A comparative study of efficiency between 2 Computer Vision algorithms

Research Question: *To what extent can Fast Region-Based Convolutional Neural Network and You Only Look Once algorithms be used to detect objects in a moving car at various times during the day?*

Subject: *Computer Science*

Word Count: 3861

Table of Contents

Introduction

Vision is one of the 5 fundamental human senses and is essential for the human brain to map their surroundings with extreme precision and identify and locate various objects. As the world has become more technologically advanced, computers have become increasingly capable of performing tasks which are commonplace for humans, such as mathematics, predictions and industrial work. Many computer features are now being considered substitutes for parts of the human body, for example a microphone as ears, a power supply as a heart and a CPU replacing a brain. Vision is generally attributed to be attained from cameras – like mobile cameras and video cameras. Computer vision is essentially defined as a field of artificial intelligence (AI) that enables computers and systems to derive meaningful information from digital images, videos and other visual inputs — and take actions or make recommendations based on that information (IBM).

Computer vision encompasses a plethora of different features and algorithms, all of which have their own unique purposes. However, all of them operate on the same basis – discerning objects after being fed in lots of data and identifying patterns in certain images to pinpoint objects (D, Prince). The technology behind this can be mainly divided into 2 aspects:

- Deep learning, which uses machine learning algorithms to permit computers to differentiate images from one another while large amounts of data are being fed in to be processed (Dertat)
- A type of Neural Network - a series of algorithms that endeavors to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates (Chen, 2021) - called a Convolutional Neural network. This

divides the image up into various pixels and identifies image features like hard edges and simple shapes to segregate objects

There are numerous tasks that computer vision can accomplish through these programming facets, all of which are very useful in today's world. They include – object tracking, which locates an object and follows its motion for a period of time; content-based object retrieval that finds images on large data stores based on a keyword; image classification that matches a given part of an image to an object; and object detection which does the opposite of image classification and matches the object to the image.

Even within the realm of object detection, there is present more categories of computer vision. Single object localization isolates a single appearance of a certain object within an image, such as one airplane which is parked at an airport. Multiple object localization ascertains the location of one or more of the same object – using the same example, all the airplanes present at the airport, and draws a bounding box around the object to localize them. Object detection identifies all the objects, and draws the bounding box on all the different types of objects.

Multiple applications of computer vision are evident in the world nowadays, in multiple industries, some of which include:

- Object localization and tracking can be used to locate fugitives in manhunts for the criminology industry
- The use of content based image retrieval in most popular search engines such as Google, Yahoo and Bing

- Object detection is present in Google Translate features where one can point their phone at wording in a language and obtain the translation of those words in another language
- Object localization can be found in television broadcasts for sports as the ball is located and the camera tracks it in object tracking
- Autonomous vehicles use these algorithms to act as proximity warnings as well as identification of which object is present (Lewis), as will be explored here

Several object detection algorithms exist, one of which is Fast Region-Based Convolutional Neural Network (Fast R-CNN). CNN, as aforementioned, is a type of neural network that divides the image into pixel-based regions and uses common and simple features to perform the task of object location. Fast R-CNN is similar, however is faster and more efficient as it takes in one image instead of 2000 region proposals – and its speed is evident from Figure 1.

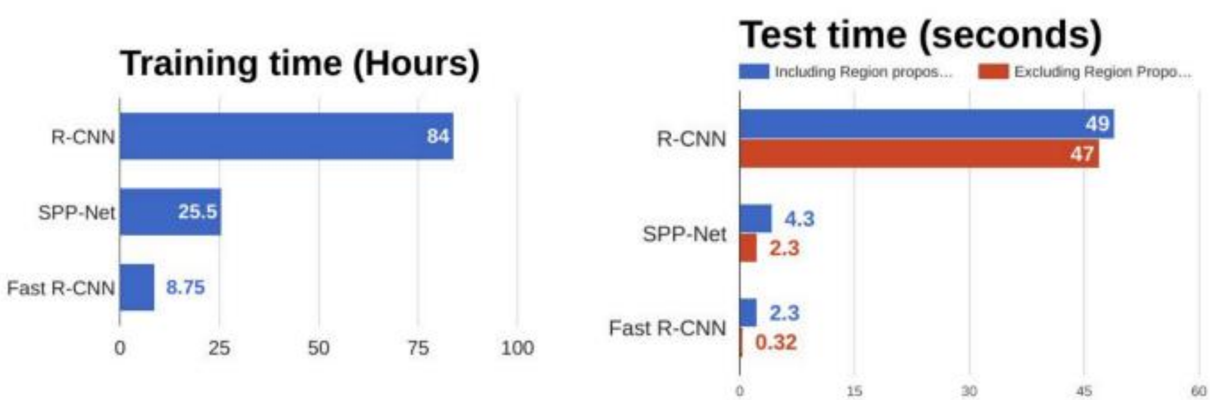


Figure 1 – Training time and test time of CNN algorithms (Gandhi, 2018)

Another algorithm is 'You Only Look Once' (YOLO) which operates differently to the CNNs. It functions by dividing the image into a N by N grid – N being a variable which differs based on the resolution of the image. Then, each square in the grid is associated with a class probability and offset value for each object and using those values the various objects are mapped. Although YOLO algorithms are significantly faster than others, they do struggle with identifying small objects as they encompass a tiny area in the grid.

Thus, this paper explores research conducted into the question - To what extent can Fast R-CNN and You Only Look Once algorithms be used to detect objects in a moving car at various times during the day? To delve further into this topic, multiple trials of data using multiple objects were collected from a moving car, and inputted into these two algorithms; following which the time taken for the object to be found, and the accuracy with which the object were used to evaluate the data.

Data Collection

2A) The Scenario

Autonomous vehicles are developments which are revolutionizing the transport industry, as they eliminate all human error experienced by drivers, and can also easily optimize routes and speeds. Object detection algorithms are used to not only improve safety by acting as proximity warnings, but also minimize travel time by identifying various objects on the road and thus deciding which lane to travel on. For this investigation, majority of the training photos were obtained from the company 'Waymo' – a subsidiary of Google – which operates in Phoenix, Arizona and released most of their data to public for research purposes. These training images will be those of the taxis behind pedestrians, other cars etc.

The verification videos will be taken by myself with the help of a second person driving a car at cruise control, at a constant velocity towards an object autonomous vehicles would commonly encounter. These videos will then be fed into the same algorithms that encountered training images to observe time and efficiency of the same. The algorithms to be used are the aforementioned Fast R-CNN and YOLO programs on Python.

2B) Methodology

2B.1) Hypothesis

If the size of the object is larger, then YOLO will be a faster algorithm whereas Fast R-CNN will be more efficient for smaller objects because YOLO uses bounding boxes and predictions which are less useful for small objects

2B.2) Training the algorithm

1) Firstly, online data libraries were used to obtain basic Python codes for both Fast R-CNN and YOLO algorithms to act as a framework
2) Then, the codes were modified to fit the required purpose by adding a module which notifies the user of the time taken for the program after debugging

Figure 2 – A segment of the Waymo Open Data Set (Etherington, 2019)

3) The Waymo Open Data Set was opened to save a portion of the 1,000 20-second driving segments available
4) Following that,the segments were run through each of the algorithms to train them to identify different common objects, such as the example shown in Figure 2

2B.3) Collecting the validation data

1) To begin, a narrow road with minimal traffic and a clear path was identified
2) At 7am, 2 horizontal lines were marked out that were 6.3 metres apart
3) Then, 2 people set up a Jeep Wrangler 2-door to cruise control at 5 km/h far behind the first horizontal line
4) The second person set up an iPhone 12 max in the car to record, and pressed record as the car started moving
5) The first person moved along the second horizontal line as the car moved forwards, and the second person stopped recording as the car reached the first horizontal line.
6) Steps 4 and 5 were repeated 9 more times and the average of the 10 trials was taken for further calculations
7) The cruise control setting was then changed to 10 km/h, and steps 4 to 6 were repeated for increments of 5 km/h until 30 km/h, as well as for a stationary car
8) Then, the human was substituted for other objects in the vicinity which cars might encounter, such as a boomgate and a lamppost
9) The sizes and shapes of these objects were measured before data was collected
10) These other objects only had 2 samples – one with a stationary Jeep and one with a speed of 30 km/h for the Jeep
11) As with the human, 10 trials were conducted for the other objects and the average was used
12) This data collected formed ‘Case I’ while identical data was collected at 5:30 pm for ‘Case II’ in the evening and at 11pm for ‘Case III’ at night

13) Lastly, data was also collected at a signal with a stationary car as multiple cars and pedestrians crossed the road to allow for detection of a large number of objects (‘Case IV’)

2B.4) Analyzing the data

1) Each sample from each Case was put onto both of the algorithms that were previously trained and noted down the time and efficiency for each one
2) In doing so, A map of all the objects in the image was obtained (Appendices D and E)
3) Then, code was added that allowed for objects to be classified and only one object to be detected
4) Finally, the data of speed against average time taken was graphed and trends were identified to analyze some of the key features and formed conclusions

2C) The Algorithms

2C.1) Fast Region-Based Convolutional Neural Network

Image Training Process:

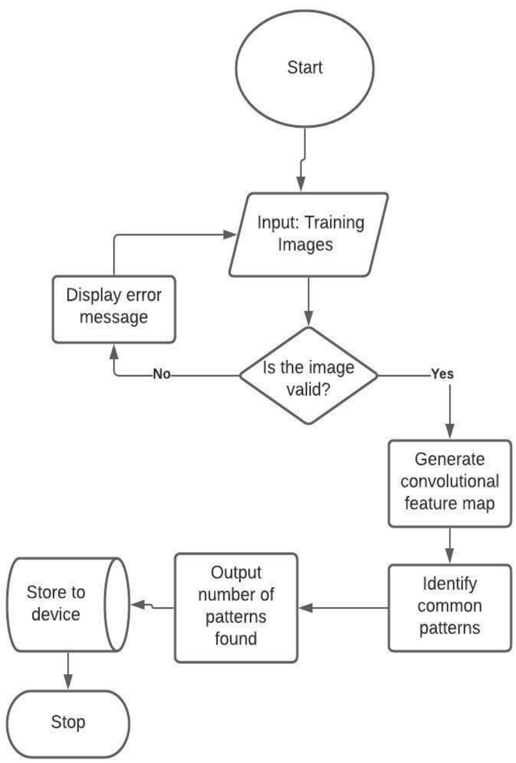


Figure 3 – Fast R-CNN Part 1

Image verification process:

