ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

UNIVERSITÉ DU QUÉBEC

SYS843

Réseaux de Neurones et Systèmes Flous

Études Expérimentales: Real-Time Object Detection

PRÉSENTÉ À: Ismail Ben Ayed

PAR

Bozan XU - XUXB02079305

MONTRÉAL, 23 Décembre 2019

Table of Contents

Table of Figures

Introduction

In recent years, deep learning has been actively applied in various fields including medical imaging, autonomous driving, and social media services. The development of sensors and GPU along with deep learning models has accelerated research into autonomous vehicles based on deep neural network architectures. An autonomous vehicle with self-driving capability without human intervention must accurately detect other cars, pedestrians, and different traffic signs in real time to ensure safe and correct control decisions. To detect such objects, various sensors such as cameras, lidars, and radio detectors are generally installed on autonomous cars. Among these sensors, the camera sensor accurately identifies objects based on texture and color features and is more cost-effective than other sensors. In particular, object detection using cameras is becoming more and more important in the industry as it achieves a higher level of accuracy than the human eye in terms of object detection, and consequently it has become an essential method in autonomous navigation systems.

A real-time object detection algorithm for autonomous vehicles should satisfy two conditions. First, it requires a high detection accuracy of objects on the road. Second, a real-time inference speed is essential for rapid response. Deep-learning based object detection algorithms, which are indispensable in autonomous vehicles, can be classified into two categories: two-stage and one-stage detectors. Two-stage detectors, such as Faster R-CNN and R-FCN, conduct a first stage of region proposal generation, followed by a second stage of object classification and bounding box regression. These methods are generally more accurate but have longer inference speed. From an analytical point of view, these algorithms are, while accurate, very computationally intensive, such that they are too slow for real-time applications, and simply do not run on embedded

systems. One-stage detectors, such as SSD and YOLO, on the other hand, conduct object classification and bounding box regression concurrently without a region proposal stage. These methods are faster, but achieves slightly lower accuracy.

At the beginning of the semester, I fixed the objective of building an autopilot system that could detect objects from the point of view of a self-driving car. Specifically, I wanted to build and train a model that will be used for autonomous driving. As it was concluded in my literature review, the superior algorithm for real-time object detection is YOLOv3. It scores 10 point higher in mAP-50 and runs three times faster than competitor SSD.

The rest of this report is organized as follows. In the next section, I will briefly go over the previously studied YOLO algorithm. I will summarize the evolution of the algorithm and highlight some of the key differences between each generation. Following that section, I will present the specific procedures I used to build a YOLOv2 and a YOLOv3 model. This section will shed light on the datasets I used, the classifier backbone I chose, and the training steps. To conclude the report, I will discuss the significance of observations and add my interpretation of the trade-off between speed and accuracy to validate the theories studied in the literature review.

Summary of studied techniques

YOLO

YOLO's architecture looks like any other CNN. The network comprises of 24 convolutional layers followed by two fully connected layers. The alternating use of 1x1 reduction layers to reduce the depth of the features space followed by a 3x3 convolutional layer was inspired by the GoogLeNet (Inception) model [1].

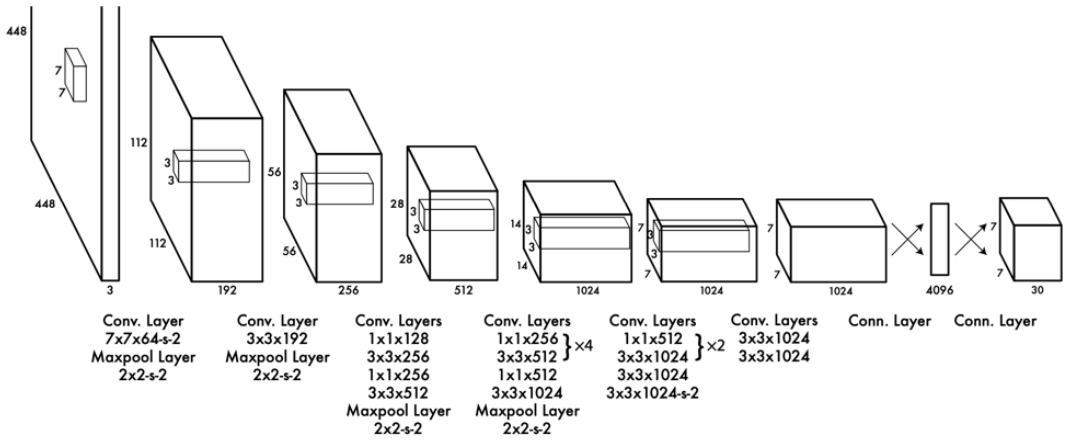


Figure 1: YOLO architecture

As we have seen in the literature review, YOLO detects objects by dividing an input image into a 8x8 grid where each grid cell predicts only one bounding box. The feature map of the YOLO output layer outputs bounding box coordinates, object class, and a class confidence score for each of the predicted bounding boxes, hence enabling YOLO to detect multiple objects with a single inference. Therefore, the time it takes to make an inference is much lower than that of two-staged methods.

Figure 2: YOLO detecting Santas.

However, owing to the processing of the grid unit and limited bounding boxes, localization errors are large and the accuracy is not top tier, especially for objects that are close to each other. As illustrated in Figure 2, there are nine Santas in the lower left corner, but YOLO can only detect five. Thus, YOLO is unsuitable for autopilot applications.

YOLOv2

To address these problems, YOLOv2 was proposed. YOLOv2 is the second version of YOLO with the objective of improving the detection accuracy significantly while making it faster.

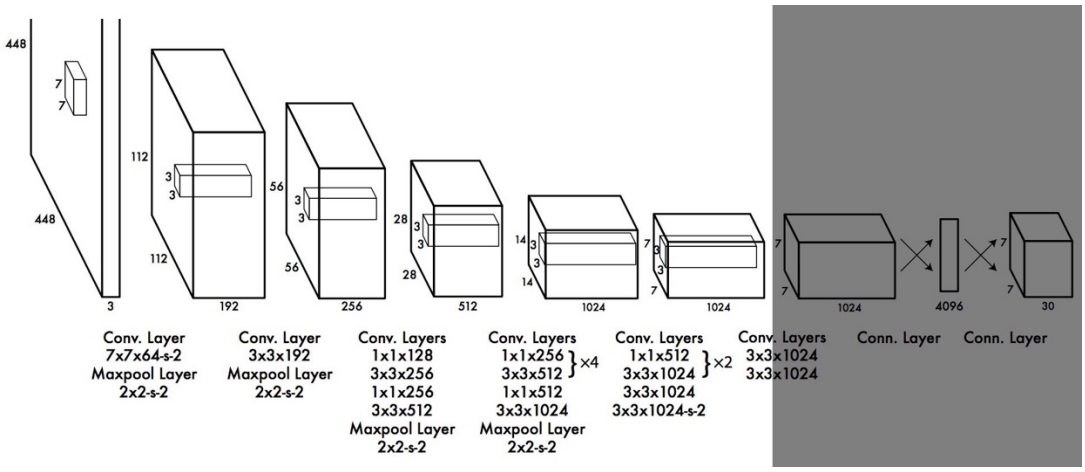


Figure 3: The fully connected layers in YOLO are removed.

Initially, YOLO makes arbitrary guesses on the shape of the bounding boxes. These guesses may work well for some objects but badly for others. In the real-life domain, the bounding boxes are not arbitrary. Cars have very similar shapes and pedestrians have similar aspect ratios. To remedy this problem, Redmon et al ran k-means clustering to find five anchor boxes that have the best coverage for the training data.

Furthermore, the fully connected layers that predict the bounding boxes are replaced with three 3x3 convolutional layers that form a passthrough module that brings features from a higher resolution layer directly to the detector. This change allows YOLOv2 to detect some of the smaller objects.

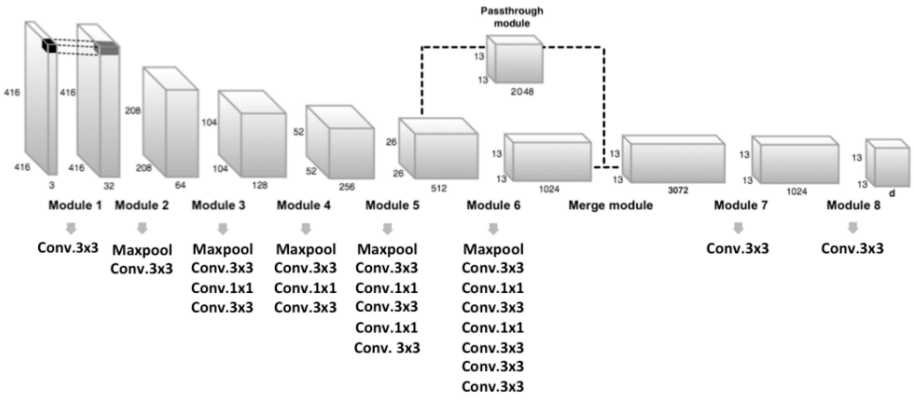


Figure 4:YOLOv2 architecture featuring a passthrough layer.

To further improve the accuracy, YOLOv2 introduces batch normalization at every convolution layer and a high resolution classifier by retuning the classifier with 448x448 pictures. However, the detection accuracy for small or dense objects is still low. Therefore, YOLOv2 is may not be suitable for autonomous driving applications, where traffic signs and lights may be small.

YOLOv3

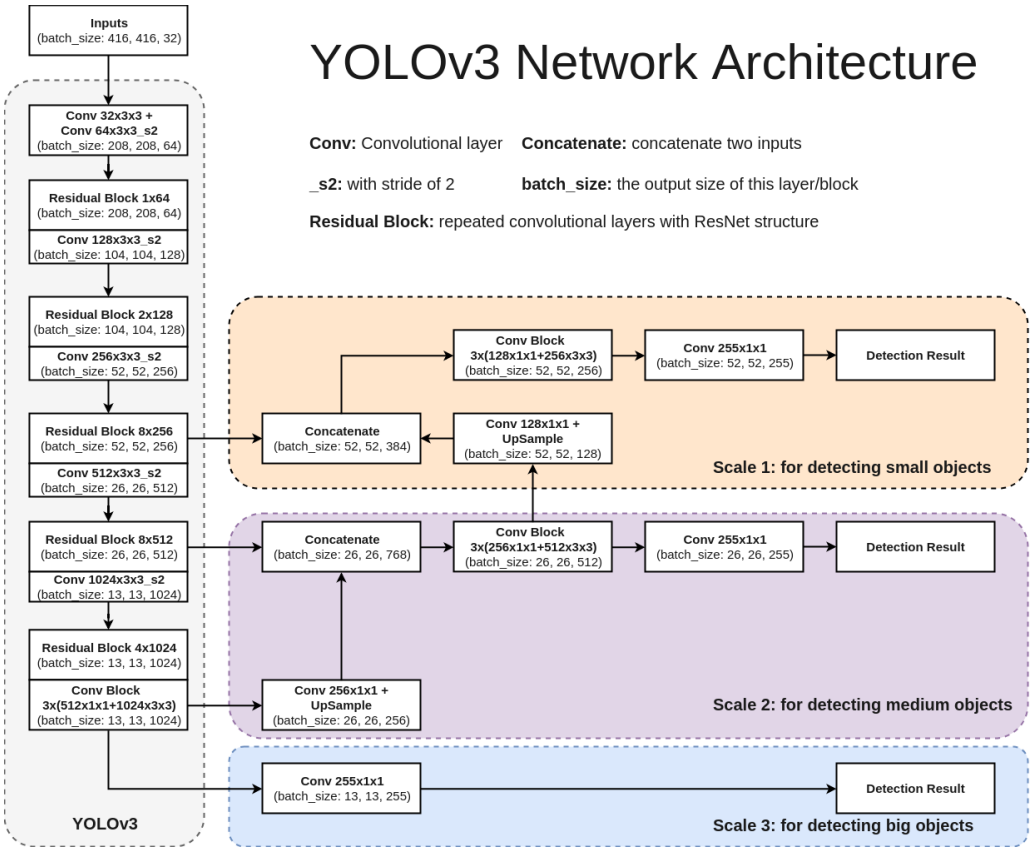


Figure 5: YOLOv3 architecture.

To overcome the disadvantages of YOLOv2, YOLOv3 was proposed. YOLOv3 is built on top of a new feature extractor network, named Darknet-53. It is a hybrid of successive 3x3 and 1x1 convolutional layers and residual network (ResNet). YOLOv3 applies residual blocks to solve the vanishing gradient problem of deep networks and uses an up-sampling and concatenation method that preserves fine-grained features for small object detection. It is done by going back

by two layers, up-sampling by 2 and concatenating the previous layers. The most salient feature is the detection at three different scales in a similar manner that is used in a feature pyramid network. YOLOv3 applies 1x1 kernels on feature maps at 3 different places in the network. At those 3 places, the dimensions of the input image get downsampled, respectively, by 32, 16, and 8. This allows YOLOv3 to detect objects with various sizes. To be more specific, the YOLOv3 network takes an input image and outputs bounding box coordinates, objectness score, and class scores from three detection layers. The predictions made at all three layers are concatenated and processed by non-maximum suppression. Because YOLOv3 is a fully convolutional network consisting only of small-sized convolution filers of 1x11 and 3x3, the inference speed is as fast as YOLOv2. Therefore, in terms of the trade-off between accuracy and speed, YOLOv3 is the most suitable for autopilot applications. In fact, it is widely used in research.

Tiny-YOLOv3

Another interesting algorithm for real-time object detection is a derivative of YOLOv3. It is often not considered as a state-of-the-art model as its accuracy drops by about 20 mAP on the MS COCO Dataset in comparison with YOLOv3. Tiny-YOLOv3 has a reduced number of convolutional layers. Its basic structure has only 7 convolutional layers, and features are extracted by using a small number of 1x1 and 3x3 convolutional layers. Tiny-YOLOv3 uses a pooling layer instead of YOLOV3's convolutional layer with a step size of 2 to achieve dimensionality reduction.

