REPORT

ECE457B COURSE PROJECT, Winter 2019

Pedestrian Detection

Lingyi Li, 20566166, lingyi.li@uwaterloo.ca

Yun-Ting Tsai, 20575493, sylvia.tsai@uwaterloo.ca

Ziyue Wang, 20574111, sydney.wang@uwaterloo.ca

Haowen Zhan, 20551502, haowen.zhan@uwaterloo.ca

(Group 2)

Abstract

In Canada, it is estimated that 160,000 car accidents and 2,800 to 2,900 accident-related deaths occur every year [1]. This paper presents a software-based solution for pedestrian detection in crowded traffic scenes for self-driving vehicles to eliminate accidents and increase road safety.

The solution is developed based on the assumption that this problem cannot be solved effectively with conventional programming approaches. Instead, a computational intelligence approach might be more suitable for this given problem. This paper is mainly focused on the software side of the solution while assuming the input of the software is gathered from the vehicle's onboard camera(s).

The solution is implemented with YOLOv3, a fully convolutional neural network based object detection system. For pre-processing, frames in the Caltech Pedestrian Dataset training videos are taken based on a fixed sample rate, and annotations are converted into YOLO-compatible text format. Based on the bounding box dimensions from the target annotations, the optimal anchors for this algorithm are determined using a heuristic implementation of the K-means clustering algorithm. The model is then given a set of pre-trained weights as the starting point, and trained with weights from certain layers frozen to reduce training time and improve performance. The self-trained model is able to achieve a throughput of around 50 FPS on videos with a resolution of 416*416 using one Nvidia Tesla T4 GPU.

Several other models are then developed by using different parameters for number of epochs, batch size, and number of training data points. It is found that although training time appears to increase linearly with both the amount of training data and the number of epochs, they both improve accuracy. Doubling the batch size from 8 to 16 results in a small decrease in performance.

The self-trained YOLOv3 model is then benchmarked against pretrained YOLOv3, pre-trained Faster R-CNN and pre-trained SDD models. The custom-developed measure of accuracy takes both the bounding box area and confidence score into account, and is based on Intersection over Union (IoU). The self-trained YOLOv3 model is extremely fast with relatively high accuracy in comparison to the other pretrained models.

Once the software solution has been trained to the desired accuracy, the neural network can be deployed either on a driverless vehicle as part of the decision-making process or on a human-driven vehicle as a warning system.

Introduction and Motivation

Pedestrian detection is of great interest to researchers in the area of computer vision. Its real-life applications include robotics, surveillance, assistive technology, traffic safety, and automated driving. Automotive applications are particularly important as they have the potential to save millions of lives. Our goal is to design a neural network system that can detect and locate pedestrians in real-time.

This problem is intrinsically challenging mainly due to the unpredictable nature of pedestrians. Pedestrians can have varying physiques and actions, and their clothes (including unconventional clothing and costumes) and accessories might also hinder the detection process. Pedestrians might overlap or partially occlude each other [2]. There are also factors that could make detection more difficult in general, such as weather (e.g. heavy rain could induce reflection on the ground, fog reduces overall visibility) and time of day.

High accuracy and low latency are two main requirements for our solution. The goal is to achieve a detection accuracy that is high enough to be relied upon while keeping the latency under a threshold with limited compute power. High accuracy and low latency are required because the vehicle's control system needs sufficient time to respond to a change of environment. Failure to apply the proper reactive mechanisms under a certain time could lead to traffic collisions, with severe consequences such as potential injuries and loss of lives, and other monetary losses.

We decided to use YOLOv3, a fully convolutional neural network based approach to solve this problem because it is one of the fastest object detection algorithms. It is believed to be a great choice for real-time detection without too much loss of accuracy.

Theoretical Background

YOLO Architecture

YOLO is an object detection system that can determine if the target object is present in the image and locate it. YOLO uses a convolutional neural network as its architecture. In each convolutional layer, the convolution operation is applied to its input, and the output is passed onto the next layer. Figure 1 illustrates YOLO's convolutional neural network layer architecture. This is a simplified version of the model with a total of 24 convolutional layers together and two fully connected layers. Two definitions apply here. As Figure 1 shows, there are conv layer as well as maxpool layer. Conv layer applies filter to the input images, and pool layer reduce the output's dimensionality. Each pool layer takes summarization operation for each image patch. The maxpool layer takes the maximum value of the pool layers and process to the next Conv layer. The process helps reduce dimensionality and thus reduces the number of parameters, which in turn lessens the system's computational burden [3].