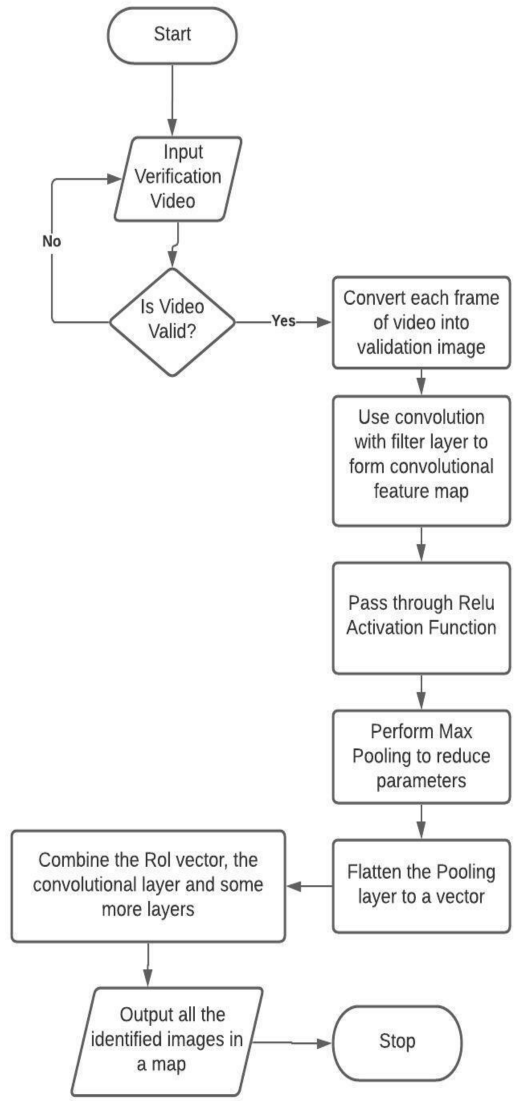


Figure 4 – Fast R-CNN Part 2

Convolution is the process by which the inputted convolutional layer (image) is merged with a filter obtained from the training images to form a convolutional feature map which is easier to interpret (Khandelwal, 2020). Both the layer and the filter are visualized as grids – the layer as M by M and the filter as N by N, wherein the value of N is smaller for the filter (aka kernel) and the grids are filled up with binary 0s and 1s. Convolution is performed by sliding the filter over N^2 grid slots in the layer – the 4 corners, some points on each edge and inside them, and the center. In doing so, the corresponding grid slot of the filter and the layer are multiplied and the sum of all the multiples is considered for the convolutional feature map. The distance between two operations of the process is called the 'stride'.

An example is a 5x5 layer and a 3x3 filter, as shown

1	0	0	1	1
0	1	0	0	0
0	1	1	1	1
1	0	1	1	0
0	1	1	0	0

0	1	0
1	1	0
1	0	1

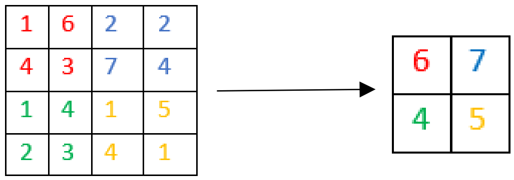
Considering the 1 in the top left, convolution would be as follows:

0x1	1x0	0x0
1x0	1x1	0x0
1x0	0x1	1x1

⇨ 2

After convolution takes place, the Feature Map is passed through the Rectified Linear Unit **(ReLu) Activation Function** = R(z), wherein if z is less than 0, R(z)=0 and if z is greater than 0, R(z) = z, thus allowing for non-linearity and improving efficiency.

Following this, the process of **Max Pooling** is performed in which the feature map is fed in, with hypothetical proportions N by N by H – and max pooling considers a pooling window of 4 blocks each and extracts the highest value from the window onto the new layer – thus forming a layer of proportions $\frac{N}{2}$ by $\frac{N}{2}$ by H. An example is shown below:



The next process conducted is called **Flattening**, wherein the 3 numbers after pooling ($\frac{N}{2}$ by $\frac{N}{2}$ by H) are turned into a 1-D vector so that further processing can occur

Lastly, the Flattened pooling layer, convolutional layer and some others are superposed on one another and the numbers obtained are matched against the ones from the training images.

If any patterns of numbers are then noticed by the AI, then the object is detected.

<u>2C.2) You Only Look Once:</u>

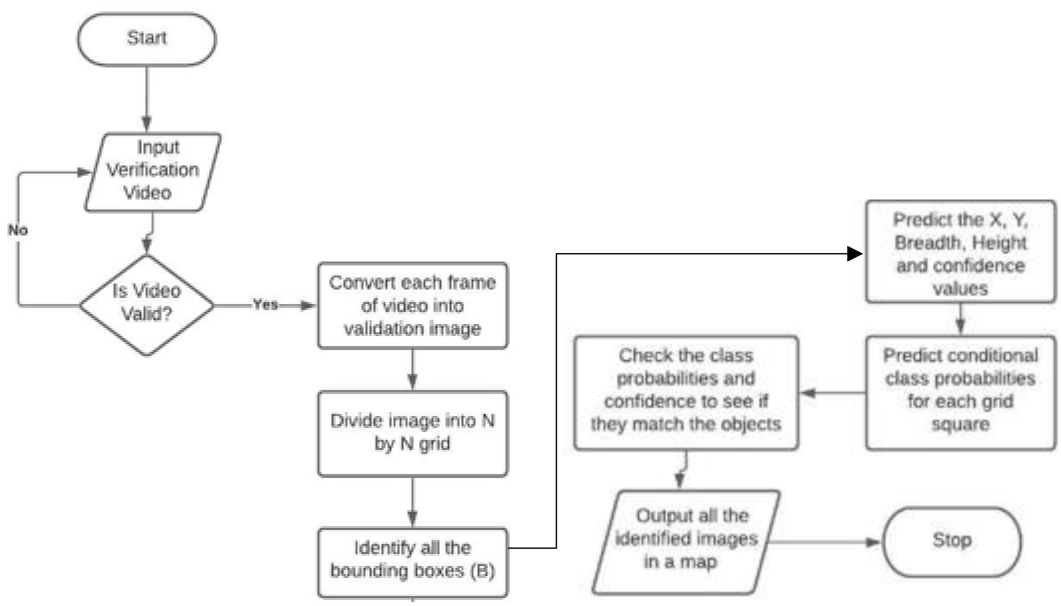


Figure 5 = YOLO algorithm

The first part of the YOLO process is dividing the image into **an N-by-N grid** of boxes, as shown to the right. Generally, if the center of an object is within one of the boxes, then that box is responsible for detecting that object. Following that, each grid has a certain number of **bounding boxes** associated with it and those bounding boxes are the ones used to check if an object is present.

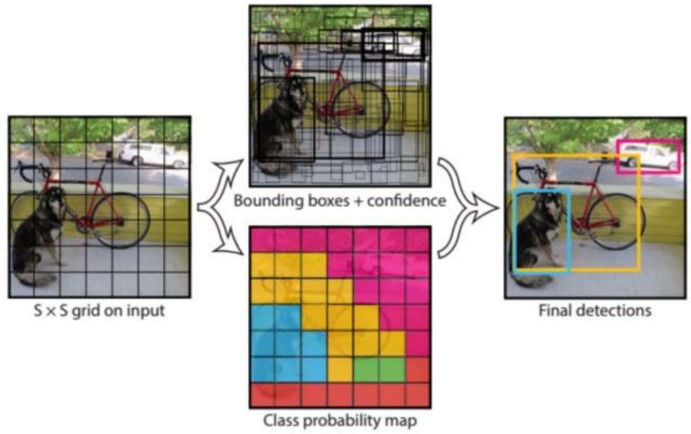


Figure 6 = Process of YOLO algorithm (Chablani,2017)

Following that, each bounding box is used to perform 5 predictions:

- (X,Y) = the coordinates of the center of the object
- (W,H) = The width and height of the object relative to the whole image
- Confidence = The probability the object is in that position

The confidence can be expressed mathematically as:

$$C(object) = P(object) \times IOU$$

Wherein IOU is the ratio of the intersection to the union between the two data sets of the predicted object and the ground truth (info from training images). If the object is certainly in the bounding box, then the confidence value will equal IOU

Another set of probabilities is calculated – The class specific probability for each object (which distinguishes one type of object from another), calculated conditionally. One set is calculated for each grid box:

$$P(Class|Object) = \frac{P(Class \cap Object)}{P(Object)}$$

$$P(Class|Object) \times C(Object) = P(Class) \times IOU$$

The maximum value of the second equation will yield the required output as it encodes the chances of the object being in the box, as well as how well the object fits in the bounding box (Gandhi, 2018). Thus, the objects can then be detected.

Data

3A) Detecting a human:

These tables display the results of the time taken for the python code to detect a human, for each speed in each case. This is taken from the image of the last frame moving at that particular speed, and the time values shown here were obtained from the python time() module

Table 1 = Fast R-CNN algorithm

Case I = 7 am		Case II = 5:30 pm		Case III = 11 pm		Case IV = Traffic signal	
Speed (km/h)	Average time taken (s)	Speed (km/h)	Average time taken (s)	Speed (km/h)	Average time taken (s)	Speed (km/h)	Average time taken (s)
0	0.388	0	0.299	0	0.202	0	0.150
5	0.400	5	0.308	5	0.210	30	0.176
10	0.405	10	0.312	10	0.215		
15	0.411	15	0.313	15	0.222		
20	0.417	20	0.317	20	0.229		
25	0.420	25	0.322	25	0.234		
30	0.427	30	0.334	30	0.236		

Table 2 = YOLO algorithm

Case I = 7 am		Case II = 5:30 pm		Case III = 11 pm		Case IV = Traffic signal	
Speed (km/h)	Average time taken (s)	Speed (km/h)	Average time taken (s)	Speed (km/h)	Time taken (s)	Average time taken (s)	Average time taken (s)
0	0.340	0	0.255	0	0.175	0	0.130
5	0.345	5	0.260	5	0.177	30	0.155
10	0.349	10	0.265	10	0.181		
15	0.353	15	0.269	15	0.184		
20	0.358	20	0.274	20	0.186		
25	0.362	25	0.280	25	0.190		
30	0.366	30	0.284	30	0.193		

These graphs show how the time taken varies for Fast R-CNN and YOLO algorithms for the purpose of detecting a human at various times in the day