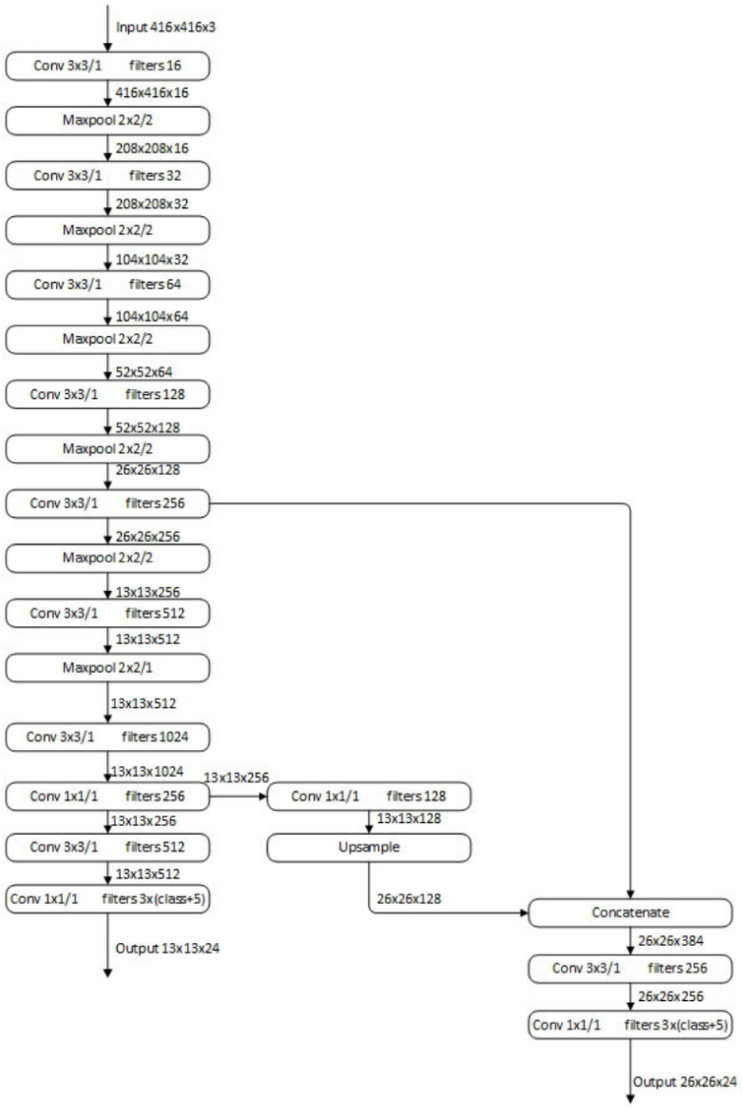


Figure 6:Tiny-YOLOv3 architecture.

With these changes, the model is about 10 times faster and lighter than YOLOv3. I am curious about the performance of Tiny-YOLOv3 as it is probably the only YOLO that can run in real time on cheap embedded systems.

Methodology

Tools

The main programming languages used to build this project is Python 3.7. The reason for that is because it is well documented and compatible with many machine learning frameworks, such as TensorFlow 2.0, PyTorch, and Keras. I did not end up using any of these frameworks, however, as AlexeyAB did a terrific job at building DarkNet in C and his work is now used by many. The rest of the code is written in Python for its ease of use and prototyping speed.

Due to the large number of data that needed to be processed, I executed most of the heavier code on a Google Colab notebook. The hardware I was lent had these specs:

- CPU: Single Core Hyperthreaded Xeon Processors @2.3GHz
- RAM: 13GB
- GPU: NVIDIA Tesla K80, 2496 CUDA cores, 12GB GDDR5 VRAM

The operating system is a Ubuntu 18.04.

Dataset

Darknet-53 is the backbone feature extractor used in the YOLOv3 paper. Darknet-53 is pre-trained on ImageNet 1000 class classification.

Since the objective of this project is to explore autonomous driving, I chose to train YOLOv3 on the Berkeley DeepDrive (BDD100k) Dataset. The BDD100k dataset contains videos from diverse locations around the United States, in a wide range of weather conditions and settings such as, rainy, overcast, sunny, at night, and during the day. Most importantly, for this project,

this dataset includes a hundred thousand still frames extracted from these videos along with bounding boxes and labels for 10 classes of objects, segmentation, and lane lines.

And these are the classes:

Classes	
Traffic light	Truck
Traffic sign	Rider
Car	Bike
Person	Motor
Bus	Train

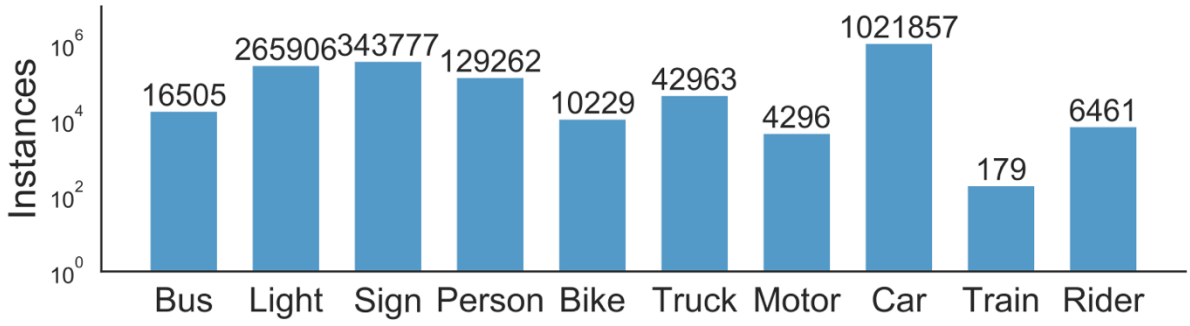


Figure 7: Statistics of different types of objects.

The training, validation, and testing sets were split in a ratio of 7:1:2, but because there were some missing labels, I ended up with a dataset that looked like this:

	Number of Images
Training	69 863
Validation	10 000
Testing	20 000

The enormous size of these datasets was the reason I needed to lend a GPU from Google.

Evaluation criteria

For real-time applications, such as autonomous driving, inference speed and quality of detections are the two most important metrics. Inference speed is the time it takes for a trained neural network to apply its capabilities to infer information about new data. To collect that information, the execution time will be printed out for every input image. To be more precise, the average time taken to infer 10000 images will be recorded. The inverse of this quantity is FPS. As for quality of detection, it will be measured in terms of mean average precision (mAP) on the testing set.

Implementation

CUDA

The GPU that I was lent is a NVIDIA Tesla K80. To be able to use it, I needed to make sure that I installed the right version of cuDNN for the right version of CUDA. To check the CUDA release version on Google Colab, I run

```
/usr/local/cuda/bin/nvcc --version
```

Because CUDA 10.0 is preinstalled on the Google Colab runtime, I downloaded the corresponding version of cuDNN from the NVIDIA website and uploaded it on my Google Drive. I then unzipped the cuDNN files from My Drive directly to the CUDA folder in the VM:

```
tar -xzvf gdrive/My\ Drive/cuDNN/cudnn-10.0-linux-x64-v7.5.0.56.tgz -
C /usr/local/
```

Darknet

As mentioned above, the backbone feature extractor that was used is a slightly modified version of the Darknet found at https://github.com/AlexeyAB/darknet/. It is an open source neural

network framework written in C and CUDA. It is easy to install and supports GPU computations. To build darknet, I simply run:

```
cd darknet && make
```

Pre-trained weights to the backbone network is readily downloadable using the command

```
wget https://pjreddie.com/media/files/darknet53.conv.74
```

Image Acquisition

The images from the BDD dataset are downloadable with a student account. For this project, I only needed

- bdd100k_images.zip
- bdd100k_labels_release.zip.

The BDD dataset has its own labelling and it is not compatible with the format of Darknet. Thus, some processing needs to be done to extract the relevant information for retraining.

Image Preparation

In order to reformat BDD labels for darknet, I used a function provided by sanwong15. The script resizes the bounding boxes' dimensions and converts the labels from the .json file to a .txt file for each image, formatted in the correct darknet format.

The scripts are found in the Annexes.

Configuration

The configuration file `YOLOv3_BDD.cfg` contains the architecture of YOLOv3. We find the implementation of each layer in the backbone Darknet as well as the YOLOv3 fully connected layers. For training, we set the batch size to 64 and subdivisions to 16, but for validation and evaluation, we set batch size and subdivisions to 1.

I created a file BDD.data that references the 10 classes used for training, the training, validation and test files, the list of names, as well as a location where backups of the training weights will be saved. BDD.data is referenced in the Annexes.

Loading Dataset

Since I worked on Google Colab, the only way to use the dataset was to upload a zipped folder containing each set of images to the VM. To unzip the folders, I run:

```
unzip /content/images.zip -d /content/yolov3/
```

Training

The training of the fully connected layers started with random weights and was done in 35000 iterations with a fixed learning rate of 0.0001, a batch size of 64 and subdivisions of 16. I used the usual momentum of 0.9 and a decay of 0.0005 for backpropagation. For this project, the IoU threshold was set to 0.75 for all classes. The weights were backed up at every 1000 iterations in a backup folder for validation. To start the training, I run:

```
./darknet detector train cfg/BDD.data cfg/YOLOv3_BDD.cfg darknet53.conv.74
```

Because Google Colab only allows a maximum continuous runtime of 12h, I save the training weights in a backup folder on my Google Drive, so I can resume training 12h later.

Evaluation

For validation, I first run the command:

```
./darknet detector valid cfg/BDD.data cfg/YOLOv3_BDD.cfg
YOLOv3_BDD.weights
```

Then, I used a script that computes the bounding box IoU to calculate the mAP for each class that can be found on https://github.com/ucbdrive/bdd-data by running the command:

```
python evaluate.py det gt_bdd_val.json ../results/bdd_results.json
```

For the testing set, I run the same commands, but change a line in the BDD.data file to indicate the use of the testing set rather than the validation set. Because the testing set is not annotated, it had to be evaluated on the [BDD evaluation server].

Execution

For detection on a single image, I run the command

```
./darknet detector test cfg/BDD.data cfg/YOLOv3_BDD.cfg YOLOv3_BDD.weights
data/example.jpg
```

It takes an input image from the `data/` folder and runs the `test` function to make predictions on the image. It then saves the image in a .jpg file. Because I am working on Google Colab, I used some helper functions to interface and display images on Google Colab. The `imShow` function can be found in the Annexes. To display the predictions on the VM, I run the command:

```
imShow('predictions.jpg')
```

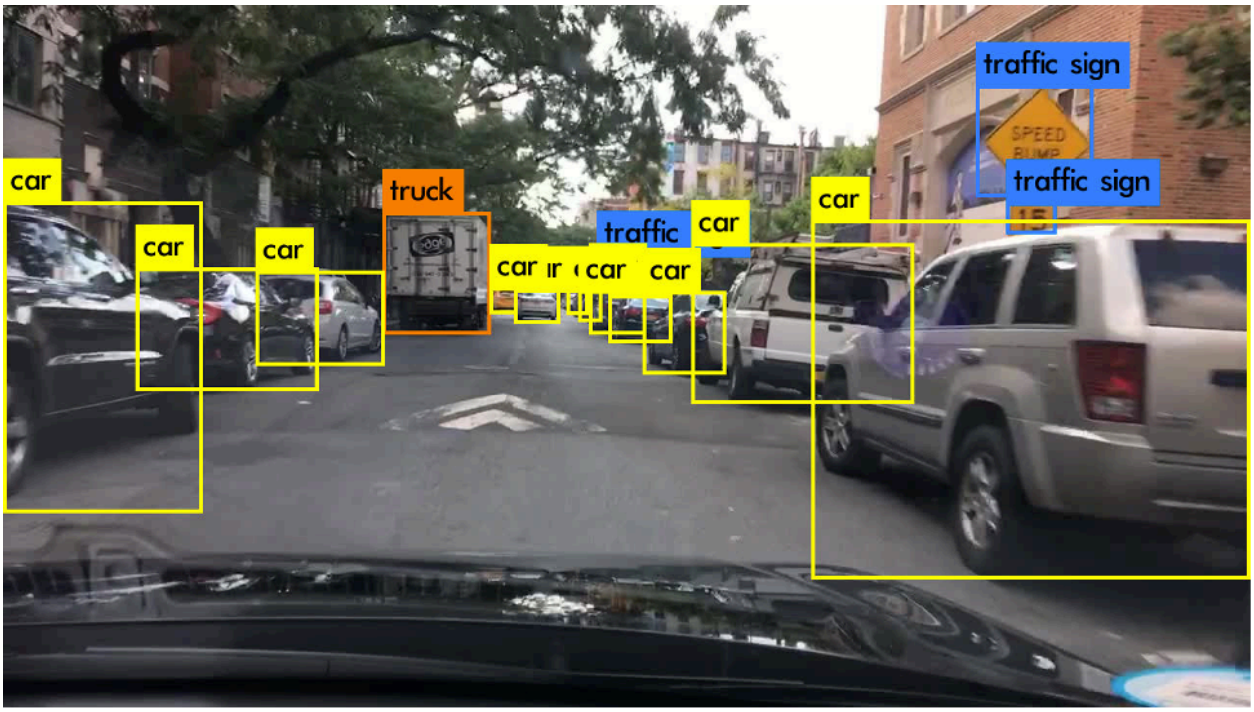


Figure 8: Detection example.

Experimental Results

Example Detection

Take the image in Figure 8, on Google Colab, these are the detection results:

```
data/example.jpg: Predicted in 0.028163 seconds.
truck: 90%
traffic sign: 88%
traffic sign: 80%
traffic sign: 70%
car: 93%
car: 92%
car: 92%
car: 90%
car: 88%
car: 87%
car: 87%
car: 84%
car: 83%
car: 74%
car: 71%
car: 66%
```

Figure 9: Example scores.

The objects are ranked in order of confidence score. Although I did not display the confidence score on each predicted bounding box, we see that even when objects are partially hidden, the detection still works. In fact, the bounding box annotations follow the object very closely.