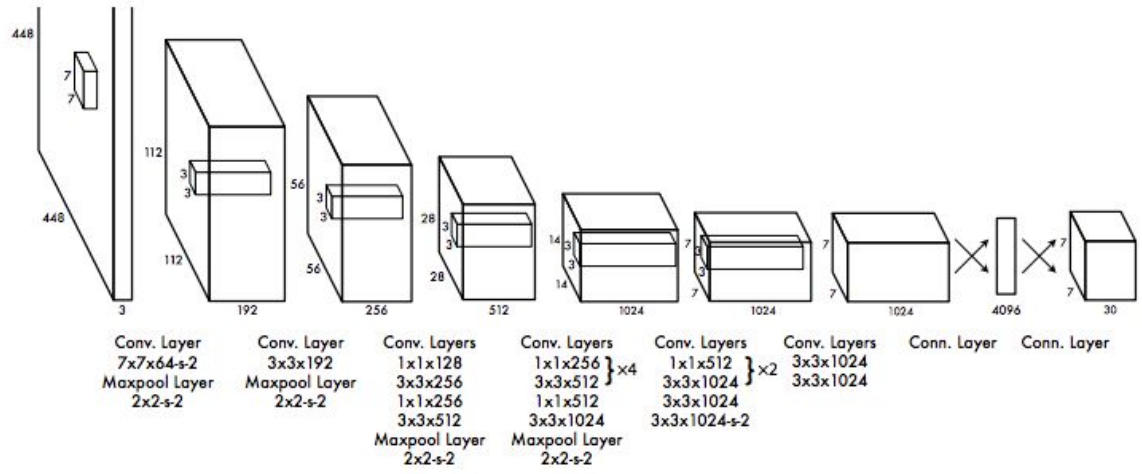

Figure 1. YOLO's Architecture

Unlike the other single convolutional neural networks that use only part of an image to train its network, YOLO trains on complete images while improves training speed and the mean average precision. According to YOLO's official paper, YOLOv1 runs at 45 fps with no batch processing and can achieve 150 fps in its fast version. This high fps enables the possibility of real-time pedestrian detection in videos with less than 25ms latency [4]. Since YOLO retains information about the complete image, the possibility for mis-detecting background patches is decreased. Moreover, YOLO's ability to generalize also provides a more reliable performance in the presence of unexpected inputs. However, YOLO has its limitations. For instance, YOLOv1 has a lagging accuracy in comparison to other detection systems and it has poor performance when detecting small objects.

To overcome said limitations, some features were updated in YOLOv3 and a new classifier network was trained. Although the problem of not detecting small objects still exists, the new version provides a considerable improvement in APs performance. Figure 2 benchmarks the performance of YOLOv3

against other training methods. YOLOv3 has a significant performance improvement over YOLOv2, but there are still gaps between the performance of other methods [5].

	backbone	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L
Two-stage methods							
Faster R-CNN+++ [5]	ResNet-101-C4	34.9	55.7	37.4	15.6	38.7	50.9
Faster R-CNN w FPN [8]	ResNet-101-FPN	36.2	59.1	39.0	18.2	39.0	48.2
Faster R-CNN by G-RMI [6]	Inception-ResNet-v2 [21]	34.7	55.5	36.7	13.5	38.1	52.0
Faster R-CNN w TDM [20]	Inception-ResNet-v2-TDM	36.8	57.7	39.2	16.2	39.8	**52.1**
One-stage methods							
YOLOv2 [15]	DarkNet-19 [15]	21.6	44.0	19.2	5.0	22.4	35.5
SSD513 [11, 3]	ResNet-101-SSD	31.2	50.4	33.3	10.2	34.5	49.8
DSSD513 [3]	ResNet-101-DSSD	33.2	53.3	35.2	13.0	35.4	51.1
RetinaNet [9]	ResNet-101-FPN	39.1	59.1	42.3	21.8	42.7	50.2
RetinaNet [9]	ResNeXt-101-FPN	**40.8**	**61.1**	**44.1**	**24.1**	**44.2**	51.2
YOLOv3 608 × 608	Darknet-53	33.0	57.9	34.4	18.3	35.4	41.9

Figure 2. Performance of YOLOv3 Compared to Other Methods

The general idea of YOLO is that the system first divides the full image into an S*S grid graph, where each grid contains part of the data. Then YOLO will predict multiple bounding boxes along with confidence scores. The confidence score indicates the model's confidence in whether the grid contains the object as well as whether it is the target object. Each bounding box contains 4 location data: h(height) and w(width) for the size of the bounding box, and x-y coordinates that represent the centre of the box. YOLO will also generate a class possibility map. The formula is shown below:

$$\Pr(\text{Class}_i|\text{Object}) * \Pr(\text{Object}) * \text{IOU}_{\text{pred}}^{\text{truth}} = \Pr(\text{Class}_i) * \text{IOU}_{\text{pred}}^{\text{truth}}$$

where IoU is the intersection over union between the predicted box and the ground truth, and the term Pr(Classi) ∗ IoU truth pred calculates the confidence. This class possibility is only calculated once per gird regardless of the number of the bounding boxes [6]. After combining the result of the possibility map and the data for each bounding box, the prediction is given in the form of several bounded boxes to indicate different target objects, as shown in Figure 3.

