure 3.8 that with some cars it does predict there's a lower confidence than we've seen in previous figures. Additionally, we can see that the bike isn't picked up on, even though it was trained to detect bikes as well.

Figure 3.8

Interestingly, although trained on images in a different region of the world, it is able to pick up on some classes, just with lower confidence. This can be seen in figure 3.9 where in the back, a UPS truck is labeled as "truck" with 53.82% confidence. Additionally, a "blurred" license plate is also labeled, despite the plate being a US plate and not the longer plates it was trained with. On the other hand, it does mislabel the van in the back as a "bus", but it makes sense as to why it does this since the van looks similar to some of the bus images that it trained on.

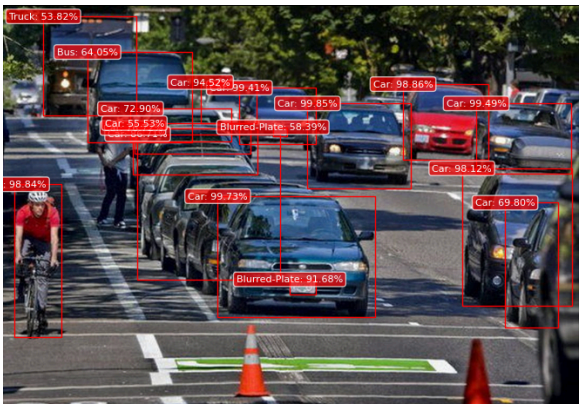

Figure 3.9

4. YOLO Model Results

For the YOLO model it was not a good result compared to the RCNN model. After having a high accuracy in the RCNN model, implementing the YOLO model seemed like a good decision so we could compare the two models. Due to implementing this model closer to the final report it did not achieve the output that we thought would have.

One of the failures that occurred was high loss numbers during training. Training the YOLO model involves minimizing the loss function. This has several factors to it one is localization loss another one is classification loss and last one is confidence loss. However when training the model the loss numbers were initially super high and it did not reduce as much as expected. Maybe this is due to the poor architectural design that was created.

Bounding boxes not drawn in the correct place.

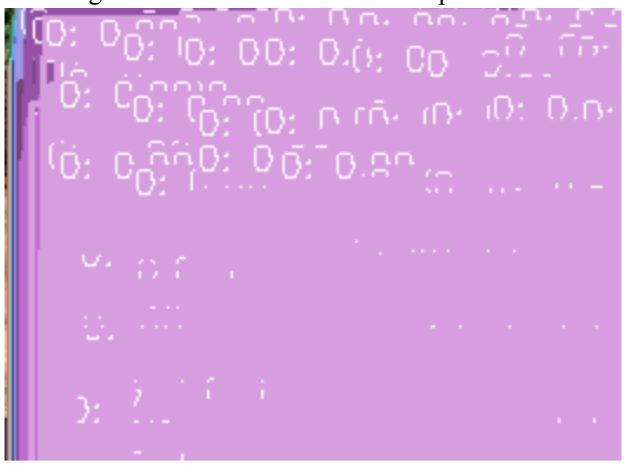

Figure 4.0

As you can see in Figure 4.0 there were too many bounding boxes that were created. The reason why there were lots of bounding boxes is because the threshold value was not created. In order to only detect the objects that have a high confidence score, implementing a threshold value was necessary. After this increased the threshold value to 0.5 but this did not draw any bounding boxes, due to the poor YOLO architecture. Reducing the threshold value to 0.05 would draw the bounding boxes in the images but it was not drawn around the objects that were supposed to have them as you can see in the Figure 4.1.

Figure 4.1

5. Discussion and Conclusions

Overall, we had good results with both our RCNN models being able to detect their respective 'objects'. The people model does a very good job of detecting pedestrians and in some places it's able to draw boxes around people that weren't even labeled in the test set's annotations file - which shows great results as it's able to generalize well. The same goes for the traffic model we trained to detect vehicles, in some instances it's able to detect new objects that weren't annotated in the test images which tells us that we got good training results and it's able to generalize. This is also further supported on random images from the internet where it's able to pick up on most vehicles.

Unfortunately, we were unable to get the results we wanted with the YOLO model. This is discussed further in 5.3.

5.1 People Model

In regards to the CNN portion of the model, we saw that this was able to generalize very well to data in only a short amount of training. I think a helping factor that played an important role in this was the normalization of the data. However, this isn't the main factor, and what really helped was a solid convolutional portion of the CNN that was able to extract the important features. The fact that there was only one class to classify (excluding the background images) made for fast training and good accuracy.

Knowing that we only had to classify people for this model, when it came to setting up the model we were able to get good results right away without the need for much adjustments needing to be made to the model.

In regards to the Faster RCNN portion we had much more trouble before we jumped ship to fine-tuning the

torchvisions model. The previous sections have covered our problems with recreating the architecture of Faster RCNN and creating its multistage training function .

As for troubles with the finetuning we didn't really run into any that wasn't par for the course . Surprisingly the hyper parameters didn't need very much tweaking either to get a good result.