Inference Speed

For inference speed, I measured the time YOLOv3 took to make predictions on the 10000-image validation set. It took 312 seconds to finish the task, which means that on average, it needs about 0.03 seconds to infer an image, as we can see from Figure 10. These numbers translate to about 32 FPS on the NVIDIA Tesla K80, which runs at around 2.91 teraflops. If I were to build this detection system on an embedded NVIDIA Jetson Nano that runs at 472 gigaflops, I would expect an FPS of 5.2. This model would not run in real-time on embedded systems.

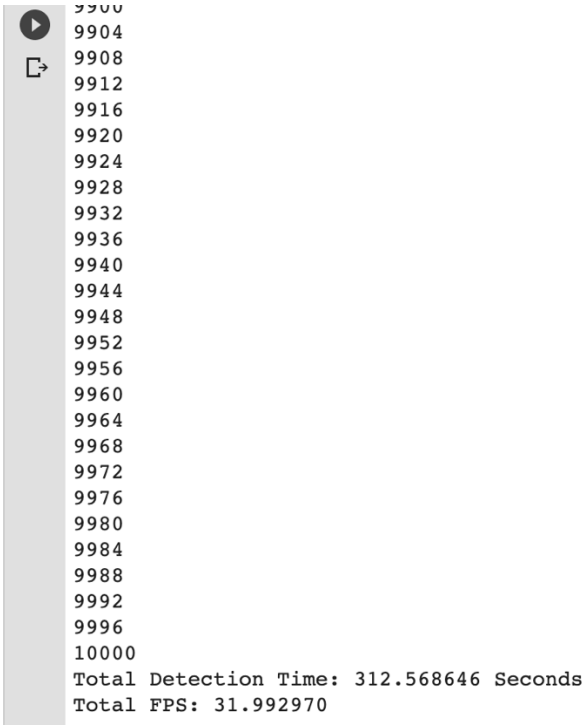


Figure 10: Detection Time.

Quality of Detection

After uploading the .json file containing the results, I get an mAP of 15.44, which is, quite frankly, pretty low considering the high quality detection shown in Figure 8.

```
AP : 8.37 (bike)
AP : 32.82 (bus)
AP : 32.96 (car)
AP : 3.53 (motor)
AP : 12.52 (person)
AP : 6.22 (rider)
AP : 6.06 (traffic light)
AP : 19.46 (traffic sign)
AP : 0.00 (train)
AP : 32.46 (truck)
[8.368295420802772, 32.82022967768842, 32.962338308273596, 3.528317480713093, 12.521039
55706777, 6.221396343004272, 6.064563796458698, 19.1457189144825003, 0.0, 32.4578699458
3501]

8.37 32.82 32.96 3.53 12.52 6.22 6.06 19.46 0.00 32.46

mAP 15.44
```

It seems like the way the mAP was calculated here is by summing the AP for each class and dividing over the number of classes. This number seems inexact, as there are many more instances of cars than trains in the dataset.

Train Detection

What caught my eye first is the 0.00 AP on detecting trains. Looking back at Figure 7, we see that there are only 179 instances of trains in the whole 100000 dataset, whereas cars occur 1021857 times, as expected for an autonomous driving dataset. So, I ran detection on a couple of photos containing trains and these are the results:

Figure 11: Train detection.

In Figure 11, on the left, there is a cargo train in the background, but the model only detects the people in front of the train. This tells us that the huge class imbalance in the dataset makes the model unable to detect trains and, in fact, as illustrated in the image on the right, the model thinks that the very obvious train is a car with a confidence score of 64%.

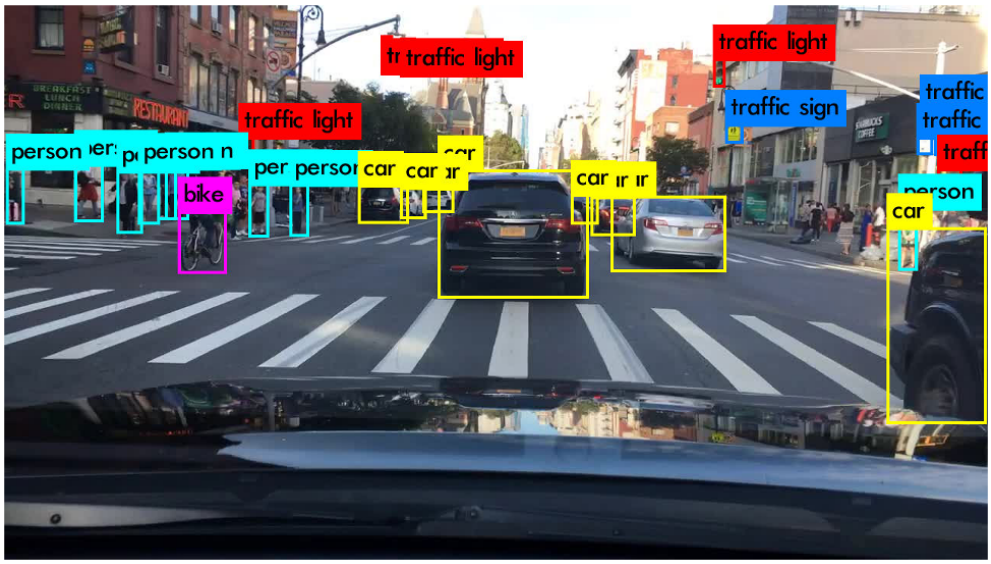


Figure 12: Detection in crowded area.

People Detection

Moving on to people detection, in Figure 12, we see that on the center left, a lot of people are detected, but in higher density zones, it only boxes the person standing at the front and leaves the others undetected. As we can see on the center right, as people become smaller than a certain threshold, the model does not detect them, but that might have to do with the IoU threshold of 0.75 that I used to lower the numbers of false positives. It also seems like the maximum distance for a person to be detected is the distance of a street intersection. This should also account for the low AP of 12.52 on people detection, as in many cases, the roads are empty, but the streets are filled with pedestrians in the distance.

Figure 13: Original photo of crowded area.

In Figure 14, we see that our model confuses two people walking side by side as one person. Personally, I do not think that this is a very serious problem, because it was able to detect that there were, in fact, someone in front of the car, and that it should probably not run over that person. But this case also lowers the mAP. I would even argue that this detector detects objects that are tough for humans to discern. Still in Figure 14, we see that there is a person detection on the left, but I can barely see a shadow.

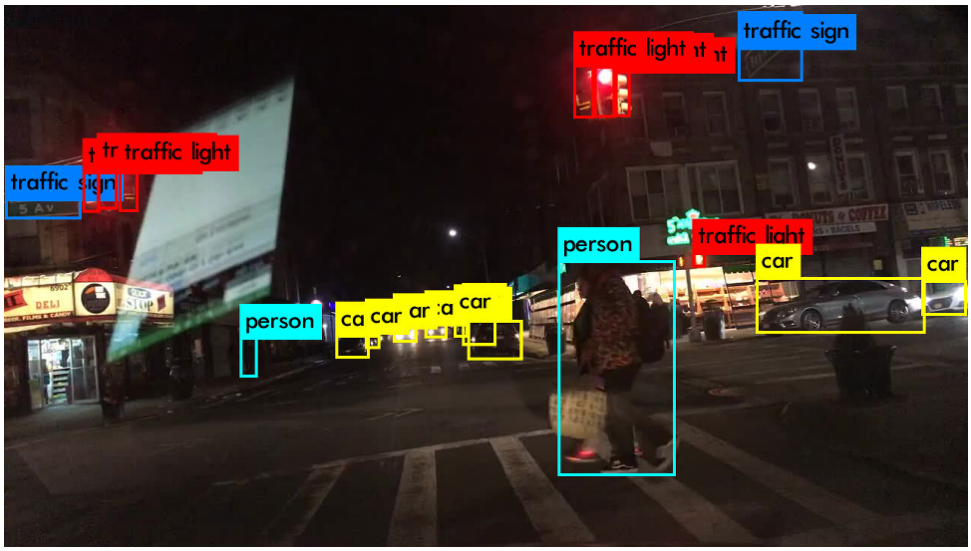
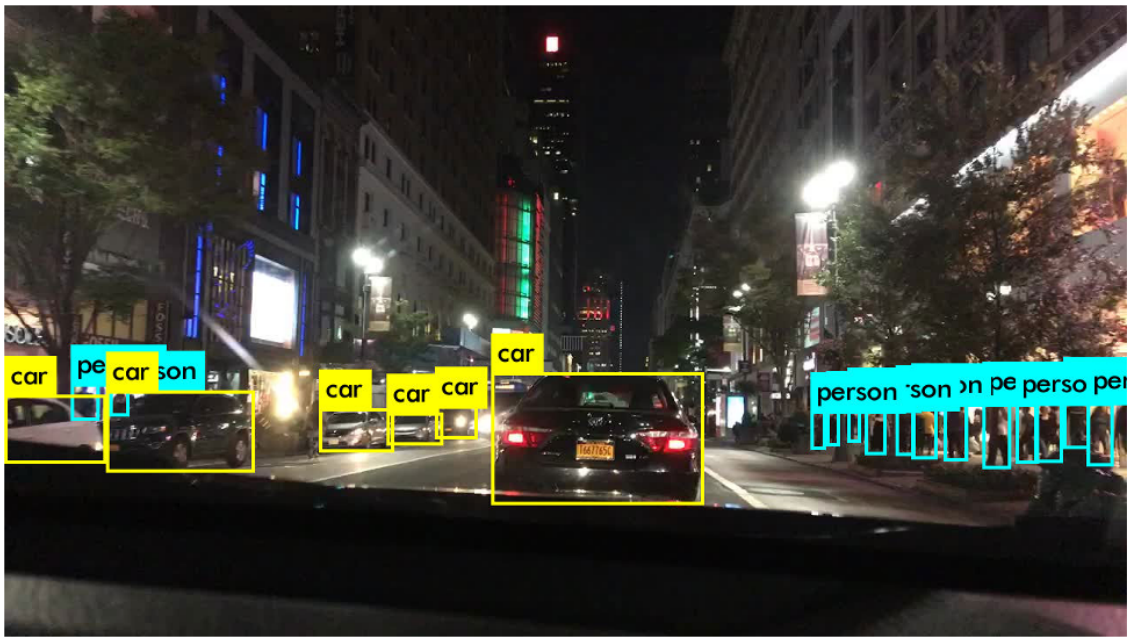


Figure 14: Two people confused as one.

Nighttime Detection

As for detection at night, the model works very well in low light situations. We see from Figure 15 that the model detected two people on the left out of the three that I can see. And, on the right, it drew boxes around many individual pedestrians, but again, there is an instance where it predicted two people as one person.



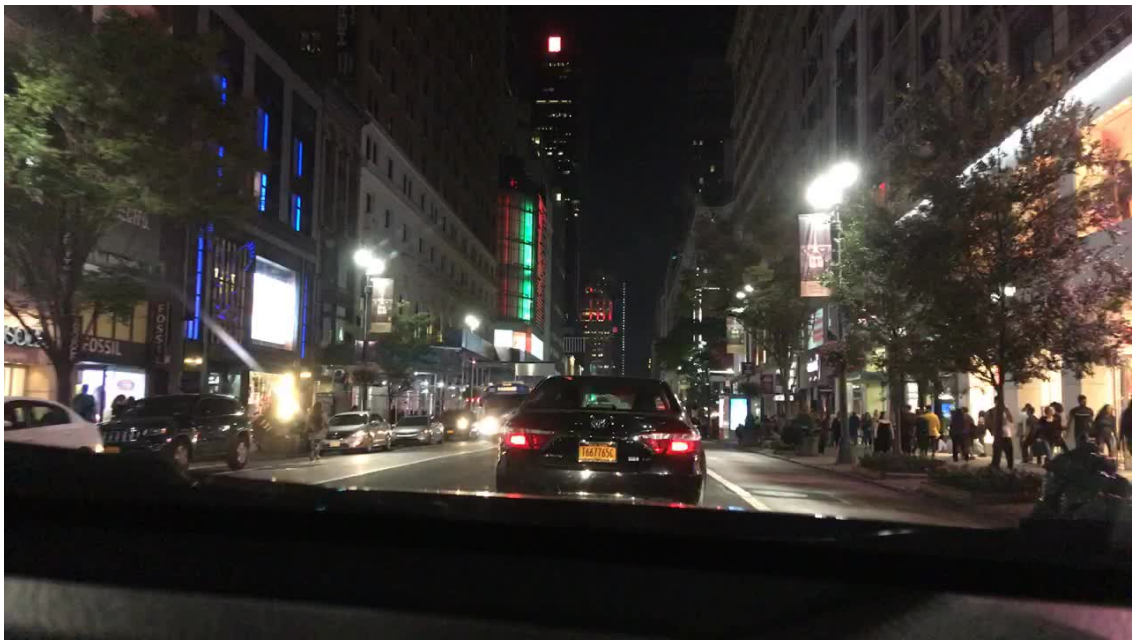


Figure 15: Nighttime detection.

Detection with Motion Blurring

Since the camera is onboard of a car in motion, the still images are sometimes blurred, especially in low light situations. In Figure 16, I cannot tell if there is a car on the left (confidence score of 50%), so it might be a false positive, but in general, it does not seem like bigger objects are affected a lot by the blurring. Smaller objects might be affected more, because a blurring displacement on smaller objects have bigger impact on its image.