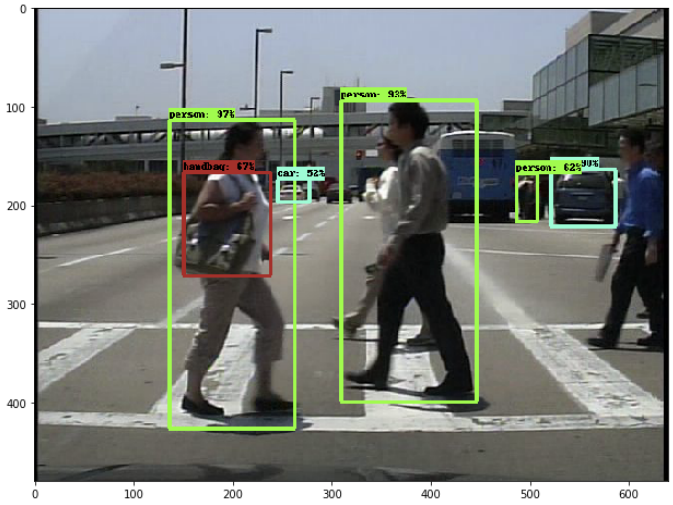

Figure 3. Illustration of Bounding Boxes

K-means Clustering

The project uses K-means clustering to determine the optimal bounding box dimensions. K-means clustering is a popular unsupervised machine learning algorithm that is easy to implement. Unsupervised learning algorithm is a type of learning algorithm where the system is presented with a number of patterns without labels or known outcomes. Then the patterns are grouped into categories through some well-defined guidelines. The main goal of k-means clustering is to group the input data in k groups. “k” represents the number of centroids (or groups), and “means” indicates the the centroids are the averages of the groups. The algorithm finds and puts the given inputs into groups through clustering [8].

Because most bounding boxes have certain height to width ratios, YOLO simplifies the prediction process by predicting offsets from a set of anchor boxes with particular height to width ratios instead. *kmeans.py* takes in the post-conversion annotation file as the input, and uses k-means clustering to output the optimal anchor dimensions based on the training annotations. K-means clustering is implemented using a heuristic algorithm that first randomly picks *k* bounding boxes as the cluster centers, then iteratively calculates the distance of each bounding box to the cluster centers and computes new cluster centers by taking the arithmetic mean of all points within each cluster until the differences between old and new cluster centers are negligible.

Solutions

Environment Setup

We created a virtual machine instance on Google Cloud to do training and inference since the computing power of our laptops is limited. The hardware configuration of our virtual machine is shown below:

- CPU: Intel Xeon 4 cores@2.20GHz
- GPU: Nvidia Tesla T4 with 16GB memory
- RAM: 25GB
- Disk: 100GB SSD

This configuration provides the computing power of a typical entry-level deep learning hardware setup. Because the free credit of a new Google Cloud account is limited to around 400 CAD, we had to be conservative in terms of hardware configuration in order to not exceed the free credit limit for the duration of the project.

Our software environment consists of TensorFlow version 1.13.1 and Keras version 2.2.4. Both of them are at their stable release versions.

Neural Network Model Selection

Our pedestrian detection system is implemented by Keras-yolo3, which is a Keras implementation of YOLOv3. Keras is a Python-based open source neural network library [9]. It has a high capacity to run on top of tensorflow, which is an important tool used widely in the field of deep learning. Because Keras is easy to use and has high build flexibility, it soon gained popularity among users. By mid-2018, Keras had 250,000 users and was ranked top 2 in the deep learning framework selections. Its popularity suggests that Keras is a stable and reliable tool [10].

There are several object detection systems being used in the market, such as YOLO, R-CNN, Spatial Pyramid Pooling (SPP-net), Single Shot Detector (SSD), and MultiGrasp. The solution's performance is largely based on the object detection system selection. An efficient and effective object detection system can save users a lot of time and provide stable and consistent results.

YOLO is short for "You Only Look Once", and it is a popular real-time object detection system. YOLOv3 is its third and latest version. YOLO performs object detection and classification. YOLO supports pre-trained datasets and also allows users to train their own datasets, which provide flexibilities for the users to use the system based on their needs. It is fast, accurate, and is the only popular system which uses the complete image as the training data. The advantage for this is that it will decrease the error significantly and prevent some edge cases. Other object detection systems all use partial image as their training data, which may lead to incomplete object or incorrect circulation problems.

R-CNN is an advance version of CNN which is short for convolutional neural network. The main problem for R-CNN is that it uses multiple classifiers and is thus very time consuming. Figure 4 shows the algorithm for R-CNN. The main algorithm is still CNN, and R-CNN extracts a fixed number of regions instead of unlimited regions used in CNN. R-CNN's approach improves the performance significantly, but speed is still a huge problem. It takes approximately 47 seconds to test one image which completely eliminates the possibility of real-time detection [11]. Based on the motivation behind this project, it is important for the neural network system to detect pedestrian immediately and notify the driver for the potential collision. Thus R-CNN is not suitable for this project.

