ISTANBUL TECHNICAL UNIVERSITY, COMPUTER ENGINEERING

BLG561E Deep Learning Term Project Report

Anıl Öztürk
Mustafa Günel
Onur Gürışık
Sertaç İkizoğlu

504181504
511191124
504191126
504191128

January 7, 2020

1 THE PROJECT DEFINITION

The project consists of developing/training a deep neural network for object detection in urban driving scenarios. We developed a network that takes a continuous video stream as the input and detects driving-related objects in each frame of the video, along with the bounding boxes of each object. The objective was to build a high performance detector that can run using a reasonable framerate, hence both detection accuracy and size of our network is important in evaluating the performance of or network. The accuracy and speed will be normalized w.r.t the best performing model. The score will be determined as;

$$score = \frac{accuracy + speed}{2}$$

2 PREPARING THE DATA

We used Berkeley DeepDrive (BDD100K) and Eatron datasets to train our model. Both datasets had outliers and noise in their data, we cleaned them as well as we can. BDD100K had **more than 6 classes**, we filtered the unnecessary classes for better training.

The BDD100K data was discrete, made up by photos from different times and places. But the Eatron data were continuous video frames. Taking all the data from the eatron video-frames **would cause overfitting** on specific vehicle stereotypes and object locations. We skip every **2** frame and take only the **third** frame from the Eatron dataset.

We splitted the Eatron data as **30%** and **70%** for training and validation sets, respectively.

We decided to execute **2-phased training**;

- In the first phase, we train our model with BDD100K train set and validated it with BDD100K validation + Eatron validation set.
- In the second phase, we train our model with BDD100K train + Eatron train sets and validated it with Eatron validation set.

The reason of doing is the noisy structure of the Eatron data. The data wasn't including the vehicles that comes from the reverse direction. So in the first training which has to be robust one, we didn't include the company data into the training phase. But we included it into our validation. The second training phase was our **fine-tuning** phase, we did it for increase the score of our model on the company data, so we should be adding the data into the training split.

We decided to use a **YOLO based** model, a YOLO model wants the data in a specific format. We converted our data to that format for performing an errorless training.

3 CREATING THE MODEL

We decided to use a **YOLOv3** based model structure for a kick-start. The reason of choosing v3 is simply the **accuracy improvement** on the small images with a trade-off with small amount of inference time.

	FPS	mAP	Parameters
YOLOv3	20	57.9	140.69 Bn
Tiny YOLOv3	220	33.1	5.56 Bn

Table 3.1: The comparison of default YOLOs with pjreddie's test environment on COCO

We have wondered about which differences makes the vanilla YOLO better than the tiny version and the which of them affects that most.