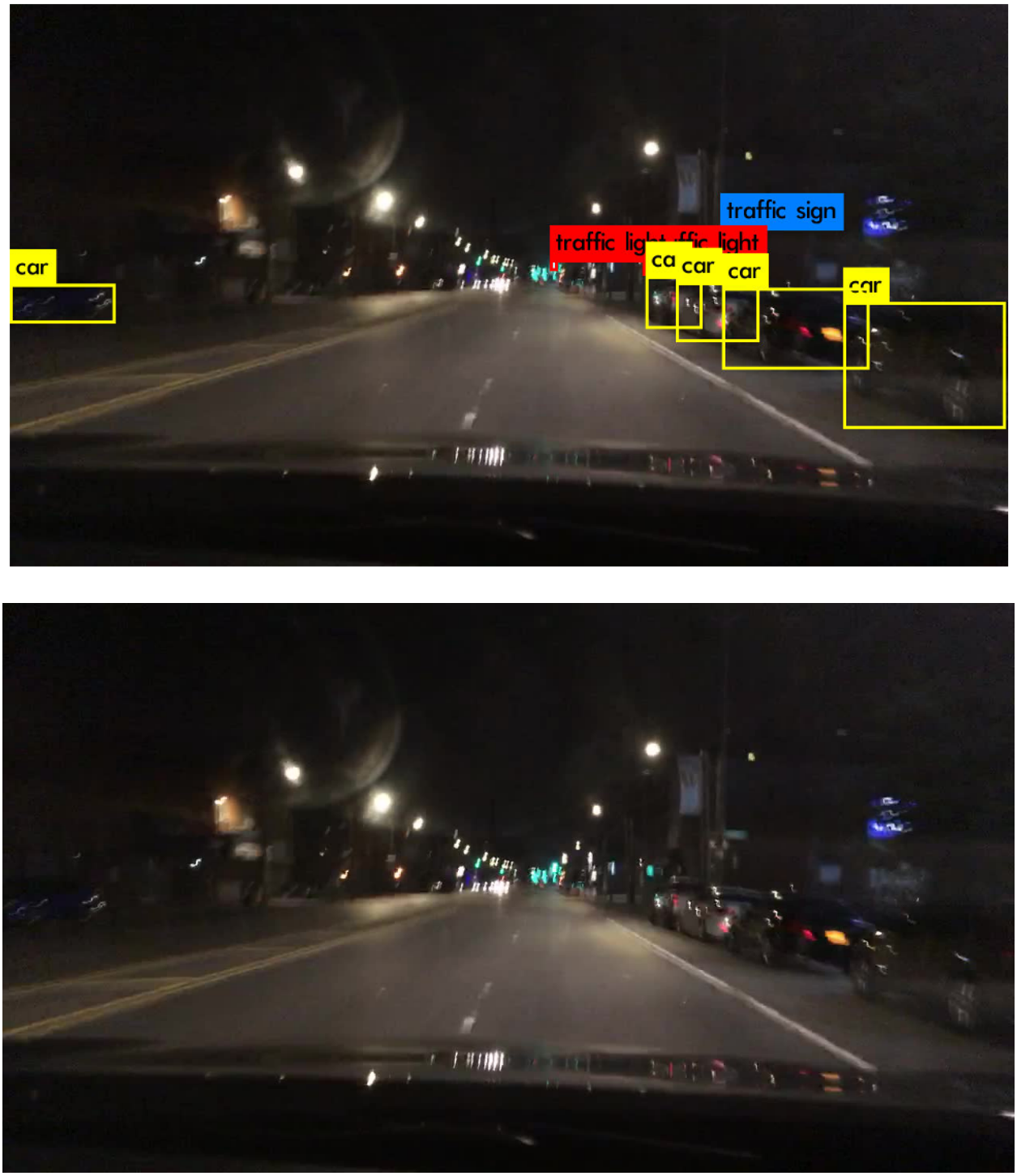


Figure 16: Motion blur on the image.

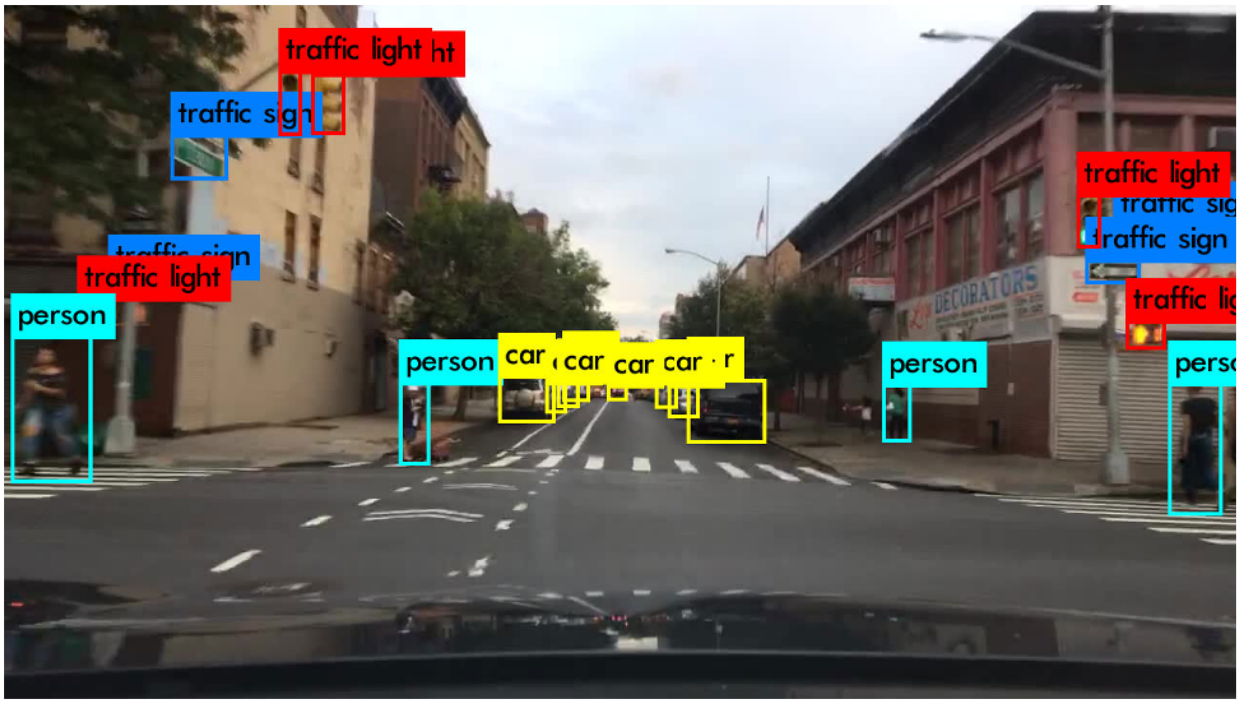


Figure 17: A bit of blurring.

Traffic Lights Detection

Since there are 265906 instances of annotated traffic lights in the dataset, we see that in every figure so far, every traffic light were detected, independent of the angle. However any spot of green, yellow, or red can be misinterpreted as a traffic light. This happens especially at night, when any reflective sign in the distance can be seen as a traffic light. Take Figure 16 for example, clearly a green reflective traffic sign got predicted as a traffic light (confidence score 51%). This is probably the main reason for the low AP of 6.06 for traffic lights detection. In Figure 19, we see exactly that on the top left corner.

Figure 18: Raindrops on the windshield.

Figure 19: Effective detection of distant traffic lights.

Detections in the rain

Since cars drive even when it pours, images captured by the camera can be blocked by rain drops. And sometimes, rain drops on the windshield can make the camera focus on the

windshield rather than in the distance, as seen in Figure 18 and 19. Our model still seems to work.

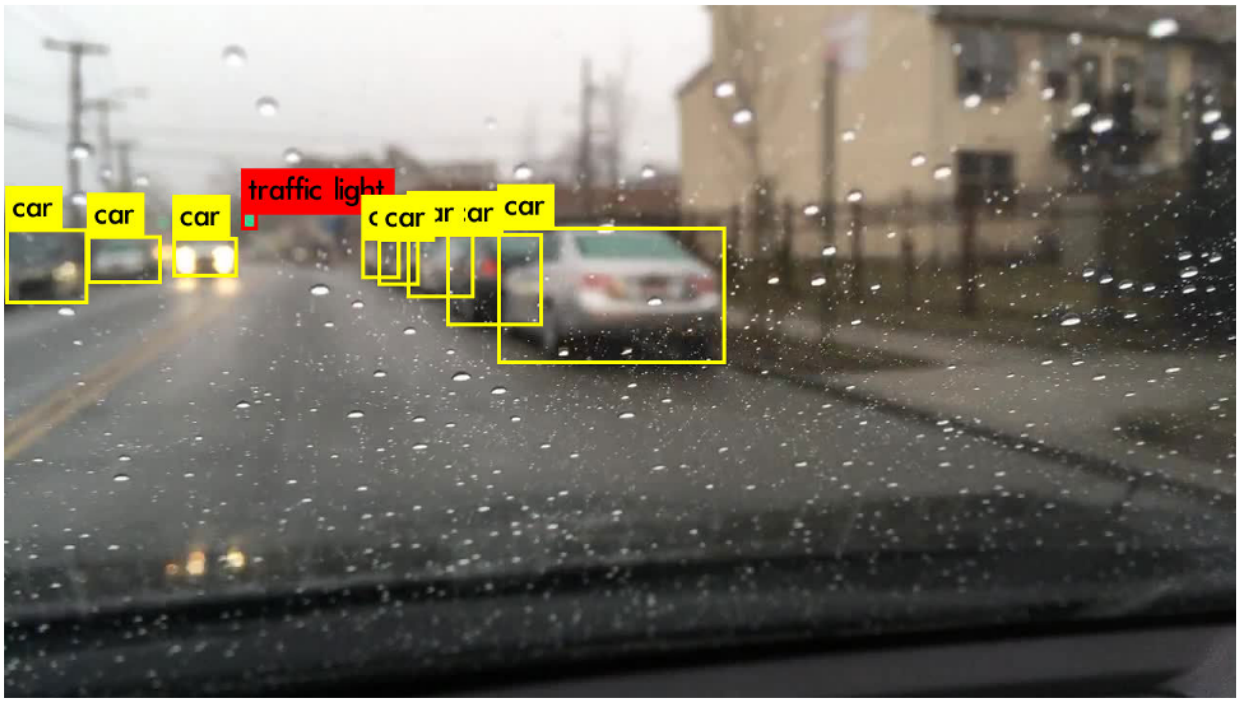


Figure 19: Out of focus image

Bike, Rider, and Motorcycle Detection

Bikes, riders, and motorcycles have low AP. From Figure 7, we see that each of these classes have under 10000 instances (versus the 1000000+ instances for cars). The class imbalance is very strong and even when we feed it an image of a motorcycle in the center, it does not recognize it.

Figure 20: Motorcycle not detected.

It is a little better for bikes, but not really.

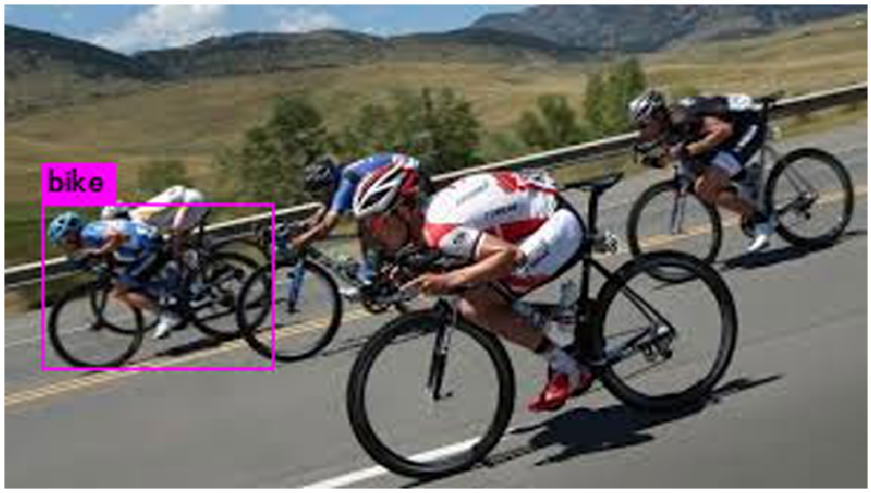


Figure 21: Bike detection.

Car Detection

Since cars are the most prominent objects in the dataset (as illustrated in Figure 7), we would expect cars to have a higher AP than 32.96. The truth is, cars are usually far from one another, and the many cars that we see are too small for the detector. Or sometimes, because of their headlights, it is difficult to make out the shape of cars around it. Take Figure 22 for example, there is clearly a car in the center of the photo that has not been detected. The reason for that is because its tail lights are dimmer than the head lights of the incoming cars.