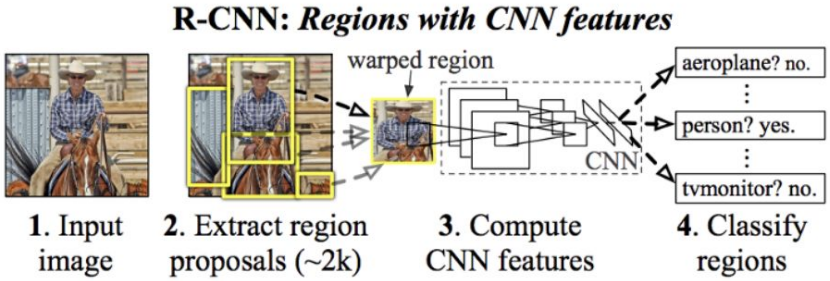

Figure 4. R-CNN Algorithm

Because of the speed problem, R-CNN has two updated versions called fast R-CNN and faster R-CNN. Both versions decease inference time. Table 1 shows the comparison between the three versions.

Table 1. Comparison between R-CNN, Fast R-CNN, and Faster R-CNN

	R-CNN	**Fast R-CNN**	**Faster R-CNN**
Test Time Per Image	50 seconds	2 seconds	0.2 seconds
Speed Up	1x	25x	250x

Spatial Pyramid Pooling (SPP-net) is a new technique at the transition of convolutional layer and fully connected layer in CNN and is made by Microsoft [12]. It was designed to fix the slow speed of CNN. SPP-net calculates the entire image once and uses the result and Selective Search to generate each patch of the CNN representation. The clever part for SPP is that it uses the last convolutional layers' spatial pooling instead of the maximum number of pooling. Figure 5 shows a graphical representation. By doing this, SPP successfully generates the fixed input size for CNN's fully connected layers. The problem of SPP is also pronounced - it is very difficult to perform back-propagation through spatial pooling layer [13].

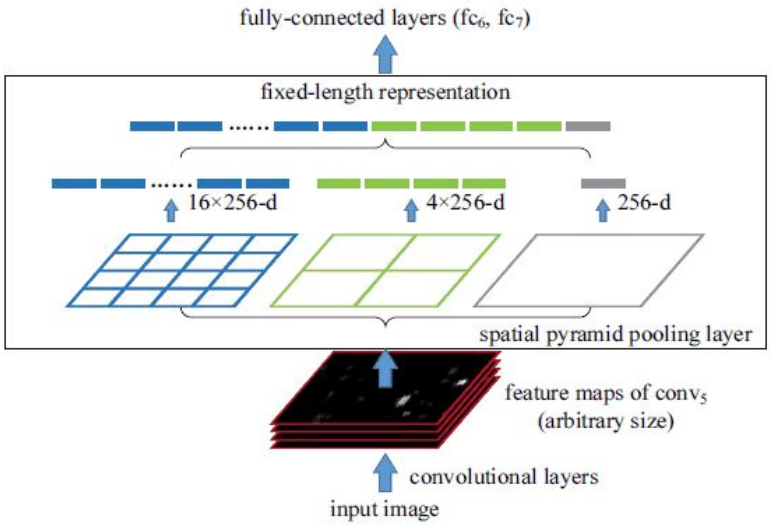

Figure 5. The Core Algorithm in SPP-net

Single Shot Detector (SSD) is another object detection system that uses convolutional neural network. Similar to YOLO, SSD also uses regression based object detectors. SSD runs convolutional neural network once and then calculates a feature map to predict the bounding box. The advantages for SSD are that it is able to handle videos and is quicker than faster R-CNN. The problem for SSD is that it requires higher computation power and is slower than YOLO. SSD is definitely a useful tool for object detection and a good choice for many other usages.

MultiGrasp is similar to the initial version for YOLO, as YOLO based their grid box to bounding box prediction on MultiGrasp. MultiGrasp indicates if there is an object in the provided test data. However, it does not contain information like size, location and boundaries [14]. Therefore it is not suitable for the project.

Based on the comparisons against several other popular object detection systems, YOLO is a good fit for our pedestrian detection system because it is fast, accurate, complete and easy to learn.

Dataset Selection

Among multiple publicly available pedestrian datasets, the Caltech Pedestrian Dataset is one of the most predominant benchmarks today. The Caltech Pedestrian Dataset is large and well-annotated. The data is collected from a 640x480 30Hz video of 10-hours driving through areas with high pedestrian concentration. The dataset annotated about 1000K total frames, 250000 labeled frames, 132K frames with pedestrians, 350K bounding boxes, and 2300 unique pedestrians [15]. It provides a wide range of pedestrian's appearance, pose, and scale; it diverses in lighting, object rotation, and includes occluded objects. Moreover, the dataset includes training and testing data, allowing different training setups.

Data Processing

The training input of the Caltech Pedestrian Dataset consists of several one-minute long videos in .seq format, and its training target/output consists of annotations in .vbb format. Neither the input nor the output can be used directly in YOLO thus conversions are necessary. We used *caltech-pedestrian-dataset-converter* as a starting point and made modifications to fit our needs.