5.2 Vehicle Model

As seen in the results section, we got very good results with the objects we were wanting to detect. There were several issues that came up, but these issues occurred due to the dataset we used not being diverse enough as it only had images of cars from one region of the world, that which wasn't from the US. Additionally, due to the angle of the training images being taken from, the model struggles a bit with being able to detect vehicles when in a birds-eye point of view. This isn't too much of an issue though as our goal was to detect traffic images while also being in traffic.

Referring back to the issue we had with the model not being able to identify other classes (mainly due to regional issues) we still had some success, just with lower confidence. Looking back at figure 3.5 we can see that the model was actually able to pick up on the UPS truck in the back and label it as truck. Although the confidence was only around 53% it's possible that with more training this confidence could be increased. However, the best way to increase the model's performance with classifying other vehicles in the US would be to get a more diverse dataset.

In regards to building the CNN to extract the features for the vehicle objects, there were some initial problems when building the architecture for the network. It seemed to have been too complex at first which resulted in no progress. But after a lot of small changes and updating the architecture of the model we came across a fast training model with high accuracy. Initially, to combat overfitting we used higher dropout rates but this actually resulted in the CNN having a harder time in training and ultimately getting less accuracy. Seeing this, we reduced the dropout rate and eventually came across a working model with high accuracy.

5.3 YOLO Model

As it's written in the results section for the YOLO model[11][12] most of the results for this model were negative. The model had very high loss numbers when training and the bounding boxes were not accurately drawn around the correct objects in the images. There's a couple reasons as to why this happened.

1. The design choice of the architecture. Such as number of layers, filter size and activation functions. This could have been better implemented in order to achieve better results.

2. All the images in the training dataset had different image sizes. resizing the image when training and scaling them back to the original image could have caused it to draw the bounding boxes around the wrong objects.

These are the main reasons why this YOLO architecture may not have had good results. In order to make this model work, it is better to start working on with the pre-trained model rather than building a YOLO model from scratch to better understand the already working model better and apply it to the original model.

6. Individual Contribution

Nathan Pang

Most of my time went to working on the Region Proposal Network architecture and combining it with the Detection Network (One of Josh's CNNs) to try and create a Faster RCNN model (which eventually got scrapped). This included writing the Region Proposal Model and trying to write a multistage training function and running out of time.

Once the from scratch Faster RCNN was scrapped I set up the FasterRCNN finetuning model which I did by following this tutorial: (pytorch.org/tutorials/intermediate/torchvision_tutorial.html)

This also involved me making the PennFudanPed dataset class as well as training the model with a pretrained backbone (mobilenet v2).

I also spent time setting up the data visualization for bounding boxes so that we could see how good our models were rather than looking at some coordinates and checking for large differences.

Josh Sawyer

A lot of my time went into working with the CNN models. This included me loading the datasets (building the dataset classes for the CNN training) and also editing them so they would work better for training.

For the pedestrian CNN I loaded the PRW dataset[8] which I processed and cropped all images to individual people (using the annotation file), and also included "random/background" images from a different dataset from kaggle[7]. I then trained a CNN model on these to 98% accuracy and used it as a backbone for the RCNN and trained that.

I additionally trained a second CNN on the Vehicle Dataset from Kaggle[2]. I had to process this data as well so I wrote a method to crop all the images based on the information provided by the annotation files, and then trained a CNN on those images and their respective classes. After training, I ended up getting a model with around 90% accuracy on the test set and then used this as a backbone for the RCNN.

Jake Follett

A lot of my time went into working on creating the YOLO model and debugging the errors messages that I have encountered. I also organizing the dataset to better fit our model and get good result.

For the YOLO model I have tried to implement YOLOv3 model[11] but this model was a little too complicated to create from scratch and there were too many errors that I was not able to figure our so I switched to YOLOv1 model[12] where the model is less complicated. In this model I was able to detect some objects but it was not accurate and I had a very high loss number when training.

The other thing I worked on was organizing the dataset to pass into our CNN model. I have worked with Josh on cropping all the objects in an image from the Traffic Vehicles Object Detection dataset[2] and save it to another folder by using the labels where all the bounding box information are located. This image of each object would then be used for the CNN model to classify images.

7. References

[1] Li, F., & Yu, Y. (2007) "Penn-Fudan Database for Pedestrian Detection and Segmentation" https://www.cis.upenn.edu/~jshi/ped_html/

[2] "Traffic Vehicles Object Detection" Traffic vehicles Object Detection | Kaggle

[3] https://github.com/pytorch/vision/tree/main/references/detection

[4] https://pytorch.org/tutorials/intermediate/torchvision_tutorial.html

[5] https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

[6] https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e

[7] Vehicle Detection Image Set | Kaggle

[8] http://zheng-lab.cecs.anu.edu.au/Project/project_prw.html

[9] https://pytorch.org/vision/main/models/faster_rcnn.html

[10] arXiv:1801.04381 [cs.CV]

[11] https://sannaperzon.medium.com/yolov3-implementation-with-training-setup-from-scratch-30ecb9751cb0

[12] https://www.youtube.com/watch?v=n9_XyCGr-MI&t=1365s