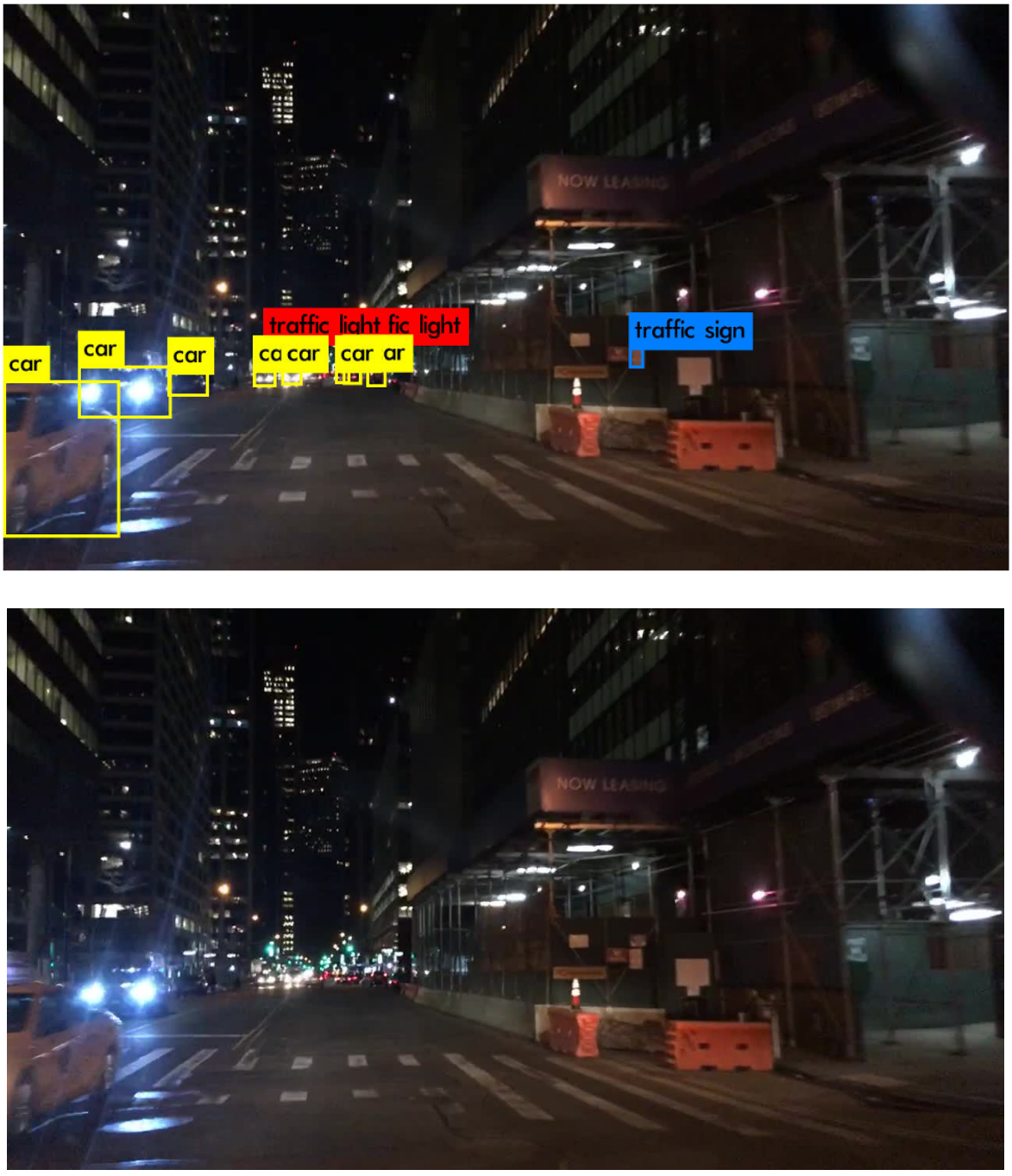


Figure 22: Blinded by lights

Data Augmentation

In general, it seems like 100:1 class imbalances create a lot of problems, which accounts for the overall low mAP. To combat class imbalance, I am thinking of different ways of data augmentation, such as adding more images of rarer classes to the training set, and cropping, and

rotating existing images of them. I could also play with the saturation and exposure levels of those images. Since we are training with 70000 images, I can take out images consisting mainly of cars. The fact that I stopped the training after 35000 iterations due to the limit on time also means that I was not able to reach optimal performance. It is still very interesting to see that cars, trucks, and buses performed the best, as they are probably the most distinct objects on the road (they are generally well separated from one another, unlike pedestrians).

Conclusion

In this experimental study, I have closely examined YOLOv3 for the purpose of real-time object detection in the setting of autonomous driving. To get to this point, I studied the most popular real-time object detection algorithms such as SSD and the different versions of YOLO and I even wrote a literature review to compare their architectural differences give them advantages in respect to each other. The review concluded that YOLOv3 was by far the faster and more accurate model to work with.

In fact, we have seen how YOLOv3 achieves low AP for pedestrians who appear small or in dense areas. The main problem with YOLOv2 was its accuracy at detecting small objects and YOLOv3 was supposed to have solved this. If we ran YOLOv2 on the same dataset, the AP would be even lower as most objects are small in those images. In their published papers, SSD had about the same accuracy as YOLOv2, but ran 3 times slower than YOLOv3, which means that we would expect an FPS of about 11, rendering it unfit for real-time detection, even on a $5000 GPU. As for YOLOv1, it lacked in accuracy and localization error. It is not for nothing that the algorithm evolved to YOLOv3.

A high accuracy and a real-time inference speed are extremely important for the safety of autonomous vehicles. On the BDD100k dataset, despite huge class imbalances, this YOLOv3 model was able to reach a mAP of 15.44, which I believe to be not very representative. For the car class, it reaches an AP of 32.96, because many cars were far away and their head lights tend to diminish the observability of cars around it. As for objects in rarer classes, class imbalance causes our model to not detect them. For pedestrians, we have seen how they only appear when they are a street intersection away from the camera, and the distance is further, the detector would not detect them.

To tackle these problems, I suggested different data augmentation techniques, but there are certainly many other courses of action to take. One of them would be to start with a more balanced dataset, like the KITTI dataset. Although I did not have the time to train SSD, YOLOv1, and YOLOv2 on the BDD100k dataset to compare with YOLOv3, this project was still very exciting. For future works, I want to add real-time object detection on low cost embedded systems. Specifically, I want to train a Tiny-YOLOv3 model. As stated previously, Tiny-YOLOv3 is 10 times lighter than YOLOv3, which means that in general, we expect it to run around 10 times faster. That would be a huge advantage for embedded systems. As estimated in a section above, the inference speed of this YOLOv3 would probably only reach 5.2 FPS on an NVIDIA Jetson Nano (costs $99), but a good guess is that Tiny-YOLOv3 will be able to run 10 times faster, which would put the FPS in the order of 50. It is in fact reported on the NVIDIA developer website that it can run inference up to 25 FPS, hence, making it eligible for real-time applications.

Annexes

YOLO Model

```
batch = 4, time_steps = 1, train = 1
    layer   filters  size/strd(dil)      input                output
    0 conv     32       3 x 3/ 1    416 x 416 x   3 ->  416 x 416 x  32 0.299 BF
    1 conv     64       3 x 3/ 2    416 x 416 x  32 ->  208 x 208 x  64 1.595 BF
    2 conv     32       1 x 1/ 1    208 x 208 x  64 ->  208 x 208 x  32 0.177 BF
    3 conv     64       3 x 3/ 1    208 x 208 x  32 ->  208 x 208 x  64 1.595 BF
    4 Shortcut Layer: 1
    5 conv    128       3 x 3/ 2    208 x 208 x  64 ->  104 x 104 x 128 1.595 BF
    6 conv     64       1 x 1/ 1    104 x 104 x 128 ->  104 x 104 x  64 0.177 BF
    7 conv    128       3 x 3/ 1    104 x 104 x  64 ->  104 x 104 x 128 1.595 BF
    8 Shortcut Layer: 5
    9 conv     64       1 x 1/ 1    104 x 104 x 128 ->  104 x 104 x  64 0.177 BF
   10 conv    128       3 x 3/ 1    104 x 104 x  64 ->  104 x 104 x 128 1.595 BF
   11 Shortcut Layer: 8
   12 conv    256       3 x 3/ 2    104 x 104 x 128 ->   52 x  52 x 256 1.595 BF
   13 conv    128       1 x 1/ 1     52 x  52 x 256 ->   52 x  52 x 128 0.177 BF
   14 conv    256       3 x 3/ 1     52 x  52 x 128 ->   52 x  52 x 256 1.595 BF
   15 Shortcut Layer: 12
   16 conv    128       1 x 1/ 1     52 x  52 x 256 ->   52 x  52 x 128 0.177 BF
   17 conv    256       3 x 3/ 1     52 x  52 x 128 ->   52 x  52 x 256 1.595 BF
   18 Shortcut Layer: 15
   19 conv    128       1 x 1/ 1     52 x  52 x 256 ->   52 x  52 x 128 0.177 BF
   20 conv    256       3 x 3/ 1     52 x  52 x 128 ->   52 x  52 x 256 1.595 BF
   21 Shortcut Layer: 18
   22 conv    128       1 x 1/ 1     52 x  52 x 256 ->   52 x  52 x 128 0.177 BF
   23 conv    256       3 x 3/ 1     52 x  52 x 128 ->   52 x  52 x 256 1.595 BF
   24 Shortcut Layer: 21
   25 conv    128       1 x 1/ 1     52 x  52 x 256 ->   52 x  52 x 128 0.177 BF
   26 conv    256       3 x 3/ 1     52 x  52 x 128 ->   52 x  52 x 256 1.595 BF
   27 Shortcut Layer: 24
   28 conv    128       1 x 1/ 1     52 x  52 x 256 ->   52 x  52 x 128 0.177 BF
   29 conv    256       3 x 3/ 1     52 x  52 x 128 ->   52 x  52 x 256 1.595 BF
   30 Shortcut Layer: 27
   31 conv    128       1 x 1/ 1     52 x  52 x 256 ->   52 x  52 x 128 0.177 BF
   32 conv    256       3 x 3/ 1     52 x  52 x 128 ->   52 x  52 x 256 1.595 BF
   33 Shortcut Layer: 30
   34 conv    128       1 x 1/ 1     52 x  52 x 256 ->   52 x  52 x 128 0.177 BF
   35 conv    256       3 x 3/ 1     52 x  52 x 128 ->   52 x  52 x 256 1.595 BF
   36 Shortcut Layer: 33
   37 conv    512       3 x 3/ 2     52 x  52 x 256 ->   26 x  26 x 512 1.595 BF
   38 conv    256       1 x 1/ 1     26 x  26 x 512 ->   26 x  26 x 256 0.177 BF
   39 conv    512       3 x 3/ 1     26 x  26 x 256 ->   26 x  26 x 512 1.595 BF
   40 Shortcut Layer: 37
   41 conv    256       1 x 1/ 1     26 x  26 x 512 ->   26 x  26 x 256 0.177 BF
   42 conv    512       3 x 3/ 1     26 x  26 x 256 ->   26 x  26 x 512 1.595 BF
   43 Shortcut Layer: 40
   44 conv    256       1 x 1/ 1     26 x  26 x 512 ->   26 x  26 x 256 0.177 BF
   45 conv    512       3 x 3/ 1     26 x  26 x 256 ->   26 x  26 x 512 1.595 BF
   46 Shortcut Layer: 43
   47 conv    256       1 x 1/ 1     26 x  26 x 512 ->   26 x  26 x 256 0.177 BF
   48 conv    512       3 x 3/ 1     26 x  26 x 256 ->   26 x  26 x 512 1.595 BF
   49 Shortcut Layer: 46
   50 conv    256       1 x 1/ 1     26 x  26 x 512 ->   26 x  26 x 256 0.177 BF
```

```
  51 conv    512       3 x 3/ 1     26 x  26 x 256 ->   26 x  26 x 512 1.595 BF
  52 Shortcut Layer: 49
  53 conv    256       1 x 1/ 1     26 x  26 x 512 ->   26 x  26 x 256 0.177 BF
  54 conv    512       3 x 3/ 1     26 x  26 x 256 ->   26 x  26 x 512 1.595 BF
  55 Shortcut Layer: 52
  56 conv    256       1 x 1/ 1     26 x  26 x 512 ->   26 x  26 x 256 0.177 BF
  57 conv    512       3 x 3/ 1     26 x  26 x 256 ->   26 x  26 x 512 1.595 BF
  58 Shortcut Layer: 55
  59 conv    256       1 x 1/ 1     26 x  26 x 512 ->   26 x  26 x 256 0.177 BF
  60 conv    512       3 x 3/ 1     26 x  26 x 256 ->   26 x  26 x 512 1.595 BF
  61 Shortcut Layer: 58
  62 conv   1024       3 x 3/ 2     26 x  26 x 512 ->   13 x  13 x1024 1.595 BF
  63 conv    512       1 x 1/ 1     13 x  13 x1024 ->   13 x  13 x 512 0.177 BF
  64 conv   1024       3 x 3/ 1     13 x  13 x 512 ->   13 x  13 x1024 1.595 BF
  65 Shortcut Layer: 62
  66 conv    512       1 x 1/ 1     13 x  13 x1024 ->   13 x  13 x 512 0.177 BF
  67 conv   1024       3 x 3/ 1     13 x  13 x 512 ->   13 x  13 x1024 1.595 BF
  68 Shortcut Layer: 65
  69 conv    512       1 x 1/ 1     13 x  13 x1024 ->   13 x  13 x 512 0.177 BF
  70 conv   1024       3 x 3/ 1     13 x  13 x 512 ->   13 x  13 x1024 1.595 BF
  71 Shortcut Layer: 68
  72 conv    512       1 x 1/ 1     13 x  13 x1024 ->   13 x  13 x 512 0.177 BF
  73 conv   1024       3 x 3/ 1     13 x  13 x 512 ->   13 x  13 x1024 1.595 BF
  74 Shortcut Layer: 71
  75 conv    512       1 x 1/ 1     13 x  13 x1024 ->   13 x  13 x 512 0.177 BF
  76 conv   1024       3 x 3/ 1     13 x  13 x 512 ->   13 x  13 x1024 1.595 BF
  77 conv    512       1 x 1/ 1     13 x  13 x1024 ->   13 x  13 x 512 0.177 BF
  78 conv   1024       3 x 3/ 1     13 x  13 x 512 ->   13 x  13 x1024 1.595 BF
  79 conv    512       1 x 1/ 1     13 x  13 x1024 ->   13 x  13 x 512 0.177 BF
  80 conv   1024       3 x 3/ 1     13 x  13 x 512 ->   13 x  13 x1024 1.595 BF
  81 conv     45       1 x 1/ 1     13 x  13 x1024 ->   13 x  13 x  45 0.016 BF
  82 yolo
[yolo] params: iou loss: mse (2), iou_norm: 0.75, cls_norm: 1.00, scale_x_y: 1.00
  83 route  79                                   ->   13 x  13 x 512
  84 conv    256       1 x 1/ 1     13 x  13 x 512 ->   13 x  13 x 256 0.044 BF
  85 upsample                 2x    13 x  13 x 256 ->   26 x  26 x 256
  86 route  85 61                                ->   26 x  26 x 768
  87 conv    256       1 x 1/ 1     26 x  26 x 768 ->   26 x  26 x 256 0.266 BF
  88 conv    512       3 x 3/ 1     26 x  26 x 256 ->   26 x  26 x 512 1.595 BF
  89 conv    256       1 x 1/ 1     26 x  26 x 512 ->   26 x  26 x 256 0.177 BF
  90 conv    512       3 x 3/ 1     26 x  26 x 256 ->   26 x  26 x 512 1.595 BF
  91 conv    256       1 x 1/ 1     26 x  26 x 512 ->   26 x  26 x 256 0.177 BF
  92 conv    512       3 x 3/ 1     26 x  26 x 256 ->   26 x  26 x 512 1.595 BF
  93 conv     45       1 x 1/ 1     26 x  26 x 512 ->   26 x  26 x  45 0.031 BF
  94 yolo
[yolo] params: iou loss: mse (2), iou_norm: 0.75, cls_norm: 1.00, scale_x_y: 1.00
         95 route  91                                   ->   26 x  26 x 256
         96 conv    128       1 x 1/ 1     26 x  26 x 256 ->   26 x  26 x 128 0.044 BF
         97 upsample                 2x    26 x  26 x 128 ->   52 x  52 x 128
         98 route  97 36                                ->   52 x  52 x 384
         99 conv    128       1 x 1/ 1     52 x  52 x 384 ->   52 x  52 x 128 0.266 BF
        100 conv    256       3 x 3/ 1     52 x  52 x 128 ->   52 x  52 x 256 1.595 BF
        101 conv    128       1 x 1/ 1     52 x  52 x 256 ->   52 x  52 x 128 0.177 BF
        102 conv    256       3 x 3/ 1     52 x  52 x 128 ->   52 x  52 x 256 1.595 BF
        103 conv    128       1 x 1/ 1     52 x  52 x 256 ->   52 x  52 x 128 0.177 BF
```

```
109.         104 conv    256       3 x 3/ 1     52 x  52 x 128 ->   52 x  52 x 256 1.595 BF
110.         105 conv     45       1 x 1/ 1     52 x  52 x 256 ->   52 x  52 x  45 0.062 BF
111.         106 yolo
112.        [yolo] params: iou loss: mse (2), iou_norm: 0.75, cls_norm: 1.00, scale_x_y: 1.0
    0
113.        Total BFLOPS 65.355
```