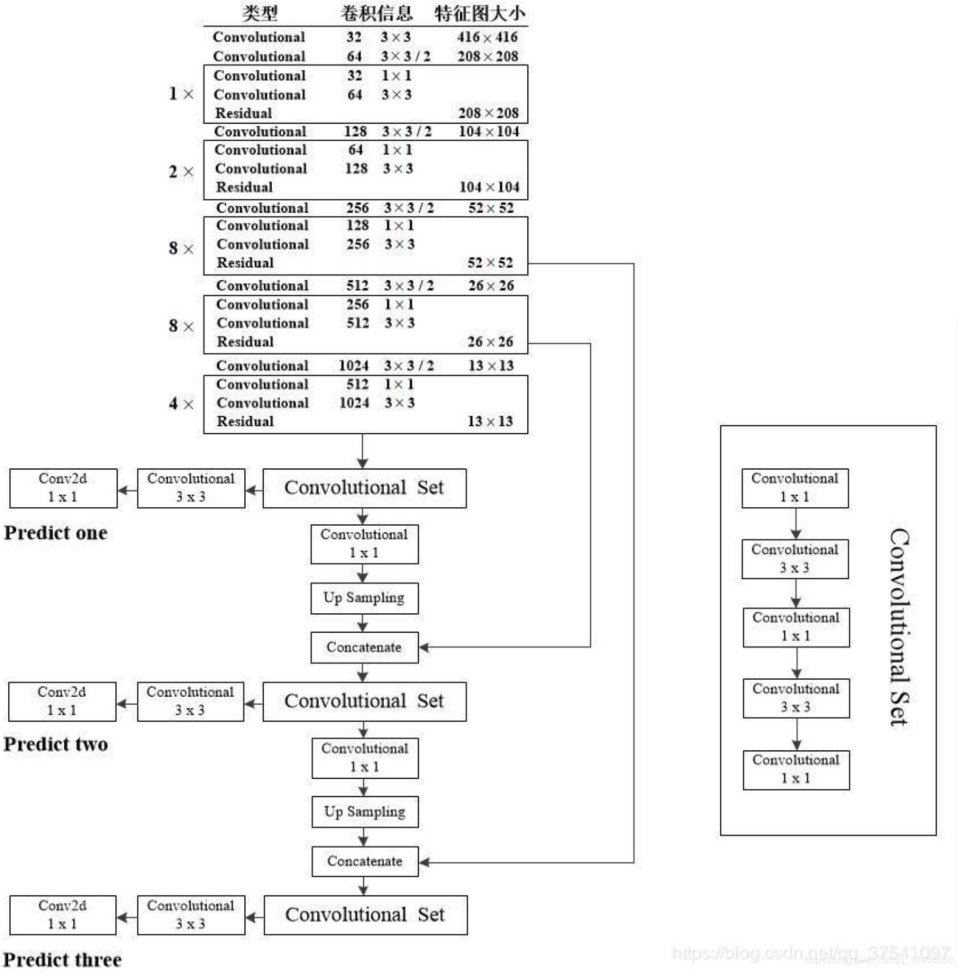


Figure 3.1: The architecture of the default YOLOv3

Layer	Type	Filters	Size/Stride	Input	Output
0	Convolutional	16	3 × 3/1	416 × 416 × 3	416 × 416 × 16
1	Maxpool		2 × 2/2	416 × 416 × 16	208 × 208 × 16
2	Convolutional	32	3 × 3/1	208 × 208 × 16	208 × 208 × 32
3	Maxpool		2 × 2/2	208 × 208 × 32	104 × 104 × 32
4	Convolutional	64	3 × 3/1	104 × 104 × 32	104 × 104 × 64
5	Maxpool		2 × 2/2	104 × 104 × 64	52 × 52 × 64
6	Convolutional	128	3 × 3/1	52 × 52 × 64	52 × 52 × 128
7	Maxpool		2 × 2/2	52 × 52 × 128	26 × 26 × 128
8	Convolutional	256	3 × 3/1	26 × 26 × 128	26 × 26 × 256
9	Maxpool		2 × 2/2	26 × 26 × 256	13 × 13 × 256
10	Convolutional	512	3 × 3/1	13 × 13 × 256	13 × 13 × 512
11	Maxpool		2 × 2/1	13 × 13 × 512	13 × 13 × 512
12	Convolutional	1024	3 × 3/1	13 × 13 × 512	13 × 13 × 1024
13	Convolutional	256	1 × 1/1	13 × 13 × 1024	13 × 13 × 256
14	Convolutional	512	3 × 3/1	13 × 13 × 256	13 × 13 × 512
15	Convolutional	255	1 × 1/1	13 × 13 × 512	13 × 13 × 255
16	YOLO				
17	**Route 13**				
18	Convolutional	128	1 × 1/1	13 × 13 × 256	13 × 13 × 128
19	Up-sampling		2 × 2/1	13 × 13 × 128	26 × 26 × 128
20	**Route 19 8**				
21	Convolutional	256	3 × 3/1	13 × 13 × 384	13 × 13 × 256
22	Convolutional	255	1 × 1/1	13 × 13 × 256	13 × 13 × 256
23	YOLO				

Figure 3.2: The architecture of the Tiny YOLOv3

We found out the one of the big differences affect the mAP score gap is **the third detection layer** in the vanilla YOLOv3. It processes the **earlier** feature maps of the network, so it actually tends to make predictions about **small objects**.

- We thought that would be a good idea to **include extra layer** into the Tiny YOLOv3 structure as an addition to the other changes. We implemented it with the same approach, we took a skip connection from the earlier feature maps of the network to feed that layer.
- We implemented another 512x512 convolutional layer after **Layer 10**
- Since we are having another detection layer and seperate convolutional operations for it, we would want to reduce the gradient count to preserve the speed advantage of the tiny model structure. We simply reduced **13th Layer** size to **128**, **14th Layer** size to **256**.
- Since we have **6** different classes for our task, we set the size of the convolutional layers right before the YOLO layers as $(5 + c) \times 3 = \mathbf{33}$
- The YOLO algorithm is based on regressing a set of pre-defined box sizes with a parametrised weights. These pre-defined sizes are called box priors or anchors. You should set them manually when you train on a custom dataset with your special classes. We calculated **9** most frequent priors (3 for big size object detection layer, 3 for small and 3 for medium size object detection layers) with runnning **k-means clustering** on BDD100K training set.