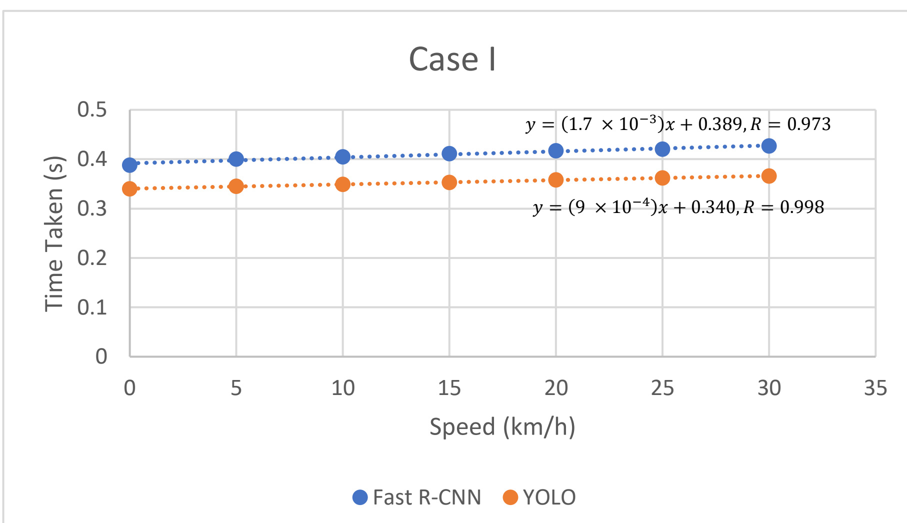


Figure 7 – Graph showing speed vs average time taken for both algorithms to detect a human at 7am

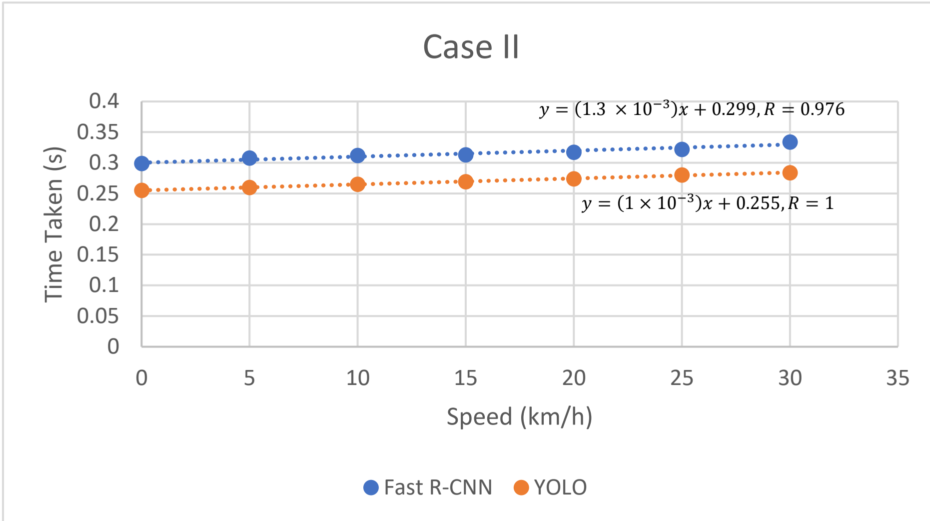


Figure 8 – Graph showing speed vs average time taken for both algorithms to detect a human at 5:30pm

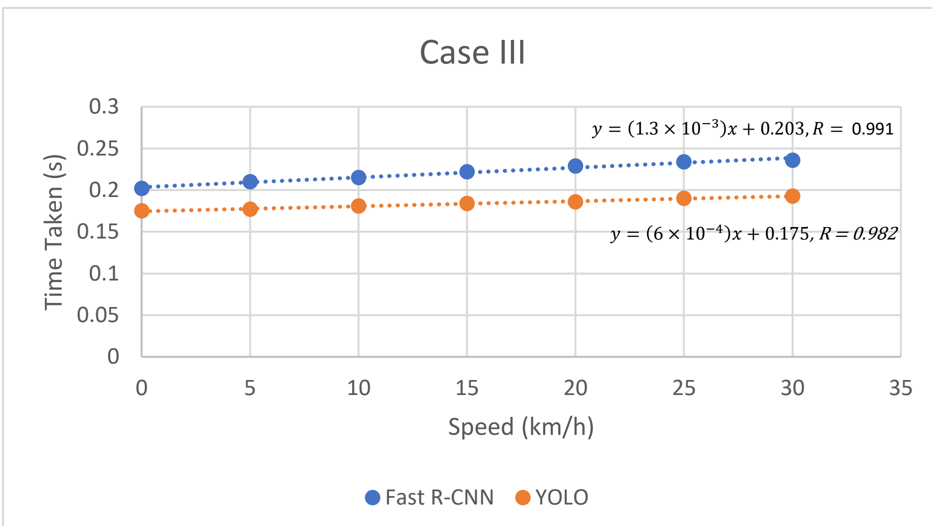


Figure 8 – Graph showing speed vs average time taken for both algorithms to detect a human at 5:30pm

Analysis:

1. When the object to be detected is a human, the YOLO algorithm is faster than the Fast R-CNN at all speeds, and at all times of the day. In Case I, the range of times for YOLO is from 0.340 to 0.366 seconds, while for Fast R-CNN the range is 0.388 to 0.427. In Case II, YOLO ranges from 0.255 – 0.284 and Fast R-CNN ranges

from 0.299 – 0.334. Case III YOLO is from 0.175 to 0.193 while Fast R-CNN is 0.202 to 0.236. For all 3 cases, the highest time taken value for YOLO is lower than the lowest value for Fast R-CNN. This is due to the fact that the bounding boxes in YOLO can predict a large human, encompassing most of the image with high confidence in a shorter period of time than it takes Fast R-CNN to perform convolution, pooling and the rest of the process.

2. The algorithms become less effective, and take more time as the speed of the car increases. At 7:30 am, when the speed of the car is increased by 5 km/h, YOLO takes 4.5 milliseconds more and Fast R-CNN takes 8.5 milliseconds more. At 5:30 pm, with the same increase in speed, YOLO – 5 ms more and Fast R-CNN – 6.5 ms more; while at 11pm there is a similar increase for both. At higher speeds, the camera attached to the car will be increasingly unstable and the borders of the object will be blurrier – which is hard for the algorithm to classify as a human, causing this increase in time. However, autonomous vehicles frequently travel at high speeds, and the car used (Jeep Wrangler) is notoriously unreliable, meaning that a quicker algorithm (YOLO) would be preferred
3. The times reduce from Case I to Case II to Case III. Case I has an average time taken of 0.232 seconds over the 140 trials; while Case II's average is 0.292 seconds and Case III's average is 0.202 seconds. This is because the trials were conducted with the car travelling eastward, thus in the morning when the sun is rising in the east the shadow of the building behind the human was acting as camouflage and making it hard to see, while that did not occur in the evening while

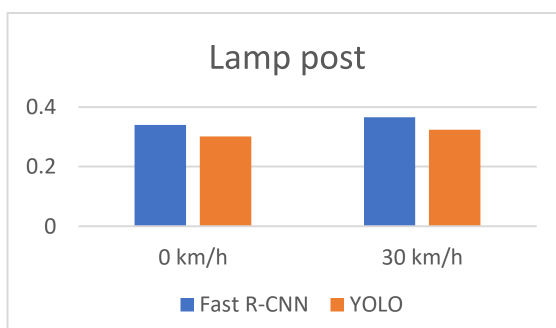
the sun was setting behind the car. The entire road was illuminated artificially in the night, which made detecting the easiest

4. The time taken for Case IV is significantly less than the other cases, for both algorithms. The average time taken here is 0.163 seconds for Fast R-CNN and 0.143 seconds for YOLO. At a signal, there are multiple people walking across and as a result the algorithms only need to search through a small portion of the image to detect a human, hence both algorithms are under 200 milliseconds in detection.
5. The data was quite precise as there were lower uncertainties, since in the raw data for each speed in each algorithm the percentage uncertainty never exceeded 10%. This is a result of the cruise control being used to keep the car's speed constant and the algorithms being well trained. The results were also accurate as they matched what was said in the Waymo open set that YOLO was more usable than Fast R-CNN.

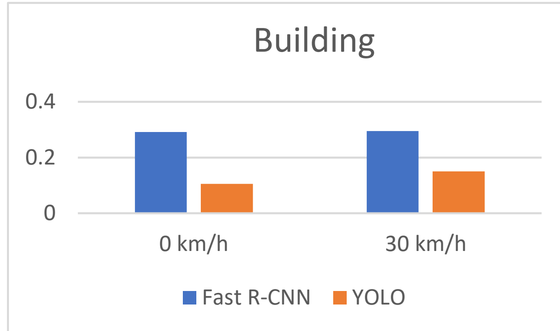
3B) Detecting a non-human object:

Throughout the data collection process, data for multiple stationary objects was also collected at speeds of 0 and 30 km/h – at 5:30 pm. These objects were of varying sizes and shapes, and the results from both algorithms are listed below

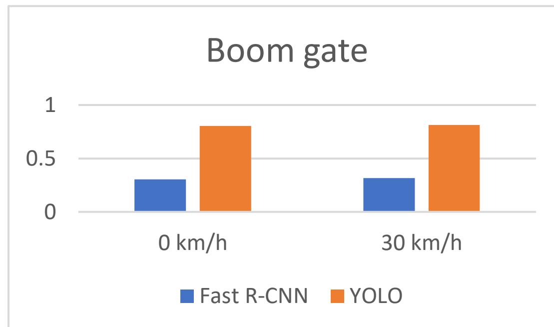
Object = Lamppost		
Speed (km/h)	Time taken (s)	
	Fast R-CNN	YOLO
0	0.340	0.301
30	0.366	0.324



Object = Building		
Speed (km/h)	Time taken (s)	
	Fast R-CNN	YOLO
0	0.291	0.105
30	0.295	0.150



Object = Boomgate		
Speed (km/h)	Time taken (s)	
	Fast R-CNN	YOLO
0	0.304	0.805
30	0.315	0.812



Object = Rock		
Speed (km/h)	Time taken (s)	
	Fast R-CNN	YOLO
0	0.355	N/A
30	0.370	N/A

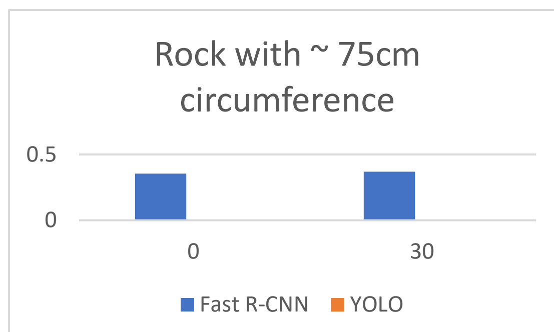


Figure 9, 10, 11,12 – Graphs for non-human object detection

<u>Analysis:</u>

1. The larger the object, the easier it is for YOLO to detect, while the size of the object doesn't affect the Fast R-CNN algorithm. For example, the building has an area of approximately 1500 square metres in the camera while the boomgate has an area of 2.5 square metres. The average time taken for Fast R-CNN is 0.293 for the

building and 0.309 for the boomgate while for YOLO the building was detected in approximately 0.128 seconds and the boomgate in around 0.808 seconds. This is a result of the fact that the YOLO algorithm forms bounding boxes and predicts values for width and height – the lower the width and height, the less confident it is.