Extracting Labels

This script formats labels from the Berkeley DeepDrive dataset to standard YOLO labels.

```
#!/usr/bin/python
f = open(image_label_txt, "w+")
import json
import csv


json_data = open('/home/user/Desktop/bdd100k_data/bdd100k/labels/ -- ')
print("----- LOADING JSON -----")
data = json.load(json_data)

# Construct a dictionary
class_dict = {}

num_of_images = len(data)
# initialze current_index
latest_class_label = 0 #Only add one when it is a new class

# List of category that need to be ignorded
list_of_category_to_ignore = ["drivable area", "lane"]


# Fill out the dictionary
for i in range(num_of_images):
    print("[STEP 1] ==============  image number i: ", i)
    print("latest_class_label: " , latest_class_label)
    current_image = data[i]
    image_name = current_image['name']
    print("image_name: ", image_name)
    # Create a Text file for each image
    image_label_txt = str(image_name[:-4]) + ".txt"
    print("image_label_txt: ", image_label_txt)



    num_of_object_in_image = len(current_image['labels']) # In the type of dict

    print('num_of_object_in_image: ' , num_of_object_in_image)
    # For each object in the current image
    for j in range(num_of_object_in_image):
        print("[STEP 2] ============== object number: ", j)
        current_object = current_image['labels'][j]
        current_object_label = current_object['category']
        print("current_object_label:   ", current_object_label)
```

```
        if current_object_label in list_of_category_to_ignore:
            print("The current object label is in the ignore list")
        else:
            # Get the X-Y info
            y1 = current_object['box2d']['y1']
            x2 = current_object['box2d']['x2']
            x1 = current_object['box2d']['x1']
            y2 = current_object['box2d']['y2']
            image_width = 1280
            image_height = 720

            # x-y => Center of the box2d
            bbox_x = (x1 + x2)/2
            bbox_y = (y1 + y2)/2

            bbox_x_normalized = bbox_x / image_width
            bbox_y_normalized = bbox_y / image_height

            bbox_width = abs(x1-x2)
            bbox_height = abs(y1-y2)

            bbox_width_normalized = bbox_width / image_width
            bbox_height_normalized = bbox_height / image_height

            print("Center X-Y & Width-Height")
            print(bbox_x, bbox_y, bbox_width, bbox_height)
            print("NORMALIZED Center X-Y & Width-Height")
            print(bbox_x_normalized, bbox_y_normalized, bbox_width_normalized,
bbox_height_normalized)


            # Check if current_object_label is already in the dictionary (Add )
            if class_dict.has_key(current_object_label):
                print("current_object_label is already in the class_dict")
                # show current size of class_dict
                print("len(class_dict):  ", len(class_dict))
            else:
                print("New Class: add to class_dict")
                print("label_to_be_added: ", current_object_label)
                class_dict[current_object_label] = latest_class_label
                # Update label count
                latest_class_label = latest_class_label + 1
                # show current size of class_dict
                print("len(class_dict):  ", len(class_dict))

            # Get the corresponding label from the class dict
            class_label_code = class_dict[current_object_label]
            print("class_label_code: ", class_label_code)
```

```
            # Prepare the line to be written down to txt file
            line_to_be_written = str(class_label_code) + " " + str(bbox_x_normalized)
+ " " + str(bbox_y_normalized) + " " + str(bbox_height_normalized) + " " +
str(bbox_width_normalized)
            print("line_to_be_written: ", line_to_be_written)
            f.write(line_to_be_written + "\n")


    f.close()
    print("=== Show current Class Dict Collection ====")
    print(class_dict.items())
    #input("Press Enter to continue...")



print(" --- END OF CLASS DICT CONSTRUCT ---")
print(class_dict.items())

# Save class dictionary to text file
f = open("class_dict.txt","w")
f.write( str(class_dict) )
f.close()

# Save class dictionary to csv file
w = csv.writer(open("class_dict.csv", "w"))
for key, val in class_dict.items():
    w.writerow([key, val])
```

Train/Val Dataset

This script creates the training and validation sets.

```
import pickle
import os
from os import listdir, getcwd
from os.path import join

# Here we need the directory of the training images
train_images_dir = '/home/user/Desktop/bdd100k_data/bdd100k/images/100k/train'
# Here we need the directory of the validation images
val_images_dir = '/home/user/Desktop/bdd100k_data/bdd100k/images/100k/val'

f = open("train.txt", "w+")

for subdirs, dirs, files in os.walk(train_images_dir):
    for filename in files:
        if filename.endswith(".jpg"):
            print("Yes")
            train_image_path = os.path.join(train_images_dir, filename)
            print(train_image_path)
            f.write(train_image_path + "\n")
f.close()


f = open("val.txt", "w+")

for subdirs, dirs, files in os.walk(val_images_dir):
    for filename in files:
        if filename.endswith(".jpg"):
            print("Yes")
            val_image_path = os.path.join(val_images_dir, filename)
            print(val_image_path)
            f.write(val_image_path + "\n")
f.close()
```

Training Script

Excerpt from the configuration file (.cfg)

```
# Training
batch=64
subdivisions=16
width=512
height=512
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1

learning_rate=0.0001
burn_in=1000
max_batches = 500200
policy=steps
steps=400000,450000
scales=.1,.1
```

BDD.data

```
1. classes= 10
2. train  = "/content/gdrive/My Drive/yolov3/cfg/train.txt"
3. #valid  = "/content/gdrive/My Drive/yolov3/cfg/test.txt"
4. valid = "/content/gdrive/My Drive/yolov3/cfg/val.txt"
5. names = "/content/gdrive/My Drive/yolov3/cfg/BDD.names"
6. backup = "/content/gdrive/My Drive/yolov3/backup/bdd100k"
```

where BDD.names contains the 10 classes.

train.txt, test.txt and val.txt contains the path to each image.

Validation Script.

Taken from https://github.com/ucbdrive/bdd-data

```
import argparse
import copy
import json
import os
from collections import defaultdict

import os.path as osp

import numpy as np
from PIL import Image


def parse_args():
    """Use argparse to get command line arguments."""
    parser = argparse.ArgumentParser()
    parser.add_argument('task', choices=['seg', 'det', 'drivable'])
    parser.add_argument('gt', help='path to ground truth')
    parser.add_argument('result', help='path to results to be evaluated')
    args = parser.parse_args()

    return args


def fast_hist(gt, prediction, n):
    k = (gt >= 0) & (gt < n)
    return np.bincount(
        n * gt[k].astype(int) + prediction[k], minlength=n ** 2).reshape(n, n)


def per_class_iu(hist):
    ious = np.diag(hist) / (hist.sum(1) + hist.sum(0) - np.diag(hist))
    ious[np.isnan(ious)] = 0
    return ious


def find_all_png(folder):
    paths = []
    for root, dirs, files in os.walk(folder, topdown=True):
        paths.extend([osp.join(root, f)
                      for f in files if osp.splitext(f)[1] == '.png'])
    return paths


def evaluate_segmentation(gt_dir, result_dir, num_classes, key_length):
```

```
    gt_dict = dict([(osp.split(p)[1][:key_length], p)
                    for p in find_all_png(gt_dir)])
    result_dict = dict([(osp.split(p)[1][:key_length], p)
                        for p in find_all_png(result_dir)])
    result_gt_keys = set(gt_dict.keys()) & set(result_dict.keys())
    if len(result_gt_keys) != len(gt_dict):
        raise ValueError('Result folder only has {} of {} ground truth files.'
                         .format(len(result_gt_keys), len(gt_dict)))
    print('Found', len(result_dict), 'results')
    print('Evaluating', len(gt_dict), 'results')
    hist = np.zeros((num_classes, num_classes))
    i = 0
    gt_id_set = set()
    for key in sorted(gt_dict.keys()):
        gt_path = gt_dict[key]
        result_path = result_dict[key]
        gt = np.asarray(Image.open(gt_path, 'r'))
        gt_id_set.update(np.unique(gt).tolist())
        prediction = np.asanyarray(Image.open(result_path, 'r'))
        hist += fast_hist(gt.flatten(), prediction.flatten(), num_classes)
        i += 1
        if i % 100 == 0:
            print('Finished', i, per_class_iu(hist) * 100)
    gt_id_set.remove(255)
    print('GT id set', gt_id_set)
    ious = per_class_iu(hist) * 100
    miou = np.mean(ious[list(gt_id_set)])
    return miou, list(ious)


def evaluate_drivable(gt_dir, result_dir):
    return evaluate_segmentation(gt_dir, result_dir, 3, 17)


def get_ap(recalls, precisions):
    # correct AP calculation
    # first append sentinel values at the end
    recalls = np.concatenate(([0.], recalls, [1.]))
    precisions = np.concatenate(([0.], precisions, [0.]))

    # compute the precision envelope
    for i in range(precisions.size - 1, 0, -1):
        precisions[i - 1] = np.maximum(precisions[i - 1], precisions[i])

    # to calculate area under PR curve, look for points
    # where X axis (recall) changes value
    i = np.where(recalls[1:] != recalls[:-1])[0]
```

```
    # and sum (\Delta recall) * prec
    ap = np.sum((recalls[i + 1] - recalls[i]) * precisions[i + 1])
    return ap


def group_by_key(detections, key):
    groups = defaultdict(list)
    for d in detections:
        groups[d[key]].append(d)
    return groups


def cat_pc(gt, predictions, thresholds):
    """
    Implementation refers to https://github.com/rbgirshick/py-faster-rcnn
    """
    num_gts = len(gt)
    image_gts = group_by_key(gt, 'name')
    image_gt_boxes = {k: np.array([[float(z) for z in b['bbox']]
                                   for b in boxes])
                      for k, boxes in image_gts.items()}
    image_gt_checked = {k: np.zeros((len(boxes), len(thresholds)))
                        for k, boxes in image_gts.items()}
    predictions = sorted(predictions, key=lambda x: x['score'], reverse=True)


    # go down dets and mark TPs and FPs
    nd = len(predictions)
    tp = np.zeros((nd, len(thresholds)))
    fp = np.zeros((nd, len(thresholds)))
    count = 0
    count_bg = 0
    for i, p in enumerate(predictions):
        box = p['bbox']
        ovmax = -np.inf
        jmax = -1
        try:
            gt_boxes = image_gt_boxes[p['name']]
            gt_checked = image_gt_checked[p['name']]
        except KeyError:
            gt_boxes = []
            gt_checked = None


        if len(gt_boxes) > 0:
            # compute overlaps
            # intersection
            ixmin = np.maximum(gt_boxes[:, 0], box[0])
```

```
            iymin = np.maximum(gt_boxes[:, 1], box[1])
            ixmax = np.minimum(gt_boxes[:, 2], box[2])
            iymax = np.minimum(gt_boxes[:, 3], box[3])
            iw = np.maximum(ixmax - ixmin + 1., 0.)
            ih = np.maximum(iymax - iymin + 1., 0.)
            inters = iw * ih

            # union
            uni = ((box[2] - box[0] + 1.) * (box[3] - box[1] + 1.) +
                   (gt_boxes[:, 2] - gt_boxes[:, 0] + 1.) *
                   (gt_boxes[:, 3] - gt_boxes[:, 1] + 1.) - inters)

            overlaps = inters / uni
            ovmax = np.max(overlaps)
            jmax = np.argmax(overlaps)

        for t, threshold in enumerate(thresholds):
            if ovmax > threshold:
                if gt_checked[jmax, t] == 0:
                    tp[i, t] = 1.
                    gt_checked[jmax, t] = 1
                else:
                    fp[i, t] = 1.
            else:
                fp[i, t] = 1.


    # compute precision recall
    fp = np.cumsum(fp, axis=0)
    tp = np.cumsum(tp, axis=0)
    recalls = tp / float(num_gts)
    # avoid divide by zero in case the first detection matches a difficult
    # ground truth
    precisions = tp / np.maximum(tp + fp, np.finfo(np.float64).eps)
    ap = np.zeros(len(thresholds))
    for t in range(len(thresholds)):
        ap[t] = get_ap(recalls[:, t], precisions[:, t])

    return recalls, precisions, ap


def evaluate_detection(gt_path, result_path):
    print("here",gt_path,result_path)
    gt = json.load(open(gt_path, 'r'))
    pred = json.load(open(result_path, 'r'))
    cat_gt = group_by_key(gt, 'category')
    cat_pred = group_by_key(pred, 'category')
    cat_list = sorted(cat_gt.keys())
```

```
    thresholds = [0.75]
    aps = np.zeros((len(thresholds), len(cat_list)))
    c = []

    for i, cat in enumerate(cat_list):
        if cat in cat_pred:
            r, p, ap = cat_pc(cat_gt[cat], cat_pred[cat], thresholds)
            aps[:, i] = ap
            print("AP : %.2f (%s)" %(ap*100, cat))
    aps *= 100
    mAP = np.mean(aps)
    return mAP, aps.flatten().tolist()


def main():
    args = parse_args()

    if args.task == 'drivable':
        mean, breakdown = evaluate_drivable(args.gt, args.result)
    elif args.task == 'seg':
        mean, breakdown = evaluate_segmentation(args.gt, args.result, 19, 17)
    elif args.task == 'det':
        mean, breakdown = evaluate_detection(args.gt, args.result)

    print(breakdown,"\n")
    print(' '.join(['{:.2f}'.format(n) for n in breakdown]))
    print('\nmAP {:.2f}'.format(mean))


if __name__ == '__main__':
    main()
```