After all these changes, our model architecture is as follows;

Layer	Type	Filters	Size / Stride	Input	Output
0	Convolutional	16	3x3 / 1	416x416x3	416x416x16
1	Maxpool		2x2 / 2	416x416x16	208x208x16
2	Convolutional	32	3x3 / 1	208x208x16	208x208x32
3	Maxpool		2x2 / 2	208x208x32	104x104x32
4	Convolutional	64	3x3 / 1	104x104x32	104x104x64
5	Maxpool		2x2 / 2	104x104x64	52x52x64
6	Convolutional	128	3x3 / 1	52x52x64	52x52x128
7	Maxpool		2x2 / 2	52x52x128	26x26x128
8	Convolutional	256	3x3 / 1	26x26x128	26x26x256
9	Maxpool		2x2 / 2	26x26x256	13x13x256
10	Convolutional	512	3x3 / 1	13x13x256	13x13x512
11	Maxpool		2x2 / 1	13x13x512	13x13x512
12	Convolutional	512	3x3 / 1	13x13x512	13x13x512
13	Convolutional	1024	1x1 / 1	13x13x512	13x13x1024
14	Convolutional	128	1x1 / 1	13x13x1024	13x13x128
15	Convolutional	256	3x3 / 1	13x13x128	13x13x256
16	Convolutional	33	1x1 / 1	13x13x256	13x13x33
17	**YOLO Detection Layer**				
18	Route 14				
19	Convolutional	128	1x1 / 1	13x13x128	13x13x128
20	Up-sampling		2x2 / 1	13x13x128	26x26x128
21	Route 20, 8				
22	Convolutional	128	3x3 / 1	26x26x384	26x26x128
23	Convolutional	256	3x3 / 1	26x26x128	26x26x256
24	Convolutional	33	1x1 / 1	26x26x256	26x26x33
25	**YOLO Detection Layer**				
26	Route 23				
27	Convolutional	64	1x1 / 1	26x26x256	26x26x64
28	Up-sampling		2x2 / 1	26x26x64	52x52x128
29	Route 28, 5				
30	Convolutional	64	3x3 / 1	52x52x192	52x52x64
31	Convolutional	128	3x3 / 1	52x52x64	52x52x128
32	Convolutional	33	1x1 / 1	52x52x128	52x52x33
33	**YOLO Detection Layer**				

Table 3.2: Our custom YOLO network

4 TRAINING PHASE

While we training our model, we selected the best model checkpoint based on the **least validation loss** in the first training phase. In the second phase, we selected our best model with respect to the **best mAP** over the validation split.

4.1 ESSENTIAL SETTINGS

We trained our model on 2 Tesla-V100 GPUs with the **NVIDIA Apex** distribution support. We cached all the images in the VRAM before the training phase to minimize the data access and augmentation times. We used these essential parameters while training our model;

- **Epochs:** 273
- **Batch Size:** 64
- **Learning Rate:** 0.002
- **Optimizer:** SGD + Momentum
- **Multi-Scale:** On
- **Weighted Images:** On

After the training, we fine-tuned our model with the same parameters except the epoch count. To fine-tune, we trained our model on **50** epochs.

4.2 DATA AUGMENTATION

We performed data-augmentation when getting batches dynamically for our training phase. We used these type of augmentations;

- +/- 10% vertical and horizontal translation
- +/- 5% rotation
- +/- 2% shear
- +/- 10% scale
- 50% probability horizontal-reflection

After that, we dynamically created a grid from these augmented-images. The results are as follows.

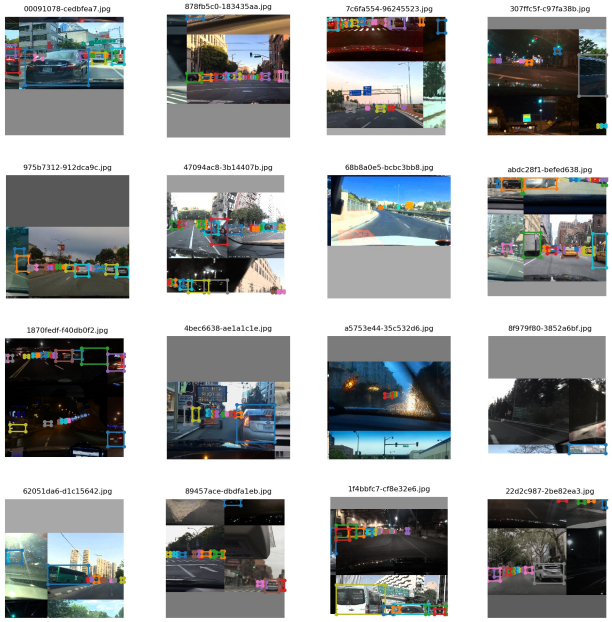


Figure 4.1: Sample augmented data for first-phase of the training

Figure 4.2: Sample augmented data for second-phase of the training (includes Eatron frames)

4.3 MULTI-SCALE APPROACH

We have used multi-scale approach while feeding images into our network in the training phase. We set the scale-up and scale-down multiplier as **1.5**.