2. YOLO algorithms are extremely inefficient with small objects. The rock, which has a small circumference, was detected within 400 milliseconds by the Fast R-CNN algorithm but was unable to be detected by the YOLO algorithm. In real life use this will be costly as small debris on the road might damage the car if it can't be detected and avoided quickly enough.
3. The results match the hypothesis, as the YOLO algorithm was quicker to detect and classify larger objects, such as humans and the building whereas it took nearly a second to detect a small boomgate and failed to detect a rock. On the other hand, Fast R-CNN was consistent and detected all objects at an equal time.

Conclusion

The research question which was investigated - 'To what extent can Fast R-CNN and You Only Look Once algorithms be used to detect objects in a moving car at various times during the day?' – was experimented on at various speeds for possible use in autonomous vehicles. The general use of object detection algorithms is to first identify objects in a given image, while pinpointing their locations; and then classify the object into a particular type. Doing so in a car, especially a driverless one, would be crucial not only for safety as the car would avoid damage-causing collisions, but also for optimizing travel time as the car can detect speed limit signs and cars ahead. Deciding which algorithm is the most advantageous, as was looked at here, can save the R&D company a substantial amount of money while also making the cars even safer and even faster.

Although both of the algorithms detected large objects quickly with near-perfect efficiency, there were some areas where one algorithm was better than the other. As stated in the data analysis, the YOLO algorithm was significantly faster for larger objects, while it was slower and very inaccurate for smaller objects. From this, one can infer that the best possible option for companies with a larger starting income is to primarily install YOLO but also install Fast R-CNN while only setting it to detect small objects; while companies with less money should just use Fast R-CNN entirely. The algorithms also reduced in efficiency as speed increased, resulting in the car having less time to break from a higher speed. A conclusion to draw is that autonomous vehicles have to be car models with good brakes to maximize safety. The final conclusion to draw is that these cars should preferably avoid driving on roads towards the sun during the day – a feature which should be added into the pathfinding algorithm of these vehicles.

Despite the investigation process yielding data that was relatively accurate and precise, there were some limitations present, and improving them would have made the results close to perfect. Firstly, the python code module Time() which measured the start and the end time, didn't only measure the time for object detection and classification but also searching for the images, initializing variables etc. which was done more in the Fast R-CNN algorithm and could cause the difference between the two algorithms to be much larger than it actually is. This can be avoided by placing the starting and ending of the time at a different part in the code. The secondary sources used, additionally, had their own individual limitations (Appendix F) and different ones could have been selected to improve the research process Also, the human in the car, after setting the cruise control, would then have to immediately start the recording as the speed hits the desired value, which involves reaction time and cound cause some uncertainty Each trial took approximately 2-3 minutes and with 10 trials for each speed and 7 speeds in total, the time was not consistent. For example, Case I was from 7am till 9am and that caused the sun to rise and reduced the time for the higher values of speed. The same happened for Case II and cane be reduced by doing each speed on a different day. The number of objects passing through the camera for each trial at the signal varied, as the pedestrian traffic changed at every red light. That could have factored into high uncertainties for that data set, and can be reduced by setting people up to move across the signal.

These results, and this investigation, can be used in the transport industry, in autonomous vehicles as this research has proven that YOLO is better for larger objects whereas Fast R-CNN is better for smaller objects. Further research can be conducted to look at the minimum size of object YOLO algorithms can detect with 100% accuracy, and thus determine which locations it is best for these algorithms to run (for example, Fast R-CNN would be more efficient in a small town while YOLO would be better in a city). All in all, computer vision is a key aspect of machine learning for the future and can revolutionize how we travel.

Works Cited

Brownlee, Jason. "A Gentle Introduction to the Rectified Linear Unit (Relu)." *Machine Learning Mastery*, 20 Aug. 2020

Chablani, Manish. "YOLO – You Only Look Once, Real Time Object DETECTION EXPLAINED." *Medium*, Towards Data Science, 31 Aug. 2017

Chen, James. "Neural Network Definition." *Investopedia*, Investopedia, 19 May 2021,

Dertat, Arden. "Applied Deep Learning – Part 4: Convolutional Neural Networks." *Medium*, Towards Data Science, 13 Nov. 2017

D., Prince Simon J. *Computer Vision: Models, Learning, and Inference*. Cambridge University Press, 2012.

Choudhury, Ambika. "Top 8 Algorithms for Object Detection." *Analytics India Magazine*, 15 Feb. 2021

Etherington, Darrell. "Waymo Releases a Self-Driving Open Data Set for Free Use by the Research Community." *TechCrunch*, TechCrunch, 21 Aug. 2019,

Gandhi, Rohith. "R-CNN, Fast R-CNN, Faster R-CNN, YOLO – Object Detection Algorithms." *Medium*, Towards Data Science, 9 July 2018,

Garima13a. "Garima13a/YOLO-Object-Detection: YOLO Is a State-of-the-Art, Real-Time Object Detection Algorithm. In This Notebook, We Will Apply the YOLO Algorithm to Detect Objects in Images." *GitHub*

Khandelwal, Renu. “Convolutional Neural Network: Feature Map and Filter Visualization.” *Medium*, Towards Data Science, 18 May 2020,

Lewis, Gene. Stanford University, pp. 1–6, *Object Detection for Autonomous Vehicles*.

Olver, Peter J., and Allen Tannenbaum. *Mathematical Methods in Computer Vision*. Springer, 2003.