Input data conversion is done in *convert_seqs.py*. We initially saved each frame of the video into a .png file. However, we soon discovered that this approach resulted in upwards of 1.6GB of images for every one-minute video. Given our resource constraints, using all frames for training was not feasible. To reduce training time and memory, we decided to include 1 frame for every 10 original frames. The reasoning behind this is that pedestrians move at a much slower speed compared to vehicles, and through manual comparison we found that every pedestrian is usually present in more than 15 frames (0.5 second). Given the topic of our project, very little training value would be lost from sampling the video frames. With this approach, we were able to reduce the number of input images down to a manageable size.

Annotations are converted in *convert_to_training_input.py*. We then repeated the same procedure for the annotations. The original .vbb annotations contain a lot of extra information that is not needed by YOLO, and we decided to use a text file as the target of our training function, with each row denoting the annotations for an input image. The rows have the format of *image_file_path box1 box2 ... boxN*, with each box having the format of *x_min,y_min,x_max,y_max,class_id*. An example would be "*path/to/img1.jpg 50,100,150,200,0 30,50,200,120,3*\n*path/to/img2.jpg 120,300,250,600,2*". The conversion is done by first using the same sampling rate to get annotations for specific frames, then extracting the bounding box coordinates, and writing it to the text file.

Training Process

Prior to training, we used *kmeans.py* on the training data to get suitable anchor box dimensions. In Figure 6, six optimal anchor box dimensions are computed.

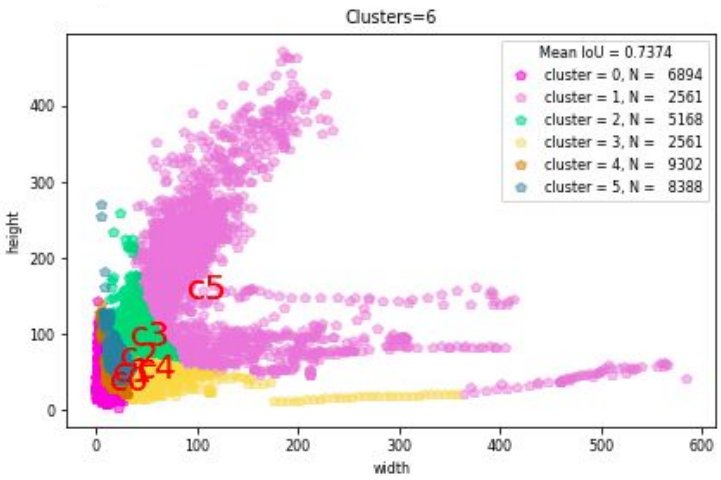

Figure 6. K-means Clustering Result for Anchors

For training, we used *keras-yolo3* as a reference. Using a set of pre-trained weights as a starting point, we freeze the weights in either the first 20 layers or all but the last 2 layers.

In order to measure the accuracy of our models, we compared it with the pre-trained YOLOv3 model, as well as the pre-trained SSD and Faster SNN models. The pre-trained models also take .png files as input, and we are using the post-conversion annotation file as the ground truth.

For consistency, we developed our custom measure of model accuracy to be used on all models for comparison purposes. It is an adaptation of Intersection over Union (IoU). IoU is a popular metric that can evaluate object detection algorithm that outputs bounding boxes as predictions [16]. In order to apply IoU, both the hand-labelled and model-predicted bounding boxes are needed. IoU is essentially the ratio of intersection (area that the two bounding boxes overlap) to union (area encompassed by both bounding boxes). Figure 7 is an illustration of how IoU is calculated.

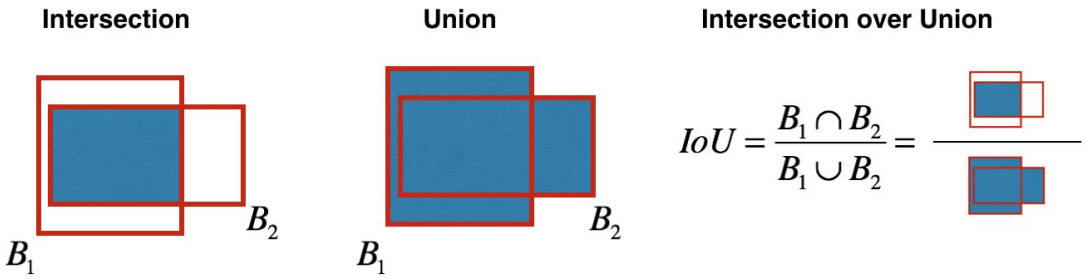

Figure 7. Intersection over Union (IoU)

Object detection cannot be evaluated with a straightforward binary-output evaluator, because the degree to which a prediction is accurate is often fuzzy. Most of the time, the predicted bounding box will overlap with the hand-labelled bounding box but they will not match exactly. The ideal evaluator should have a scoring system that rewards predictions for heavy targets overlap, which IoU does. The maximum score of 1 is achieved only if the prediction and target have the exact x-y coordinates. The value of IoU drops with a decrease in the size of overlap, and with an increase in the non-overlapping area. Figure 8 demonstrates 3 predictions measured by IoU.