Helper Functions

Scripts that were used to be more efficient on Google Colab. Thanks to Ivan Goncharov.

```
#download files
def imShow(path):
  import cv2
  import matplotlib.pyplot as plt
  %matplotlib inline

  image = cv2.imread(path)
  height, width = image.shape[:2]
  resized_image = cv2.resize(image,(3*width, 3*height), interpolation = cv2.INTER_CUBIC
)

  fig = plt.gcf()
  fig.set_size_inches(18, 10)
  plt.axis("off")
  #plt.rcParams['figure.figsize'] = [10, 5]
  plt.imshow(cv2.cvtColor(resized_image, cv2.COLOR_BGR2RGB))
  plt.show()


def upload():
  from google.colab import files
  uploaded = files.upload()
  for name, data in uploaded.items():
    with open(name, 'wb') as f:
      f.write(data)
      print ('saved file', name)
def download(path):
  from google.colab import files
  files.download(path)
```

YOLO Paper

YOLOv3: An Incremental Improvement

Joseph Redmon, Ali Farhadi
University of Washington

Abstract

We present some updates to YOLO! We made a bunch of little design changes to make it better. We also trained this new network that's pretty swell. It's a little bigger than last time but more accurate. It's still fast though, don't worry. At 320×320 *YOLOv3 runs in 22 ms at 28.2 mAP, as accurate as SSD but three times faster. When we look at the old .5 IOU mAP detection metric YOLOv3 is quite good. It achieves* 57.9 AP_{50} *in 51 ms on a Titan X, compared to* 57.5 AP_{50} *in 198 ms by RetinaNet, similar performance but 3.8*$\times$ *faster. As always, all the code is online at* https://pjreddie.com/yolo/.

1. Introduction

Sometimes you just kinda phone it in for a year, you know? I didn't do a whole lot of research this year. Spent a lot of time on Twitter. Played around with GANs a little. I had a little momentum left over from last year [12] [1]; I managed to make some improvements to YOLO. But, honestly, nothing like super interesting, just a bunch of small changes that make it better. I also helped out with other people's research a little.

Actually, that's what brings us here today. We have a camera-ready deadline [4] and we need to cite some of the random updates I made to YOLO but we don't have a source. So get ready for a TECH REPORT!

The great thing about tech reports is that they don't need intros, y'all know why we're here. So the end of this introduction will signpost for the rest of the paper. First we'll tell you what the deal is with YOLOv3. Then we'll tell you how we do. We'll also tell you about some things we tried that didn't work. Finally we'll contemplate what this all means.

2. The Deal

So here's the deal with YOLOv3: We mostly took good ideas from other people. We also trained a new classifier network that's better than the other ones. We'll just take you through the whole system from scratch so you can understand it all.

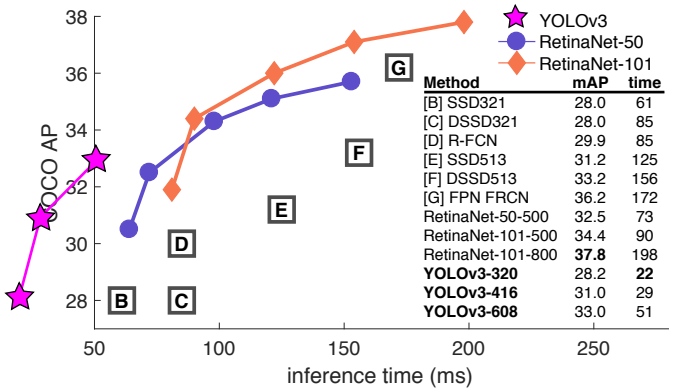


Figure 1. We adapt this figure from the Focal Loss paper [9]. YOLOv3 runs significantly faster than other detection methods with comparable performance. Times from either an M40 or Titan X, they are basically the same GPU.

2.1. Bounding Box Prediction

Following YOLO9000 our system predicts bounding boxes using dimension clusters as anchor boxes [15]. The network predicts 4 coordinates for each bounding box, t_x, t_y, t_w, t_h. If the cell is offset from the top left corner of the image by (c_x, c_y) and the bounding box prior has width and height p_w, p_h, then the predictions correspond to:

$$
\begin{aligned}
b_x &= \sigma(t_x) + c_x \\
b_y &= \sigma(t_y) + c_y \\
b_w &= p_w e^{t_w} \\
b_h &= p_h e^{t_h}
\end{aligned}
$$

During training we use sum of squared error loss. If the ground truth for some coordinate prediction is $\hat{t}_*$ our gradient is the ground truth value (computed from the ground truth box) minus our prediction: $\hat{t}_* - t_*$. This ground truth value can be easily computed by inverting the equations above.

YOLOv3 predicts an objectness score for each bounding box using logistic regression. This should be 1 if the bounding box prior overlaps a ground truth object by more than any other bounding box prior. If the bounding box prior

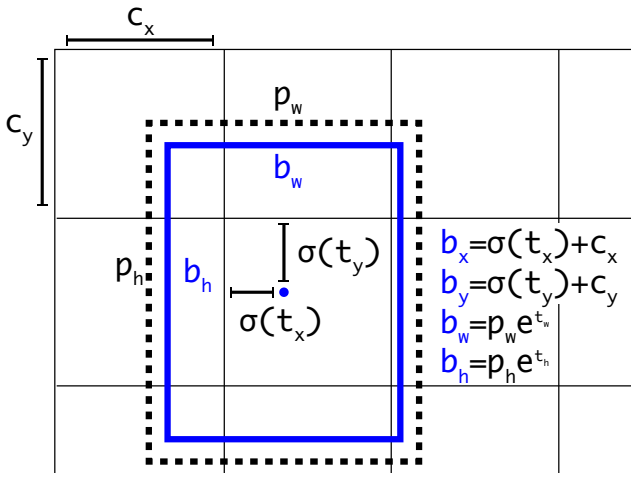


Figure 2. **Bounding boxes with dimension priors and location prediction.** We predict the width and height of the box as offsets from cluster centroids. We predict the center coordinates of the box relative to the location of filter application using a sigmoid function. This figure blatantly self-plagiarized from [15].

is not the best but does overlap a ground truth object by more than some threshold we ignore the prediction, following [17]. We use the threshold of .5. Unlike [17] our system only assigns one bounding box prior for each ground truth object. If a bounding box prior is not assigned to a ground truth object it incurs no loss for coordinate or class predictions, only objectness.

2.2. Class Prediction

Each box predicts the classes the bounding box may contain using multilabel classification. We do not use a softmax as we have found it is unnecessary for good performance, instead we simply use independent logistic classifiers. During training we use binary cross-entropy loss for the class predictions.

This formulation helps when we move to more complex domains like the Open Images Dataset [7]. In this dataset there are many overlapping labels (i.e. Woman and Person). Using a softmax imposes the assumption that each box has exactly one class which is often not the case. A multilabel approach better models the data.

2.3. Predictions Across Scales

YOLOv3 predicts boxes at 3 different scales. Our system extracts features from those scales using a similar concept to feature pyramid networks [8]. From our base feature extractor we add several convolutional layers. The last of these predicts a 3-d tensor encoding bounding box, objectness, and class predictions. In our experiments with COCO [10] we predict 3 boxes at each scale so the tensor is $N \times N \times [3 * (4 + 1 + 80)]$ for the 4 bounding box offsets, 1 objectness prediction, and 80 class predictions.

Next we take the feature map from 2 layers previous and upsample it by $2\times$. We also take a feature map from earlier in the network and merge it with our upsampled features using concatenation. This method allows us to get more meaningful semantic information from the upsampled features and finer-grained information from the earlier feature map. We then add a few more convolutional layers to process this combined feature map, and eventually predict a similar tensor, although now twice the size.

We perform the same design one more time to predict boxes for the final scale. Thus our predictions for the 3rd scale benefit from all the prior computation as well as fine-grained features from early on in the network.

We still use k-means clustering to determine our bounding box priors. We just sort of chose 9 clusters and 3 scales arbitrarily and then divide up the clusters evenly across scales. On the COCO dataset the 9 clusters were: $(10 \times 13), (16 \times 30), (33 \times 23), (30 \times 61), (62 \times 45), (59 \times 119), (116 \times 90), (156 \times 198), (373 \times 326)$.

2.4. Feature Extractor

We use a new network for performing feature extraction. Our new network is a hybrid approach between the network used in YOLOv2, Darknet-19, and that newfangled residual network stuff. Our network uses successive 3×3 and 1×1 convolutional layers but now has some shortcut connections as well and is significantly larger. It has 53 convolutional layers so we call it.... wait for it..... Darknet-53!

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Table 1. **Darknet-53.**

This new network is much more powerful than Darknet-19 but still more efficient than ResNet-101 or ResNet-152. Here are some ImageNet results:

Backbone	Top-1	Top-5	Bn Ops	BFLOP/s	FPS
Darknet-19 [15]	74.1	91.8	7.29	1246	**171**
ResNet-101[5]	77.1	93.7	19.7	1039	53
ResNet-152 [5]	**77.6**	**93.8**	29.4	1090	37
Darknet-53	77.2	**93.8**	18.7	**1457**	78

Table 2. **Comparison of backbones.** Accuracy, billions of operations, billion floating point operations per second, and FPS for various networks.

Each network is trained with identical settings and tested at 256×256, single crop accuracy. Run times are measured on a Titan X at 256×256. Thus Darknet-53 performs on par with state-of-the-art classifiers but with fewer floating point operations and more speed. Darknet-53 is better than ResNet-101 and $1.5\times$ faster. Darknet-53 has similar performance to ResNet-152 and is $2\times$ faster.

Darknet-53 also achieves the highest measured floating point operations per second. This means the network structure better utilizes the GPU, making it more efficient to evaluate and thus faster. That's mostly because ResNets have just way too many layers and aren't very efficient.

2.5. Training

We still train on full images with no hard negative mining or any of that stuff. We use multi-scale training, lots of data augmentation, batch normalization, all the standard stuff. We use the Darknet neural network framework for training and testing [14].

3. How We Do

YOLOv3 is pretty good! See table 3. In terms of COCOs weird average mean AP metric it is on par with the SSD variants but is $3\times$ faster. It is still quite a bit behind other models like RetinaNet in this metric though.

However, when we look at the "old" detection metric of mAP at IOU$=.5$ (or AP_{50} in the chart) YOLOv3 is very strong. It is almost on par with RetinaNet and far above the SSD variants. This indicates that YOLOv3 is a very strong detector that excels at producing decent boxes for objects. However, performance drops significantly as the IOU threshold increases indicating YOLOv3 struggles to get the boxes perfectly aligned with the object.

In the past YOLO struggled with small objects. However, now we see a reversal in that trend. With the new multi-scale predictions we see YOLOv3 has relatively high AP_S performance. However, it has comparatively worse performance on medium and larger size objects. More investigation is needed to get to the bottom of this.

When we plot accuracy vs speed on the AP_{50} metric (see figure 5) we see YOLOv3 has significant benefits over other detection systems. Namely, it's faster and better.

4. Things We Tried That Didn't Work

We tried lots of stuff while we were working on YOLOv3. A lot of it didn't work. Here's the stuff we can remember.