“What Is Computer Vision?” *IBM*, www.ibm.com/topics/computer-vision

Appendix

Appendix A = You Only Look Once .py files

A.1) Forms the grids and boxes, and performs operations with them

```
import time
import torch
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches


# starting time
start = time.time()


def boxes_iou(box1, box2):
    # Get the Width and Height of each bounding box
    width_box1 = box1[2]
    height_box1 = box1[3]
    width_box2 = box2[2]
    height_box2 = box2[3]

    # Calculate the area of the each bounding box
    area_box1 = width_box1 * height_box1
    area_box2 = width_box2 * height_box2

    # Find the vertical edges of the union of the two bounding boxes
    mx = min(box1[0] – width_box1 / 2.0, box2[0] – width_box2 / 2.0)
    Mx = max(box1[0] + width_box1 / 2.0, box2[0] + width_box2 / 2.0)

    # Calculate the width of the union of the two bounding boxes
    union_width = Mx – mx

    # Find the horizontal edges of the union of the two bounding boxes
    my = min(box1[1] – height_box1 / 2.0, box2[1] – height_box2 / 2.0)
    My = max(box1[1] + height_box1 / 2.0, box2[1] + height_box2 / 2.0)

    # Calculate the height of the union of the two bounding boxes
    union_height = My – my

    # Calculate the width and height of the area of intersection of the two bounding boxes
    intersection_width = width_box1 + width_box2 – union_width
    intersection_height = height_box1 + height_box2 – union_height

    # If the the boxes don't overlap then their IOU is zero
    if intersection_width <= 0 or intersection_height <= 0:
        return 0.0

    # Calculate the area of intersection of the two bounding boxes
    intersection_area = intersection_width * intersection_height

    # Calculate the area of the union of the two bounding boxes
    union_area = area_box1 + area_box2 – intersection_area

    # Calculate the IOU
    iou = intersection_area / union_area

    return iou


def nms(boxes, iou_thresh):
    # If there are no bounding boxes do nothing
    if len(boxes) == 0:
```

```
        return boxes

    # Create a PyTorch Tensor to keep track of the detection confidence
    # of each predicted bounding box
    det_confs = torch.zeros(len(boxes))

    # Get the detection confidence of each predicted bounding box
    for I in range(len(boxes)):
        det_confs[i] = boxes[i][4]

    # Sort the indices of the bounding boxes by detection confidence value in descending order.
    # We ignore the first returned element since we are only interested in the sorted indices
    _, sortIds = torch.sort(det_confs, descending=True)

    # Create an empty list to hold the best bounding boxes after
    # Non-Maximal Suppression (NMS) is performed
    best_boxes = []

    # Perform Non-Maximal Suppression
    for I in range(len(boxes)):

        # Get the bounding box with the highest detection confidence first
        box_i = boxes[sortIds[i]]

        # Check that the detection confidence is not zero
        if box_i[4] > 0:

            # Save the bounding box
            best_boxes.append(box_i)

            # Go through the rest of the bounding boxes in the list and calculate their IOU with
            # respect to the previous selected box_i.
            for j in range(I + 1, len(boxes)):
                box_j = boxes[sortIds[j]]

                # If the IOU of box_i and box_j is higher than the given IOU threshold set
                # box_j's detection confidence to zero.
                if boxes_iou(box_i, box_j) > iou_thresh:
                    box_j[4] = 0

    return best_boxes


def detect_objects(model, img, iou_thresh, nms_thresh):
    # Start the time. This is done to calculate how long the detection takes.
    start = time.time()

    # Set the model to evaluation mode.
    model.eval()

    # Convert the image from a NumPy ndarray to a PyTorch Tensor of the correct shape.
    # The image is transposed, then converted to a FloatTensor of dtype float32, then
    # Normalized to values between 0 and 1, and finally unsqueezed to have the correct
    # shape of 1 x 3 x 416 x 416
    img = torch.from_numpy(img.transpose(2, 0, 1)).float().div(255.0).unsqueeze(0)

    # Feed the image to the neural network with the corresponding NMS threshold.
    # The first step in NMS is to remove all bounding boxes that have a very low
    # probability of detection. All predicted bounding boxes with a value less than
    # the given NMS threshold will be removed.
    list_boxes = model(img, nms_thresh)

    # Make a new list with all the bounding boxes returned by the neural network
    boxes = list_boxes[0][0] + list_boxes[1][0] + list_boxes[2][0]

    # Perform the second step of NMS on the bounding boxes returned by the neural network.
    # In this step, we only keep the best bounding boxes by eliminating all the bounding boxes
    # whose IOU value is higher than the given IOU threshold
    boxes = nms(boxes, iou_thresh)
```

```
    # Stop the time.
    finish = time.time()

    # Print the time it took to detect objects
    print('\n\nIt took {:.3f}'.format(finish – start), 'seconds to detect the objects in the image.\n')

    # Print the number of objects detected
    print('Number of Objects Detected:', len(boxes), '\n')

    return boxes


def load_class_names(namesfile):
    # Create an empty list to hold the object classes
    class_names = []

    # Open the file containing the COCO object classes in read-only mode
    with open(namesfile, 'r') as fp:
        # The coco.names file contains only one object class per line.
        # Read the file line by line and save all the lines in a list.
        lines = fp.readlines()

    # Get the object class names
    for line in lines:
        # Make a copy of each line with any trailing whitespace removed
        line = line.rstrip()

        # Save the object class name into class_names
        class_names.append(line)

    return class_names


def print_objects(boxes, class_names):
    print('Objects Found and Confidence Level:\n')
    for I in range(len(boxes)):
        box = boxes[i]
        if len(box) >= 7 and class_names:
            cls_conf = box[5]
            cls_id = box[6]
            print('%i. %s: %f' % (I + 1, class_names[cls_id], cls_conf))


def plot_boxes(img, boxes, class_names, plot_labels, color=None):
    # Define a tensor used to set the colors of the bounding boxes
    colors = torch.FloatTensor([[1, 0, 1], [0, 0, 1], [0, 1, 1], [0, 1, 0], [1, 1, 0], [1, 0, 0]])

    # Define a function to set the colors of the bounding boxes
    def get_color(c, x, max_val):
        ratio = float(x) / max_val * 5
        I = int(np.floor(ratio))
        j = int(np.ceil(ratio))

        ratio = ratio – i
        r = (1 – ratio) * colors[i][c] + ratio * colors[j][c]

        return int(r * 255)

    # Get the width and height of the image
    width = img.shape[1]
    height = img.shape[0]

    # Create a figure and plot the image
    fig, a = plt.subplots(1, 1)
    a.imshow(img)

    # Plot the bounding boxes and corresponding labels on top of the image
    for I in range(len(boxes)):

        # Get the ith bounding box
```

```
        box = boxes[i]

        # Get the (x,y) pixel coordinates of the lower-left and lower-right corners
        # of the bounding box relative to the size of the image.
        x1 = int(np.around((box[0] – box[2] / 2.0) * width))
        y1 = int(np.around((box[1] – box[3] / 2.0) * height))
        x2 = int(np.around((box[0] + box[2] / 2.0) * width))
        y2 = int(np.around((box[1] + box[3] / 2.0) * height))

        # Set the default rgb value to red
        rgb = (1, 0, 0)

        # Use the same color to plot the bounding boxes of the same object class
        if len(box) >= 7 and class_names:
            cls_conf = box[5]
            cls_id = box[6]
            classes = len(class_names)
            offset = cls_id * 123457 % classes
            red = get_color(2, offset, classes) / 255
            green = get_color(1, offset, classes) / 255
            blue = get_color(0, offset, classes) / 255

            # If a color is given then set rgb to the given color instead
            if color is None:
                rgb = (red, green, blue)
            else:
                rgb = color

        # Calculate the width and height of the bounding box relative to the size of the image.
        width_x = x2 – x1
        width_y = y1 – y2

        # Set the postion and size of the bounding box. (x1, y2) is the pixel coordinate of the
        # lower-left corner of the bounding box relative to the size of the image.
        rect = patches.Rectangle((x1, y2),
                                 width_x, width_y,
                                 linewidth=2,
                                 edgecolor=rgb,
                                 facecolor='none')

        # Draw the bounding box on top of the image
        a.add_patch(rect)

        # If plot_labels = True then plot the corresponding label
        if plot_labels:
            # Create a string with the object class name and the corresponding object class probability
            conf_tx = class_names[cls_id] + ': {:.1f}'.format(cls_conf)

            # Define x and y offsets for the labels
            lxc = (img.shape[1] * 0.266) / 100
            lyc = (img.shape[0] * 1.180) / 100

            # Draw the labels on top of the image
            a.text(x1 + lxc, y1 – lyc, conf_tx, fontsize=24, color='k',
                   bbox=dict(facecolor=rgb, edgecolor=rgb, alpha=0.8))

end = time.time()

# total time taken
print(f"Runtime of the program is {end - start}")


    plt.show()
```