Figure 8. IoUs for Various Prediction Accuracy

For our algorithm, we modified IoU so that it also takes confidence score into account. We represent every frame with a 640*480 grid where every coordinate represents a pixel in the frame. For the hand-labelled annotations, the algorithm iterates through every bounding box and marks the target areas with 1's. The algorithm for constructing the ground-truth grid is shown in Figure 9.

```
ground_truth_dict = {}
with open('test_images/train.txt','r') as annotation_file:
    for line in annotation_file:
        split = line.split(' ')
        name = split[0].split('.')[0]
        ground_truth = np.zeros((640, 480))
        for box in split[1:-1]:
            xmin, ymin, xmax, ymax = [int(pix) for pix in box.split(',')[:4]]
            ground_truth[xmin:xmax, ymin:ymax] = 1
        ground_truth_dict[name] = ground_truth
```

Figure 9. Code Snippet on How to Construct the Ground-truth Grid

Figure 10 shows what the grid for the ground truth might look like for a 6*6 image. For clarity, the coordinates occupied by 0 are not shown.

	1	1			
	1	1			
	1	1			1
					1
					1

Figure 10. Ground-truth Grid for IoU Calculations

For our predicted image, we make a similar grid but use the confidence scores as the value, as shown in Figure 11 and Figure 12.

```
output_dict = np.load(dir + filename).item()
detection_grid = np.zeros((640, 480))
name = filename.split('.')[0]
for i in range(len(output_dict['detection_boxes'])):
    ymin, xmin, ymax, xmax = output_dict['detection_boxes'][i]
    xmin = int(xmin * 640)
    xmax = int(xmax * 640)
    ymin = int(ymin * 480)
    ymax = int(ymax * 480)
    score = output_dict['detection_scores'][i]
    box_grid = detection_grid[xmin:xmax, ymin:ymax]
    box_grid[box_grid < score] = score
    detection_grid[xmin:xmax, ymin:ymax] = box_grid
# calculate IOU
if(name in ground_truth_dict):
    intersection = detection_grid[ground_truth_dict[name] == 1].sum()
    union =  ground_truth_dict[name].sum() + detection_grid.sum() - intersection
    iou = intersection / union
    print(name, iou)
```

Figure 11. Code Snippet on How to Construct the Prediction Grid and Calculate the IoU

				0.2	
	0.9	0.9		0.2	
	0.9	0.9			
	0.9	0.9			
					0.7
					0.7

Figure 12. Prediction Grid for IoU Calculations

Then, intersection is calculated by summing up all the values in the prediction grid where the corresponding point in the ground-truth grid is a 1. For this example, intersection score is 0.9*6+0.7*2=6.8.

Union is calculated by summing all values from both grids, then subtraction intersection from it. For this example, union score is (6+3)+(0.9*6+0.2*2+0.7*2)=9.4 and our modified IoU value is 6.8/9.4=0.72. A snippet of the calculations are shown above in Figure 11.

Results: Analysis and Comparison

Comparison against Various Pre-trained Models

Table 2 shows the average IoU and inference time for the pre-trained models. Pre-trained Faster R-CNN model has the highest average IoU of 0.2505, but also takes the longest to inference at 4.913s. Our self-trained YOLOv3 comes second in performance at 0.2107 average IoU, and inferences magnitudes faster at 0.021s per image. Pre-trained SSD has the lowest average IoU of 0.1242, with a slow average inference time of 2.921s.

Table 2. Performance Comparison of Various Pre-trained Models

Model	Average IoU	Average inference time (s)
SSD pre-trained	0.1242	2.921
Faster R-CNN pre-trained	0.2505	4.913
YOLOv3 self-trained	0.2107	0.021

Comparison against YOLOv3 Pre-trained Model

Table 3 shows the average IoU and inference time for the YOLOv3 pre-trained and self-trained models. Both the pre-trained and self-trained models take 0.021s to inference. Our self-trained has a higher average IoU of 0.2107 compared to the 0.1757 average IoU of the pre-trained model.

Table 3. Performance Comparison of Pre-trained and Self-trained YOLOv3 Models

Model	Average IoU	Average inference time (s)
YOLOv3 pre-trained	0.1757	0.021
YOLOv3 self-trained	0.2107	0.021

Validation Loss

Table 4. Training Performance Comparisons

Number Of Training Data	Number Of Epoch	Batch Size	Validation Loss	Training Time
6195	80	8	8.348	~10h
6195	80	16	8.636	~10h
6195	40	8	10.904	~5h
1922	80	8	16.488	~3.5h