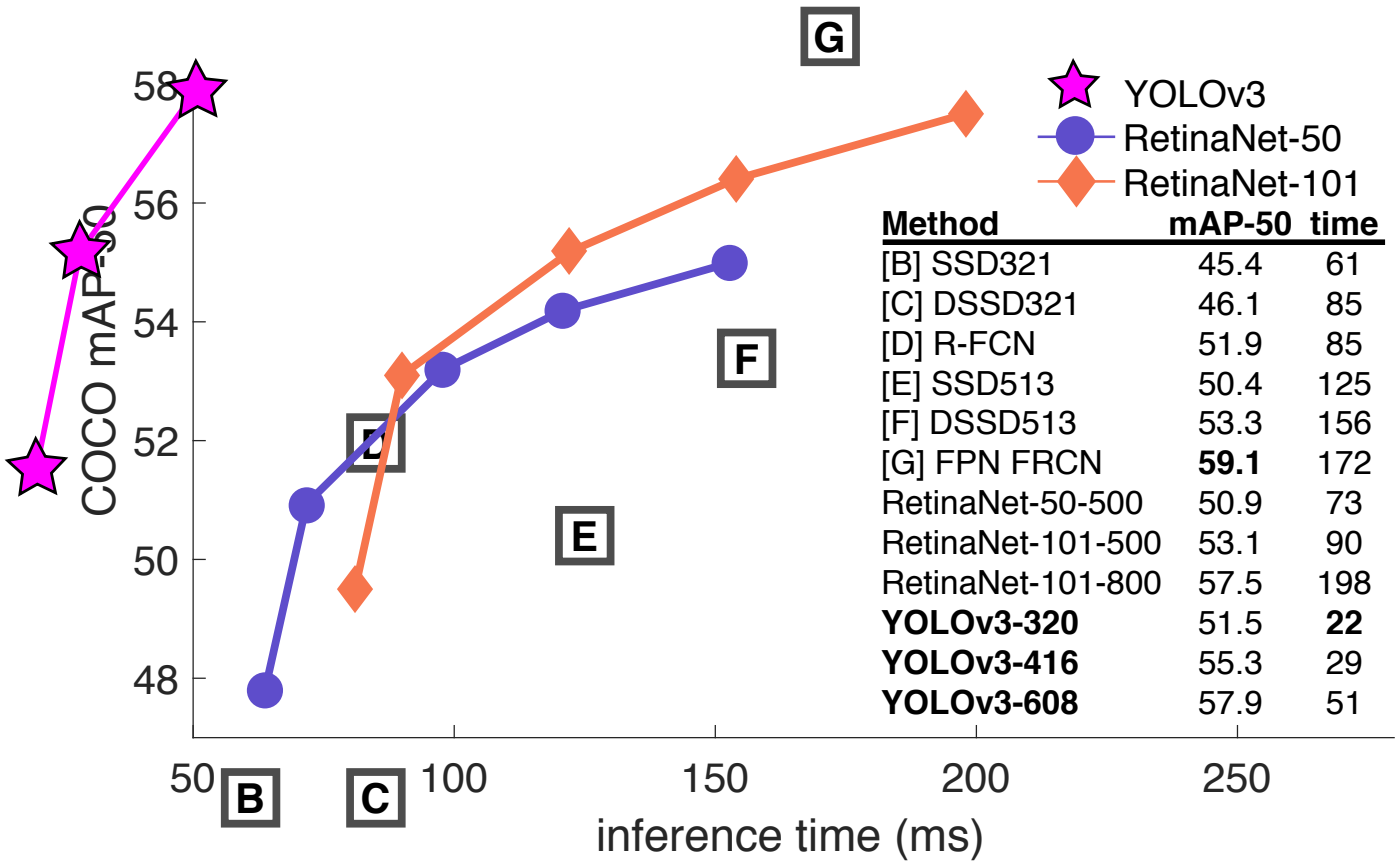
Anchor box x, y offset predictions. We tried using the normal anchor box prediction mechanism where you predict the x, y offset as a multiple of the box width or height using a linear activation. We found this formulation decreased model stability and didn't work very well.

Linear x, y predictions instead of logistic. We tried using a linear activation to directly predict the x, y offset instead of the logistic activation. This led to a couple point drop in mAP.

Focal loss. We tried using focal loss. It dropped our mAP about 2 points. YOLOv3 may already be robust to the problem focal loss is trying to solve because it has separate objectness predictions and conditional class predictions. Thus for most examples there is no loss from the class predictions? Or something? We aren't totally sure.

	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

Table 3. I'm seriously just stealing all these tables from [9] they take soooo long to make from scratch. Ok, YOLOv3 is doing alright. Keep in mind that RetinaNet has like $3.8\times$ longer to process an image. YOLOv3 is much better than SSD variants and comparable to state-of-the-art models on the AP_{50} metric.



Method	mAP-50	time
[B] SSD321	45.4	61
[C] DSSD321	46.1	85
[D] R-FCN	51.9	85
[E] SSD513	50.4	125
[F] DSSD513	53.3	156
[G] FPN FRCN	**59.1**	172
RetinaNet-50-500	50.9	73
RetinaNet-101-500	53.1	90
RetinaNet-101-800	57.5	198
YOLOv3-320	51.5	**22**
YOLOv3-416	55.3	29
YOLOv3-608	57.9	51

Figure 3. Again adapted from the [9], this time displaying speed/accuracy tradeoff on the mAP at .5 IOU metric. You can tell YOLOv3 is good because it's very high and far to the left. Can you cite your own paper? Guess who's going to try, this guy → [16]. Oh, I forgot, we also fix a data loading bug in YOLOv2, that helped by like 2 mAP. Just sneaking this in here to not throw off layout.

Dual IOU thresholds and truth assignment. Faster R-CNN uses two IOU thresholds during training. If a prediction overlaps the ground truth by .7 it is as a positive example, by $[.3 - .7]$ it is ignored, less than .3 for all ground truth objects it is a negative example. We tried a similar strategy but couldn't get good results.

We quite like our current formulation, it seems to be at a local optima at least. It is possible that some of these techniques could eventually produce good results, perhaps they just need some tuning to stabilize the training.

5. What This All Means

YOLOv3 is a good detector. It's fast, it's accurate. It's not as great on the COCO average AP between .5 and .95 IOU metric. But it's very good on the old detection metric of .5 IOU.

Why did we switch metrics anyway? The original COCO paper just has this cryptic sentence: "A full discussion of evaluation metrics will be added once the evaluation server is complete". Russakovsky et al report that that humans have a hard time distinguishing an IOU of .3 from .5! "Training humans to visually inspect a bounding box with IOU of 0.3 and distinguish it from one with IOU 0.5 is surprisingly difficult." [18] If humans have a hard time telling the difference, how much does it matter?

But maybe a better question is: "What are we going to do with these detectors now that we have them?" A lot of the people doing this research are at Google and Facebook. I guess at least we know the technology is in good hands and definitely won't be used to harvest your personal information and sell it to.... wait, you're saying that's exactly what it will be used for?? Oh.

Well the other people heavily funding vision research are the military and they've never done anything horrible like killing lots of people with new technology oh wait.....[1]

I have a lot of hope that most of the people using computer vision are just doing happy, good stuff with it, like counting the number of zebras in a national park [13], or tracking their cat as it wanders around their house [19]. But computer vision is already being put to questionable use and as researchers we have a responsibility to at least consider the harm our work might be doing and think of ways to mitigate it. We owe the world that much.

In closing, do not @ me. (Because I finally quit Twitter).

[1]The author is funded by the Office of Naval Research and Google.

References

[1] Analogy. *Wikipedia*, Mar 2018. 1

[2] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010. 6

[3] C.-Y. Fu, W. Liu, A. Ranga, A. Tyagi, and A. C. Berg. Dssd: Deconvolutional single shot detector. *arXiv preprint arXiv:1701.06659*, 2017. 3

[4] D. Gordon, A. Kembhavi, M. Rastegari, J. Redmon, D. Fox, and A. Farhadi. Iqa: Visual question answering in interactive environments. *arXiv preprint arXiv:1712.03316*, 2017. 1

[5] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 3

[6] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, et al. Speed/accuracy trade-offs for modern convolutional object detectors. 3

[7] I. Krasin, T. Duerig, N. Alldrin, V. Ferrari, S. Abu-El-Haija, A. Kuznetsova, H. Rom, J. Uijlings, S. Popov, A. Veit, S. Belongie, V. Gomes, A. Gupta, C. Sun, G. Chechik, D. Cai, Z. Feng, D. Narayanan, and K. Murphy. Open-images: A public dataset for large-scale multi-label and multi-class image classification. *Dataset available from https://github.com/openimages*, 2017. 2

[8] T.-Y. Lin, P. Dollar, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2117–2125, 2017. 2, 3

[9] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár. Focal loss for dense object detection. *arXiv preprint arXiv:1708.02002*, 2017. 1, 3, 4

[10] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014. 2

[11] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016. 3

[12] I. Newton. *Philosophiae naturalis principia mathematica*. William Dawson & Sons Ltd., London, 1687. 1

[13] J. Parham, J. Crall, C. Stewart, T. Berger-Wolf, and D. Rubenstein. Animal population censusing at scale with citizen science and photographic identification. 2017. 4

[14] J. Redmon. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2016. 3

[15] J. Redmon and A. Farhadi. Yolo9000: Better, faster, stronger. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, pages 6517–6525. IEEE, 2017. 1, 2, 3

[16] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018. 4

[17] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *arXiv preprint arXiv:1506.01497*, 2015. 2

[18] O. Russakovsky, L.-J. Li, and L. Fei-Fei. Best of both worlds: human-machine collaboration for object annotation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2121–2131, 2015. 4

[19] M. Scott. Smart camera gimbal bot scanlime:027, Dec 2017. 4

[20] A. Shrivastava, R. Sukthankar, J. Malik, and A. Gupta. Beyond skip connections: Top-down modulation for object detection. *arXiv preprint arXiv:1612.06851*, 2016. 3

[21] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. 2017. 3

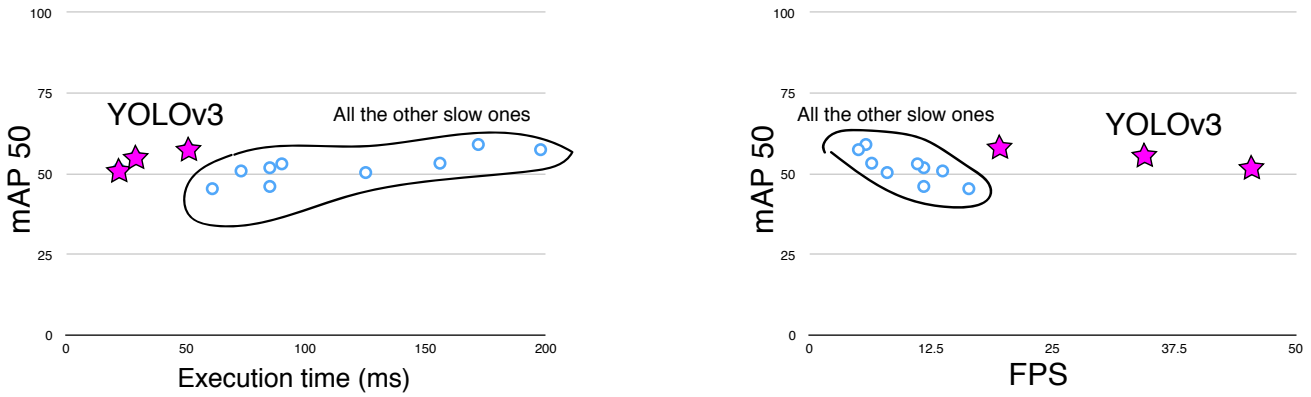


Figure 4. Zero-axis charts are probably more intellectually honest... and we can still screw with the variables to make ourselves look good!

Rebuttal

We would like to thank the Reddit commenters, labmates, emailers, and passing shouts in the hallway for their lovely, heartfelt words. If you, like me, are reviewing for ICCV then we know you probably have 37 other papers you could be reading that you'll invariably put off until the last week and then have some legend in the field email you about how you really should finish those reviews execept it won't entirely be clear what they're saying and maybe they're from the future? Anyway, this paper won't have become what it will in time be without all the work your past selves will have done also in the past but only a little bit further forward, not like all the way until now forward. And if you tweeted about it I wouldn't know. Just sayin.

Reviewer #2 AKA Dan Grossman (lol blinding who does that) insists that I point out here that our graphs have not one but two non-zero origins. You're absolutely right Dan, that's because it looks way better than admitting to ourselves that we're all just here battling over 2-3% mAP. But here are the requested graphs. I threw in one with FPS too because we look just like super good when we plot on FPS.

Reviewer #4 AKA JudasAdventus on Reddit writes "Entertaining read but the arguments against the MSCOCO metrics seem a bit weak". Well, I always knew you would be the one to turn on me Judas. You know how when you work on a project and it only comes out alright so you have to figure out some way to justify how what you did actually was pretty cool? I was basically trying to do that and I lashed out at the COCO metrics a little bit. But now that I've staked out this hill I may as well die on it.

See here's the thing, mAP is already sort of broken so an update to it should maybe address some of the issues with it or at least justify why the updated version is better in some way. And that's the big thing I took issue with was the lack of justification. For PASCAL VOC, the IOU threshold was "set deliberately low to account for inaccuracies in bounding boxes in the ground truth data" [2]. Does COCO have better labelling than VOC? This is definitely possible since COCO has segmentation masks maybe the labels are more trustworthy and thus we aren't as worried about inaccuracy. But again, my problem was the lack of justification.

The COCO metric emphasizes better bounding boxes but that emphasis must mean it de-emphasizes something else, in this case classification accuracy. Is there a good reason to think that more precise bounding boxes are more important than better classification? A miss-classified example is much more obvious than a bounding box that is slightly shifted.

mAP is already screwed up because all that matters is per-class rank ordering. For example, if your test set only has these two images then according to mAP two detectors that produce these results are JUST AS GOOD:

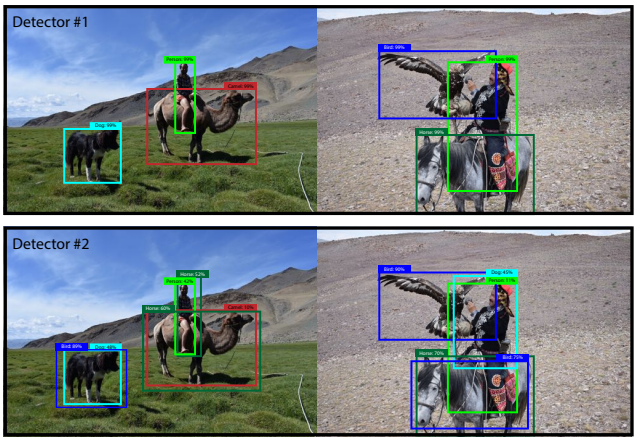


Figure 5. These two hypothetical detectors are perfect according to mAP over these two images. They are both perfect. Totally equal.

Now this is OBVIOUSLY an over-exaggeration of the problems with mAP but I guess my newly retconned point is that there are such obvious discrepancies between what people in the "real world" would care about and our current metrics that I think if we're going to come up with new metrics we should focus on these discrepancies. Also, like, it's already mean average precision, what do we even call the COCO metric, average mean average precision?

Here's a proposal, what people actually care about is given an image and a detector, how well will the detector find and classify objects in the image. What about getting rid of the per-class AP and just doing a global average precision? Or doing an AP calculation per-image and averaging over that?

Boxes are stupid anyway though, I'm probably a true believer in masks except I can't get YOLO to learn them.