A.2) Performs all convolutions and probability functions

```
'Import torch
import torch.nn as nn
```

```
import numpy as np


class YoloLayer(nn.Module):
    def __init__(self, anchor_mask=[], num_classes=0, anchors=[], num_anchors=1):
        super(YoloLayer, self).__init__()
        self.anchor_mask = anchor_mask
        self.num_classes = num_classes
        self.anchors = anchors
        self.num_anchors = num_anchors
        self.anchor_step = len(anchors) / num_anchors
        self.coord_scale = 1
        self.noobject_scale = 1
        self.object_scale = 5
        self.class_scale = 1
        self.thresh = 0.6
        self.stride = 32
        self.seen = 0

    def forward(self, output, nms_thresh):
        self.thresh = nms_thresh
        masked_anchors = []

        for m in self.anchor_mask:
            masked_anchors += self.anchors[m * self.anchor_step☹m + 1) * self.anchor_step]

        masked_anchors = [anchor / self.stride for anchor in masked_anchors]
        boxes = get_region_boxes(output.data, self.thresh, self.num_classes, masked_anchors, len(self.anchor_mask))

        return boxes


class Upsample(nn.Module):
    def __init__(self, stride=2):
        super(Upsample, self).__init__()
        self.stride = stride

    def forward(self, x):
        stride = self.stride
        assert (x.data.dim() == 4)
        B = x.data.size(0)
        C = x.data.size(1)
        H = x.data.size(2)
        W = x.data.size(3)
        ws = stride
        hs = stride
        x = x.view(B, C, H, 1, W, 1).expand(B, C, H, stride, W, stride).contiguous().view(B, C, H * stride, W * stride)
        return x


# for route and shortcut
class EmptyModule(nn.Module):
    def __init__(self):
        super(EmptyModule, self).__init__()

    def forward(self, x):
        return x


# support route shortcut
class Darknet(nn.Module):
    def __init__(self, cfgfile):
        super(Darknet, self).__init__()
        self.blocks = parse_cfg(cfgfile)
        self.models = self.create_network(self.blocks)  # merge conv, bn,leaky
        self.loss = self.models[len(self.models) – 1]

        self.width = int(self.blocks[0][‘width’])
        self.height = int(self.blocks[0][‘height’])
```

```
    self.header = torch.IntTensor([0, 0, 0, 0])
    self.seen = 0

def forward(self, x, nms_thresh):
    ind = -2
    self.loss = None
    outputs = dict()
    out_boxes = []

    for block in self.blocks:
        ind = ind + 1
        if block['type'] == 'net':
            continue
        elif block['type'] in ['convolutional', 'upsample']:
            x = self.models[ind](x)
            outputs[ind] = x
        elif block['type'] == 'route':
            layers = block['layers'].split(',')
            layers = [int(i) if int(i) > 0 else int(i) + ind for l in layers]
            if len(layers) == 1:
                x = outputs[layers[0]]
                outputs[ind] = x
            elif len(layers) == 2:
                x1 = outputs[layers[0]]
                x2 = outputs[layers[1]]
                x = torch.cat((x1, x2), 1)
                outputs[ind] = x
        elif block['type'] == 'shortcut':
            from_layer = int(block['from'])
            activation = block['activation']
            from_layer = from_layer if from_layer > 0 else from_layer + ind
            x1 = outputs[from_layer]
            x2 = outputs[ind – 1]
            x = x1 + x2
            outputs[ind] = x
        elif block['type'] == 'yolo':
            boxes = self.models[ind](x, nms_thresh)
            out_boxes.append(boxes)
        else:
            print('unknown type %s' % (block['type']))

    return out_boxes

def print_network(self):
    print_cfg(self.blocks)

def create_network(self, blocks):
    models = nn.ModuleList()

    prev_filters = 3
    out_filters = []
    prev_stride = 1
    out_strides = []
    conv_id = 0
    for block in blocks:
        if block['type'] == 'net':
            prev_filters = int(block['channels'])
            continue
        elif block['type'] == 'convolutional':
            conv_id = conv_id + 1
            batch_normalize = int(block['batch_normalize'])
            filters = int(block['filters'])
            kernel_size = int(block['size'])
            stride = int(block['stride'])
            is_pad = int(block['pad'])
            pad = (kernel_size – 1) // 2 if is_pad else 0
            activation = block['activation']
            model = nn.Sequential()
            if batch_normalize:
```

```
                model.add_module('conv{0}'.format(conv_id),
                                 nn.Conv2d(prev_filters, filters, kernel_size, stride, pad, bias=False))
                model.add_module('bn{0}'.format(conv_id), nn.BatchNorm2d(filters))
            else:
                model.add_module('conv{0}'.format(conv_id),
                                 nn.Conv2d(prev_filters, filters, kernel_size, stride, pad))
            if activation == 'leaky':
                model.add_module('leaky{0}'.format(conv_id), nn.LeakyReLU(0.1, inplace=True))
            prev_filters = filters
            out_filters.append(prev_filters)
            prev_stride = stride * prev_stride
            out_strides.append(prev_stride)
            models.append(model)
        elif block['type'] == 'upsample':
            stride = int(block['stride'])
            out_filters.append(prev_filters)
            prev_stride = prev_stride // stride
            out_strides.append(prev_stride)
            models.append(Upsample(stride))
        elif block['type'] == 'route':
            layers = block['layers'].split(',')
            ind = len(models)
            layers = [int(i) if int(i) > 0 else int(i) + ind for I in layers]
            if len(layers) == 1:
                prev_filters = out_filters[layers[0]]
                prev_stride = out_strides[layers[0]]
            elif len(layers) == 2:
                assert (layers[0] == ind – 1)
                prev_filters = out_filters[layers[0]] + out_filters[layers[1]]
                prev_stride = out_strides[layers[0]]
            out_filters.append(prev_filters)
            out_strides.append(prev_stride)
            models.append(EmptyModule())
        elif block['type'] == 'shortcut':
            ind = len(models)
            prev_filters = out_filters[ind – 1]
            out_filters.append(prev_filters)
            prev_stride = out_strides[ind – 1]
            out_strides.append(prev_stride)
            models.append(EmptyModule())
        elif block['type'] == 'yolo':
            yolo_layer = YoloLayer()
            anchors = block['anchors'].split(',')
            anchor_mask = block['mask'].split(',')
            yolo_layer.anchor_mask = [int(i) for I in anchor_mask]
            yolo_layer.anchors = [float(i) for I in anchors]
            yolo_layer.num_classes = int(block['classes'])
            yolo_layer.num_anchors = int(block['num'])
            yolo_layer.anchor_step = len(yolo_layer.anchors) // yolo_layer.num_anchors
            yolo_layer.stride = prev_stride
            out_filters.append(prev_filters)
            out_strides.append(prev_stride)
            models.append(yolo_layer)
        else:
            print('unknown type %s' % (block['type']))

    return models

def load_weights(self, weightfile):
    print()
    fp = open(weightfile, 'rb')
    header = np.fromfile(fp, count=5, dtype=np.int32)
    self.header = torch.from_numpy(header)
    self.seen = self.header[3]
    buf = np.fromfile(fp, dtype=np.float32)
    fp.close()

    start = 0
    ind = -2
    counter = 3
```

```
        for block in self.blocks:
            if start >= buf.size:
                break
            ind = ind + 1
            if block['type'] == 'net':
                continue
            elif block['type'] == 'convolutional':
                model = self.models[ind]
                batch_normalize = int(block['batch_normalize'])
                if batch_normalize:
                    start = load_conv_bn(buf, start, model[0], model[1])
                else:
                    start = load_conv(buf, start, model[0])
            elif block['type'] == 'upsample':
                pass
            elif block['type'] == 'route':
                pass
            elif block['type'] == 'shortcut':
                pass
            elif block['type'] == 'yolo':
                pass
            else:
                print('unknown type %s' % (block['type']))

            percent_comp = (counter / len(self.blocks)) * 100

            print('Loading weights. Please Wait...{:.2f}% Complete'.format(percent_comp), end='\r', flush=True)

            counter += 1


def convert2cpu(gpu_matrix):
    return torch.FloatTensor(gpu_matrix.size()).copy_(gpu_matrix)


def convert2cpu_long(gpu_matrix):
    return torch.LongTensor(gpu_matrix.size()).copy_(gpu_matrix)


def get_region_boxes(output, conf_thresh, num_classes, anchors, num_anchors, only_objectness=1, validation=False):
    anchor_step = len(anchors) // num_anchors
    if output.dim() == 3:
        output = output.unsqueeze(0)
    batch = output.size(0)
    assert (output.size(1) == (5 + num_classes) * num_anchors)
    h = output.size(2)
    w = output.size(3)

    all_boxes = []
    output = output.view(batch * num_anchors, 5 + num_classes, h * w).transpose(0, 1).contiguous().view(5 + num_classes,
                                                                                                         batch * num_anchors * h * w)

    grid_x = torch.linspace(0, w – 1, w).repeat(h, 1).repeat(batch * num_anchors, 1, 1).view(
        batch * num_anchors * h * w).type_as(output)  # cuda()
    grid_y = torch.linspace(0, h – 1, h).repeat(w, 1).t().repeat(batch * num_anchors, 1, 1).view(
        batch * num_anchors * h * w).type_as(output)  # cuda()
    xs = torch.sigmoid(output[0]) + grid_x
    ys = torch.sigmoid(output[1]) + grid_y

    anchor_w = torch.Tensor(anchors).view(num_anchors, anchor_step).index_select(1, torch.LongTensor([0]))
    anchor_h = torch.Tensor(anchors).view(num_anchors, anchor_step).index_select(1, torch.LongTensor([1]))
    anchor_w = anchor_w.repeat(batch, 1).repeat(1, 1, h * w).view(batch * num_anchors * h * w).type_as(output)  # cuda()
    anchor_h = anchor_h.repeat(batch, 1).repeat(1, 1, h * w).view(batch * num_anchors * h * w).type_as(output)  # cuda()
    ws = torch.exp(output[2]) * anchor_w
    hs = torch.exp(output[3]) * anchor_h

    det_confs = torch.sigmoid(output[4])
    cls_confs = torch.nn.Softmax(dim=1)(output[5:5 + num_classes].transpose(0, 1)).detach()
    cls_max_confs, cls_max_ids = torch.max(cls_confs, 1)
    cls_max_confs = cls_max_confs.view(-1)
```

```
    cls_max_ids = cls_max_ids.view(-1)

    sz_hw = h * w
    sz_hwa = sz_hw * num_anchors
    det_confs = convert2cpu(det_confs)
    cls_max_confs = convert2cpu(cls_max_confs)
    cls_max_ids = convert2cpu_long(cls_max_ids)
    xs = convert2cpu(xs)
    ys = convert2cpu(ys)
    ws = convert2cpu(ws)
    hs = convert2cpu(hs)
    if validation:
        cls_confs = convert2cpu(cls_confs.view(-1, num_classes))

    for b in range(batch):
        boxes = []
        for cy in range(h):
            for cx in range(w):
                for I in range(num_anchors):
                    ind = b * sz_hwa + I * sz_hw + cy * w + cx
                    det_conf = det_confs[ind]
                    if only_objectness:
                        conf = det_confs[ind]
                    else:
                        conf = det_confs[ind] * cls_max_confs[ind]

                    if conf > conf_thresh:
                        bcx = xs[ind]
                        bcy = ys[ind]
                        bw = ws[ind]
                        bh = hs[ind]
                        cls_max_conf = cls_max_confs[ind]
                        cls_max_id = cls_max_ids[ind]
                        box = [bcx / w, bcy / h, bw / w, bh / h, det_conf, cls_max_conf, cls_max_id]
                        if (not only_objectness) and validation:
                            for c in range(num_classes):
                                tmp_conf = cls_confs[ind][c]
                                if c != cls_max_id and det_confs[ind] * tmp_conf > conf_thresh:
                                    box.append(tmp_conf)
                                    box.appendI
                        boxes.append(box)
        all_boxes.append(boxes)

    return all_boxes


def parse_cfg(cfgfile):
    blocks = []
    fp = open(cfgfile, 'r')
    block = None
    line = fp.readline()
    while line != '':
        line = line.rstrip()
        if line == '' or line[0] == '#':
            line = fp.readline()
            continue
        elif line[0] == '[':
            if block:
                blocks.append(block)
            block = dict()
            block['type'] = line.lstrip('[').rstrip(')')
            # set default value
            if block['type'] == 'convolutional':
                block['batch_normalize'] = 0
        else:
            key, value = line.split('=')
            key = key.strip()
            if key == 'type':
                key = '_type'
            value = value.strip()
```

```
            block[key] = value
        line = fp.readline()

    if block:
        blocks.append(block)
    fp.close()
    return blocks


def print_cfg(blocks):
    print('layer     filters    size              input                output');
    prev_width = 416
    prev_height = 416
    prev_filters = 3
    out_filters = []
    out_widths = []
    out_heights = []
    ind = -2
    for block in blocks:
        ind = ind + 1
        if block['type'] == 'net':
            prev_width = int(block['width'])
            prev_height = int(block['height'])
            continue
        elif block['type'] == 'convolutional':
            filters = int(block['filters'])
            kernel_size = int(block['size'])
            stride = int(block['stride'])
            is_pad = int(block['pad'])
            pad = (kernel_size – 1) // 2 if is_pad else 0
            width = (prev_width + 2 * pad – kernel_size) // stride + 1
            height = (prev_height + 2 * pad – kernel_size) // stride + 1
            print('%5d %-6s %4d  %d x %d / %d   %3d x %3d x%4d   ->   %3d x %3d x%4d' % (
            ind, 'conv', filters, kernel_size, kernel_size, stride, prev_width, prev_height, prev_filters, width,
            height, filters))
            prev_width = width
            prev_height = height
            prev_filters = filters
            out_widths.append(prev_width)
            out_heights.append(prev_height)
            out_filters.append(prev_filters)
        elif block['type'] == 'upsample':
            stride = int(block['stride'])
            filters = prev_filters
            width = prev_width * stride
            height = prev_height * stride
            print('%5d %-6s           * %d   %3d x %3d x%4d   ->   %3d x %3d x%4d' % (
            ind, 'upsample', stride, prev_width, prev_height, prev_filters, width, height, filters))
            prev_width = width
            prev_height = height
            prev_filters = filters
            out_widths.append(prev_width)
            out_heights.append(prev_height)
            out_filters.append(prev_filters)
        elif block['type'] == 'route':
            layers = block['layers'].split(',')
            layers = [int(i) if int(i) > 0 else int(i) + ind for I in layers]
            if len(layers) == 1:
                print('%5d %-6s %d' % (ind, 'route', layers[0]))
                prev_width = out_widths[layers[0]]
                prev_height = out_heights[layers[0]]
                prev_filters = out_filters[layers[0]]
            elif len(layers) == 2:
                print('%5d %-6s %d %d' % (ind, 'route', layers[0], layers[1]))
                prev_width = out_widths[layers[0]]
                prev_height = out_heights[layers[0]]
                assert (prev_width == out_widths[layers[1]])
                assert (prev_height == out_heights[layers[1]])
                prev_filters = out_filters[layers[0]] + out_filters[layers[1]]
            out_widths.append(prev_width)
```

```
            out_heights.append(prev_height)
            out_filters.append(prev_filters)
        elif block['type'] in ['region', 'yolo']:
            print('%5d %-6s' % (ind, 'detection'))
            out_widths.append(prev_width)
            out_heights.append(prev_height)
            out_filters.append(prev_filters)
        elif block['type'] == 'shortcut':
            from_id = int(block['from'])
            from_id = from_id if from_id > 0 else from_id + ind
            print('%5d %-6s %d' % (ind, 'shortcut', from_id))
            prev_width = out_widths[from_id]
            prev_height = out_heights[from_id]
            prev_filters = out_filters[from_id]
            out_widths.append(prev_width)
            out_heights.append(prev_height)
            out_filters.append(prev_filters)
        else:
            print('unknown type %s' % (block['type']))


def load_conv(buf, start, conv_model):
    num_w = conv_model.weight.numel()
    num_b = conv_model.bias.numel()
    conv_model.bias.data.copy_(torch.from_numpy(buf[start:start + num_b]));
    start = start + num_b
    conv_model.weight.data.copy_(torch.from_numpy(buf[start:start + num_w]).view_as(conv_model.weight.data));
    start = start + num_w
    return start


def load_conv_bn(buf, start, conv_model, bn_model):
    num_w = conv_model.weight.numel()
    num_b = bn_model.bias.numel()
    bn_model.bias.data.copy_(torch.from_numpy(buf[start:start + num_b]));
    start = start + num_b
    bn_model.weight.data.copy_(torch.from_numpy(buf[start:start + num_b]));
    start = start + num_b
    bn_model.running_mean.copy_(torch.from_numpy(buf[start:start + num_b]));
    start = start + num_b
    bn_model.running_var.copy_(torch.from_numpy(buf[start:start + num_b]));
    start = start + num_b
    conv_model.weight.data.copy_(torch.from_numpy(buf[start:start + num_w]).view_as(conv_model.weight.data));
    start = start + num_w
    return start"
```