We varied the parameters in four different ways to compare the training performance. In Table 4, the best performance occurs when we use 6195 training images with 80 epoches and a batch size of 8, which has a validation loss of 8.348. The validation loss represents the accuracy, the lower the validation loss, the higher the accuracy. However, it takes about 10 hours to train. With a larger batch size, for example, 16, the validation loss slightly decreases. Thus, it is not true that large batch size performs better. Appropriate batch size improves the accuracy and a change of batch size does not affect total training time. Decrease the number of epoch from 80 to 40 faster the training time, but increases the validation loss. People may consider decrease the epoch size to save some time when accuracy is not a prior requirement. A decrease of the number of training data has a dramatically impact on the validation loss. 1922 is approximately one third of 6195, and it doubles the validation losses from 8.348 to 16.488. Even though the training time only take about 3.5 hours, the huge accuracy drops is not acceptable. From the test cases described above and the chart shown in Table 4, number of training data and number of epoch proportionally influence the total training time. The more data we have for training and the more epoch we used to improve weights, the better the accuracy the training result will be. Batch size affects validation loss more, more experiments are required to find a proper batch size for the dataset.

Transfer learning algorithm is used in this project. By applying the algorithm, we need to freeze some layers first. Freeze a layer means we exclude the layer from training [17]. We use the pre-trained data as our starting point and we have a total of 44 layers, which means in the first 42 epoches, we only change the last two layers and leave other layers as the same as the pre-trained model. By doing this, it saves us some time from doing back propagation to all the layers but the last two. It is also better for fine-tuning the model[18]. Figure 13 shows the training and validation loss of the best model discussed above. As we can see from the figure, the graph plateaued in the first 42 epoches because of the action we did above. The reason why we still need the last two layers to be changed is that we still want to update the weights to optimize the model. If we freeze all the layers and train the model for the first 42 epoches, it is a waste of time since nothing has been changed.

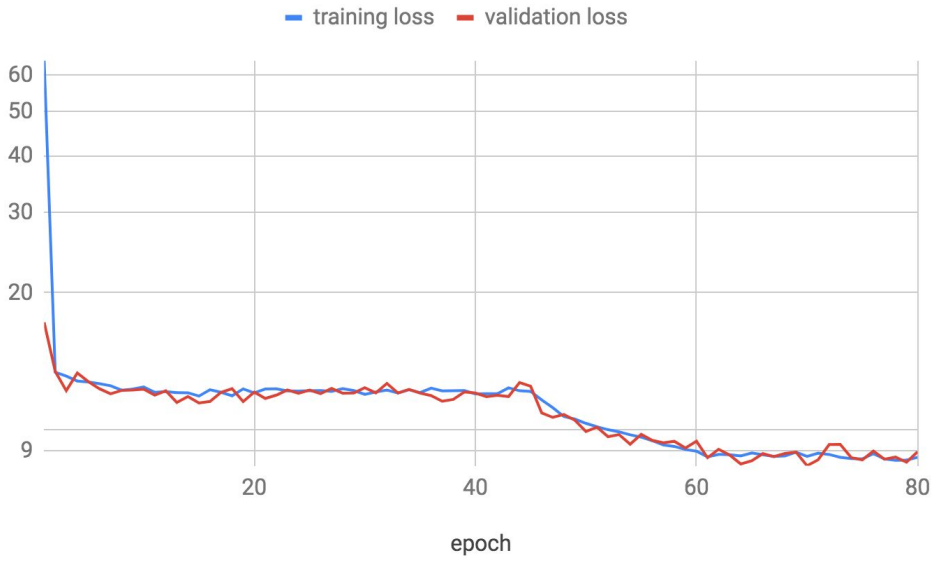

Figure 13 Training and Validation Loss of Best Model

After the 42nd epoch, we unfreeze all the layers and it is clear that both the validation loss and training loss decreases. Figure 14 and Figure 15 illustrate how to freeze and unfreeze the layer. Basically, we have a parameter named *trainable*, it is a boolean value that indicates whether the layer should be trained or not. After we setup the value for the *trainable* parameter, we need to run *compile()* to apply the changes.

```
if freeze_body in [1, 2]:
    # Freeze the darknet body or freeze all but 2 output layers.
    num = (20, len(model_body.layers)-2)[freeze_body-1]
    for i in range(num): model_body.layers[i].trainable = False
    print('Freeze the first {} layers of total {} layers.'.format(num, len(model_body.layers)))
```

Figure 14. Code Snippet on How to Freeze Layers

```
for i in range(len(model.layers)):
    model.layers[i].trainable = True
model.compile(optimizer=Adam(lr=1e-4), loss={'yolo_loss': lambda y_true, y_pred: y_pred})
print('Unfreeze all of the layers.')
```

Figure 15. Code Snippet on How to Unfreeze Layers

Conclusions

Our self-trained YOLOv3 model with custom anchor sizes can detect pedestrians in images and videos. It was found that a model trained with 6195 images, 80 epochs and a batch size of 8 can achieve better detection accuracy than the pre-trained YOLOv3, pre-trained Faster R-CNN and pre-trained SDD models. Moreover, the self-trained model achieved a throughput of around 50 FPS on videos with a resolution of 416*416 using one Nvidia Tesla T4 GPU. The experimental results validated the efficiency and accuracy of our self-trained model.