Appendix B = Fast R-CNN .py files

B.1) Draws the filter and sample images, etc.

```
from caffe.proto import caffe_pb2
from google.protobuf import text_format
import pydot

# Internal layer and blob styles.
LAYER_STYLE_DEFAULT = {'shape': 'record', 'fillcolor': '#6495ED',
        'style': 'filled'}
NEURON_LAYER_STYLE = {'shape': 'record', 'fillcolor': '#90EE90',
        'style': 'filled'}
BLOB_STYLE = {'shape': 'octagon', 'fillcolor': '#E0E0E0',
       'style': 'filled'}

def get_pooling_types_dict():
    """Get dictionary mapping pooling type number to type name
    """
    desc = caffe_pb2.PoolingParameter.PoolMethod.DESCRIPTOR
    d = {}
```

```
    for k,v in desc.values_by_name.items():
        d[v.number] = k
    return d


def determine_edge_label_by_layertype(layer, layertype):
    """Define edge label based on layer type
    """

    if layertype == 'Data':
        edge_label = 'Batch ' + str(layer.data_param.batch_size)
    elif layertype == 'Convolution':
        edge_label = str(layer.convolution_param.num_output)
    elif layertype == 'InnerProduct':
        edge_label = str(layer.inner_product_param.num_output)
    else:
        edge_label = '""'

    return edge_label


def determine_node_label_by_layertype(layer, layertype, rankdir):
    """Define node label based on layer type
    """

    if rankdir in ('TB', 'BT'):
        # If graph orientation is vertical, horizontal space is free and
        # vertical space is not; separate words with spaces
        separator = ' '
    else:
        # If graph orientation is horizontal, vertical space is free and
        # horizontal space is not; separate words with newlines
        separator = '\n'

    if layertype == 'Convolution':
        # Outer double quotes needed or else colon characters don't parse
        # properly
        node_label = '"%s%s(%s)%skernel size: %d%sstride: %d%spad: %d"' %\
                     (layer.name,
                      separator,
                      layertype,
                      separator,
                      layer.convolution_param.kernel_size,
                      separator,
                      layer.convolution_param.stride,
                      separator,
                      layer.convolution_param.pad)
    elif layertype == 'Pooling':
        pooling_types_dict = get_pooling_types_dict()
        node_label = '"%s%s(%s %s)%skernel size: %d%sstride: %d%spad: %d"' %\
                     (layer.name,
                      separator,
                      pooling_types_dict[layer.pooling_param.pool],
                      layertype,
                      separator,
                      layer.pooling_param.kernel_size,
                      separator,
                      layer.pooling_param.stride,
                      separator,
                      layer.pooling_param.pad)
    else:
        node_label = '"%s%s(%s)"' % (layer.name, separator, layertype)
    return node_label


def choose_color_by_layertype(layertype):
    """Define colors for nodes based on the layer type
    """
    color = '#6495ED'  # Default
    if layertype == 'Convolution':
```

```
      color = '#FF5050'
    elif layertype == 'Pooling':
      color = '#FF9900'
    elif layertype == 'InnerProduct':
      color = '#CC33FF'
    return color


def get_pydot_graph(caffe_net, rankdir, label_edges=True):
  pydot_graph = pydot.Dot(caffe_net.name, graph_type='digraph', rankdir=rankdir)
  pydot_nodes = {}
  pydot_edges = []
  for layer in caffe_net.layer:
    name = layer.name
    layertype = layer.type
    node_label = determine_node_label_by_layertype(layer, layertype, rankdir)
    if (len(layer.bottom) == 1 and len(layer.top) == 1 and
        layer.bottom[0] == layer.top[0]):
      # We have an in-place neuron layer.
      pydot_nodes[name + '_' + layertype] = pydot.Node(
          node_label, **NEURON_LAYER_STYLE)
    else:
      layer_style = LAYER_STYLE_DEFAULT
      layer_style['fillcolor'] = choose_color_by_layertype(layertype)
      pydot_nodes[name + '_' + layertype] = pydot.Node(
          node_label, **layer_style)
    for bottom_blob in layer.bottom:
      pydot_nodes[bottom_blob + '_blob'] = pydot.Node(
        '%s' % (bottom_blob), **BLOB_STYLE)
      edge_label = '""'
      pydot_edges.append({'src': bottom_blob + '_blob',
                          'dst': name + '_' + layertype,
                          'label': edge_label})
    for top_blob in layer.top:
      pydot_nodes[top_blob + '_blob'] = pydot.Node(
        '%s' % (top_blob))
      if label_edges:
        edge_label = determine_edge_label_by_layertype(layer, layertype)
      else:
        edge_label = '""'
      pydot_edges.append({'src': name + '_' + layertype,
                          'dst': top_blob + '_blob',
                          'label': edge_label})
  # Now, add the nodes and edges to the graph.
  for node in pydot_nodes.values():
    pydot_graph.add_node(node)
  for edge in pydot_edges:
    pydot_graph.add_edge(
        pydot.Edge(pydot_nodes[edge['src']], pydot_nodes[edge['dst']],
                   label=edge['label']))
  return pydot_graph

def draw_net(caffe_net, rankdir, ext='png'):
  """Draws a caffe net and returns the image string encoded using the given
  extension.
  Input:
    caffe_net: a caffe.proto.caffe_pb2.NetParameter protocol buffer.
    ext: the image extension. Default 'png'.
  """
  return get_pydot_graph(caffe_net, rankdir).create(format=ext)

def draw_net_to_file(caffe_net, filename, rankdir='LR'):
  """Draws a caffe net, and saves it to file using the format given as the
  file extension. Use '.raw' to output raw text that you can manually feed
  to graphviz to draw graphs.
  """
  ext = filename[filename.rfind('.')+1:]
  with open(filename, 'wb') as fid:
    fid.write(draw_net(caffe_net, rankdir, ext))
```

B.2) Code which detects an object

```
import numpy as np
import os
import time

import caffe

start = time.time()

class Detector(caffe.Net):
    """
    Detector extends Net for windowed detection by a list of crops or
    selective search proposals.
    """
    def __init__(self, model_file, pretrained_file, mean=None,
                 input_scale=None, raw_scale=None, channel_swap=None,
                 context_pad=None):
        """
        Take
          mean, input_scale, raw_scale, channel_swap: params for
          preprocessing options.
        context_pad: amount of surrounding context to take s.t. a `context_pad`
          sized border of pixels in the network input image is context, as in
          R-CNN feature extraction.
        """
        caffe.Net.__init__(self, model_file, pretrained_file, caffe.TEST)

        # configure pre-processing
        in_ = self.inputs[0]
        self.transformer = caffe.io.Transformer(
          {in_: self.blobs[in_].data.shape})
        self.transformer.set_transpose(in_, (2,0,1))
        if mean is not None:
          self.transformer.set_mean(in_, mean)
        if input_scale is not None:
          self.transformer.set_input_scale(in_, input_scale)
        if raw_scale is not None:
          self.transformer.set_raw_scale(in_, raw_scale)
        if channel_swap is not None:
          self.transformer.set_channel_swap(in_, channel_swap)

        self.configure_crop(context_pad)


    def detect_windows(self, images_windows):
        """
        Do windowed detection over given images and windows. Windows are
        extracted then warped to the input dimensions of the net.
        Take
        images_windows: (image filename, window list) iterable.
        context_crop: size of context border to crop in pixels.
        Give
        detections: list of {filename: image filename, window: crop coordinates,
          predictions: prediction vector} dicts.
        """
        # Extract windows.
        window_inputs = []
        for image_fname, windows in images_windows:
          image = caffe.io.load_image(image_fname).astype(np.float32)
          for window in windows:
            window_inputs.append(self.crop(image, window))

        # Run through the net (warping windows to input dimensions).
        in_ = self.inputs[0]
        caffe_in = np.zeros((len(window_inputs), window_inputs[0].shape[2])
                  + self.blobs[in_].data.shape[2:],
                  dtype=np.float32)
        for ix, window_in in enumerate(window_inputs):
          caffe_in[ix] = self.transformer.preprocess(in_, window_in)
```

```
    out = self.forward_all(**{in_: caffe_in})
    predictions = out[self.outputs[0]].squeeze(axis=(2,3))

    # Package predictions with images and windows.
    detections = []
    ix = 0
    for image_fname, windows in images_windows:
        for window in windows:
            detections.append({
                'window': window,
                'prediction': predictions[ix],
                'filename': image_fname
            })
            ix += 1
    return detections


def detect_selective_search(self, image_fnames):
    """
    Do windowed detection over Selective Search proposals by extracting
    the crop and warping to the input dimensions of the net.
    Take
    image_fnames: list
    Give
    detections: list of {filename: image filename, window: crop coordinates,
        predictions: prediction vector} dicts.
    """
    import selective_search_ijcv_with_python as selective_search
    # Make absolute paths so MATLAB can find the files.
    image_fnames = [os.path.abspath(f) for f in image_fnames]
    windows_list = selective_search.get_windows(
        image_fnames,
        cmd='selective_search_rcnn'
    )
    # Run windowed detection on the selective search list.
    return self.detect_windows(zip(image_fnames, windows_list))


def crop(self, im, window):
    """
    Crop a window from the image for detection. Include surrounding context
    according to the `context_pad` configuration.
    Take
    im: H x W x K image ndarray to crop.
    window: bounding box coordinates as ymin, xmin, ymax, xmax.
    Give
    crop: cropped window.
    """
    # Crop window from the image.
    crop = im[window[0]:window[2], window[1]:window[3]]

    if self.context_pad:
        box = window.copy()
        crop_size = self.blobs[self.inputs[0]].width  # assumes square
        scale = crop_size / (1. * crop_size – self.context_pad * 2)
        # Crop a box + surrounding context.
        half_h = (box[2] – box[0] + 1) / 2.
        half_w = (box[3] – box[1] + 1) / 2.
        center = (box[0] + half_h, box[1] + half_w)
        scaled_dims = scale * np.array((-half_h, -half_w, half_h, half_w))
        box = np.round(np.tile(center, 2) + scaled_dims)
        full_h = box[2] – box[0] + 1
        full_w = box[3] – box[1] + 1
        scale_h = crop_size / full_h
        scale_w = crop_size / full_w
        pad_y = round(max(0, -box[0]) * scale_h)  # amount out-of-bounds
        pad_x = round(max(0, -box[1]) * scale_w)

        # Clip box to image dimensions.
        im_h, im_w = im.shape[:2]
```

```
            box = np.clip(box, 0., [im_h, im_w, im_h, im_w])
            clip_h = box[2] – box[0] + 1
            clip_w = box[3] – box[1] + 1
            assert(clip_h > 0 and clip_w > 0)
            crop_h = round(clip_h * scale_h)
            crop_w = round(clip_w * scale_w)
            if pad_y + crop_h > crop_size:
                crop_h = crop_size – pad_y
            if pad_x + crop_w > crop_size:
                crop_w = crop_size – pad_x

            # collect with context padding and place in input
            # with mean padding
            context_crop = im[box[0]:box[2], box[1]:box[3]]
            context_crop = caffe.io.resize_image(context_crop, (crop_h, crop_w))
            crop = np.ones(self.crop_dims, dtype=np.float32) * self.crop_mean
            crop[pad_y☹pad_y + crop_h), pad_x☹pad_x + crop_w)] = context_crop

        return crop


    def configure_crop(self, context_pad):
        “””
        Configure crop dimensions and amount of context for cropping.
        If context is included, make the special input mean for context padding.
        Take
        context_pad: amount of context for cropping.
        “””
        # crop dimensions
        in_ = self.inputs[0]
        tpose = self.transformer.transpose[in_]
        inv_tpose = [tpose[t] for t in tpose]
        self.crop_dims = np.array(self.blobs[in_].data.shape[1:])[inv_tpose]
        #.transpose(inv_tpose)
        # context padding
        self.context_pad = context_pad
        if self.context_pad:
            in_ = self.inputs[0]
            transpose = self.transformer.transpose.get(in_)
            channel_order = self.transformer.channel_swap.get(in_)
            raw_scale = self.transformer.raw_scale.get(in_)
            # Padding context crops needs the mean in unprocessed input space.
            mean = self.transformer.mean.get(in_)
            if mean is not None:
                inv_transpose = [transpose[t] for t in transpose]
                crop_mean = mean.copy().transpose(inv_transpose)
                if channel_order is not None:
                    channel_order_inverse = [channel_order.index(i)
                                            for I in range(crop_mean.shape[2])]
                    crop_mean = crop_mean[:,:, channel_order_inverse]
                if raw_scale is not None:
                    crop_mean /= raw_scale
                self.crop_mean = crop_mean
            else:
                self.crop_mean = np.zeros(self.crop_dims, dtype=np.float32)

end = time.time()
```

B.3) Code which classifies the detected object

```
import numpy as np
import time

import caffe
```

```
start = time.time()

class Classifier(caffe.Net):
    """
    Classifier extends Net for image class prediction
    by scaling, center cropping, or oversampling.
    """
    def __init__(self, model_file, pretrained_file, image_dims=None,
                 mean=None, input_scale=None, raw_scale=None,
                 channel_swap=None):
        """
        Take
        image_dims: dimensions to scale input for cropping/sampling.
            Default is to scale to net input size for whole-image crop.
            mean, input_scale, raw_scale, channel_swap: params for
            preprocessing options.
        """
        caffe.Net.__init__(self, model_file, pretrained_file, caffe.TEST)

        # configure pre-processing
        in_ = self.inputs[0]
        self.transformer = caffe.io.Transformer(
            {in_: self.blobs[in_].data.shape})
        self.transformer.set_transpose(in_, (2,0,1))
        if mean is not None:
            self.transformer.set_mean(in_, mean)
        if input_scale is not None:
            self.transformer.set_input_scale(in_, input_scale)
        if raw_scale is not None:
            self.transformer.set_raw_scale(in_, raw_scale)
        if channel_swap is not None:
            self.transformer.set_channel_swap(in_, channel_swap)

        self.crop_dims = np.array(self.blobs[in_].data.shape[2:])
        if not image_dims:
            image_dims = self.crop_dims
        self.image_dims = image_dims


    def predict(self, inputs, oversample=True):
        """
        Predict classification probabilities of inputs.
        Take
        inputs: iterable of (H x W x K) input ndarrays.
        oversample: average predictions across center, corners, and mirrors
                when True (default). Center-only prediction when False.
        Give
        predictions: (N x C) ndarray of class probabilities
                 for N images and C classes.
        """
        # Scale to standardize input dimensions.
        input_ = np.zeros((len(inputs),
            self.image_dims[0], self.image_dims[1], inputs[0].shape[2]),
            dtype=np.float32)
        for ix, in_ in enumerate(inputs):
            input_[ix] = caffe.io.resize_image(in_, self.image_dims)

        if oversample:
            # Generate center, corner, and mirrored crops.
            input_ = caffe.io.oversample(input_, self.crop_dims)
        else:
            # Take center crop.
            center = np.array(self.image_dims) / 2.0
            crop = np.tile(center, (1, 2))[0] + np.concatenate([
                -self.crop_dims / 2.0,
                self.crop_dims / 2.0
            ])
            input_ = input_[:, crop[0]:crop[2], crop[1]:crop[3], :]
```

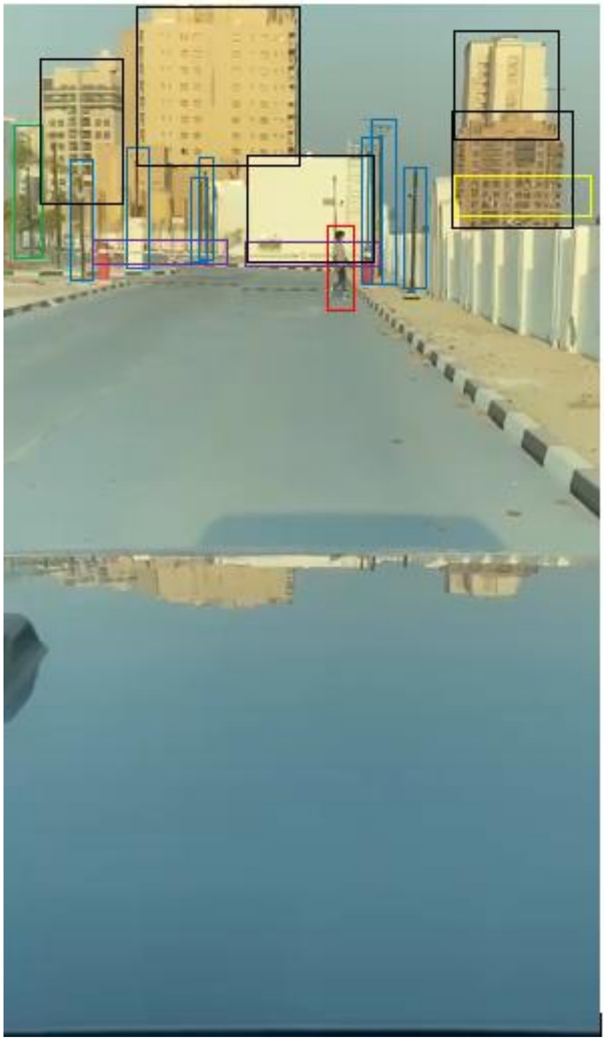
```
    # Classify
    caffe_in = np.zeros(np.array(input_.shape)[[0,3,1,2]],
                        dtype=np.float32)
    for ix, in_ in enumerate(input_):
        caffe_in[ix] = self.transformer.preprocess(self.inputs[0], in_)
    out = self.forward_all(**{self.inputs[0]: caffe_in})
    predictions = out[self.outputs[0]]

    # For  oversampling, average predictions across crops.
    if oversample:
        predictions = predictions.reshape((len(predictions) / 10, 10, -1))
        predictions = predictions.mean(1)

    return predictions
end = time.time()
```

Appendix C = Extract from the Waymo Open Data Set

Appendix D = Sample image after Fast R-CNN algorithm



Red = Human

Blue = Lamppost

Green = Tree

Purple = Boomgate

Yellow = Traffic Signal

Black = Building

Appendix E = Sample image after YOLO algorithm

Green = Human

Yellow = Lamppost

Blue = Tree

Black = Boomgate

N/A = Traffic Signal

Red = Building

Appendix F – Secondary Source Evaluation

Source	Strengths	Weaknesses
Github code (Garima13a)	• Includes all the methods and functions required for the programming, with low redundancy and high efficiency	• Not catered to th images and purpose, so the code was modifies slightly
Stanford University Report (Lewis)	• A detailed explanation & report with thorough experimentation conducted by students of the university • Very reliable source, with Stanford being one of the best universities in the world in this field	• The information provided was not always perfectly related to the topic, with other algorithms referenced • More focused on the mathematical aspect than the computer science one
TowardsDataScience – algorithms explained (Chablani), (Gandhi)	• Well structured, with step-by-step	• A lot of the keywords were not fully

	explanations on how the algorithms function • A good mix between pictures and text to allow for all types of readers to understand	explained and were rather confusing
Waymo Open data set (Etherington)	• Good series of images that worked effectively and were interpreted perfectly by the algorithms as training images	• Were centered around roads in Phoenix, Arizona, and thus weren't as efficient in training the algorithms to interpret the samples from another city
What is Computed Vision? – overview (IBM)	• A broad and intriguing overview was provided regarding computer vision and some of its key sub-concepts	• A lack of depth was evident, as the concepts were not detailed
Computer Vision book (D Prince)	• Includes all the concepts about	• A lot of small text which gets

	computer vision, and describes them in detail	considerably monotonous to read after a while