It was found that although training time appears to increase linearly with both the amount of training data and the number of epochs, they both improve accuracy in terms of validation loss. Doubling the batch size also resulted in a small decrease in performance.

From testing the self-trained model with custom video inputs, it was shown that the pedestrian's posture and attire along with the illumination and occlusion have a major impact on the detection process. In Figure 16, a screenshot of pedestrian detection using our custom video input is provided.

Our proposed pedestrian detection model can be integrated into real-life applications such as traffic safety and autonomous driving. In the future, the self-trained model can be improved by increasing the number of training sets with more diverse pedestrian images.

Figure 16. Screenshot of Custom Video Input

References

[1] “Car accident facts and figures in Canada,” thestar.com, 10-May-2018. [Online]. Available: https://www.thestar.com/sponsored_sections/2018/05/09/car-accident-facts-and-figures-in-canada.html. [Accessed: 22-Jan-2019].

[2] “Pedestrian Detection in Crowded Scenes.” [Online]. Available: https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/leibe-cvpr-05.pdf. [Accessed: 22-Jan-2019].

[3] “Convnet: Implementing Maxpool Layer with Numpy,” Convnet: Implementing Maxpool Layer with Numpy - Agustinus Kristiadi's Blog. [Online]. Available: https://wiseodd.github.io/techblog/2016/07/18/convnet-maxpool-layer/. [Accessed: 31-Mar-2019].

[4] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection,” 2016. [Online]. Available: https://pjreddie.com/media/files/papers/yolo_1.pdf. [Accessed: 30-Mar-2019].

[5] M. Chablani, “YOLO - You only look once, real time object detection explained,” Towards Data Science, 21-Aug-2017. [Online]. Available: https://towardsdatascience.com/yolo-you-only-look-once-real-time-object-detection-explained-492dc9230006. [Accessed: 31-Mar-2019].

[6] M. Menegaz, “Understanding YOLO,” Hacker Noon, 28-Mar-2018. [Online]. Available: https://hackernoon.com/understanding-yolo-f5a74bbc7967. [Accessed: 31-Mar-2019].

[7] M. Hollemans, Real-time object detection with YOLO. [Online]. Available: https://machinethink.net/blog/object-detection-with-yolo/. [Accessed: 31-Mar-2019].

[8] A. Trevino, “Introduction to K-means Clustering,” Oracle DataScience.com. [Online]. Available: https://www.datascience.com/blog/k-means-clustering. [Accessed: 31-Mar-2019].

[9] “Keras: The Python Deep Learning library,” Home - Keras Documentation. [Online]. Available: https://keras.io/. [Accessed: 31-Mar-2019].

[10] “Why use Keras?,” Why use Keras - Keras Documentation. [Online]. Available: https://keras.io/why-use-keras/. [Accessed: 31-Mar-2019].

[11] R. Gandhi, “R-CNN, Fast R-CNN, Faster R-CNN, YOLO - Object Detection Algorithms,” Towards Data Science, 09-Jul-2018. [Online]. Available: https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e. [Accessed: 31-Mar-2019].

[12] Tsang, “Review: SPPNet -1st Runner Up (Object Detection), 2nd Runner Up (Image Classification) in ILSVRC...,” Medium, 01-Sep-2018. [Online]. Available: https://medium.com/coinmonks/review-sppnet-1st-runner-up-object-detection-2nd-runner-up-image-classification-in-ilsvrc-906da3753679. [Accessed: 31-Mar-2019].

[13] “Zero to Hero: Guide to Object Detection using Deep Learning: Faster R-CNN,YOLO,SSD,” CV, 28-Dec-2017. [Online]. Available: https://cv-tricks.com/object-detection/faster-r-cnn-yolo-ssd/. [Accessed: 31-Mar-2019].

[14] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection,” 2016. [Online]. Available: https://pjreddie.com/media/files/papers/yolo_1.pdf. [Accessed: 30-Mar-2019].

[15] P. Dollar, C. Wojek, B. Schiele, and P. Perona, “Pedestrian Detection: A Benchmark.” [Online]. Available: https://pdollar.github.io/files/papers/DollarCVPR09peds.pdf. [Accessed: 30-Mar-2019].

[16] “Intersection over Union (IoU) for object detection,” PyImageSearch, 24-Jun-2018. [Online]. Available: https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/. [Accessed: 07-Apr-2019].

[17] P. Jay, “Transfer Learning using Keras,” Medium, 15-Apr-2017. [Online]. Available: https://medium.com/@14prakash/transfer-learning-using-keras-d804b2e04ef8. [Accessed: 06-Apr-2019].

[18] “Keras FAQ: Frequently Asked Keras Questions,” FAQ - Keras Documentation. [Online]. Available: https://keras.io/getting-started/faq/#how-can-i-freeze-keras-layers. [Accessed: 06-Apr-2019].