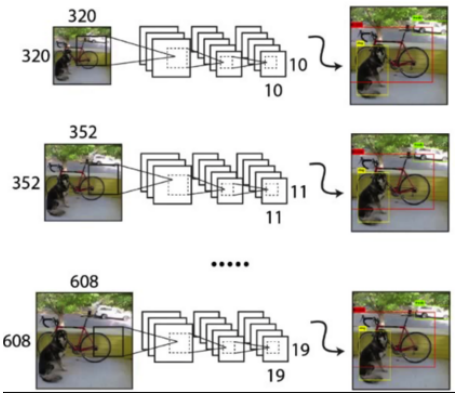


Figure 4.3: Multi-Scale Illustration

4.4 WEIGHTING IMAGES

We weight the images based on the loss on each class of the model while training. If a class' classification loss is higher than the others, the images that contains objects from that class more likely to be shown to our network in the process. This will bring faster-converging model as result.

5 TRAINING RESULTS

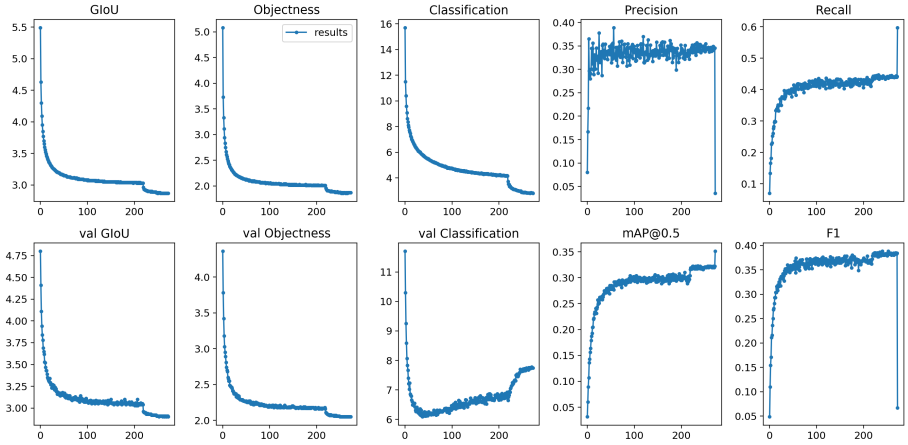


Figure 5.1: Results from first-phase of the training

In the fine-tuning process, the mAP and F1 values stayed as about what they were before. But we get an improvement over box regression, object confidence and classification losses.

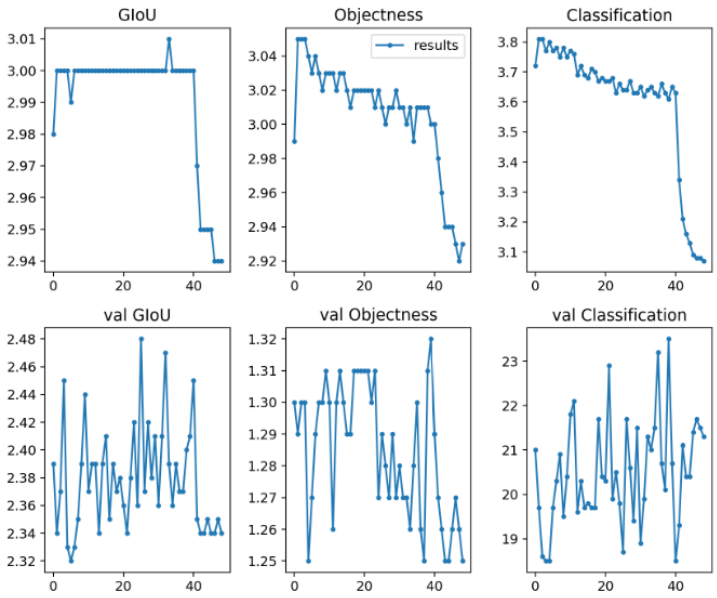
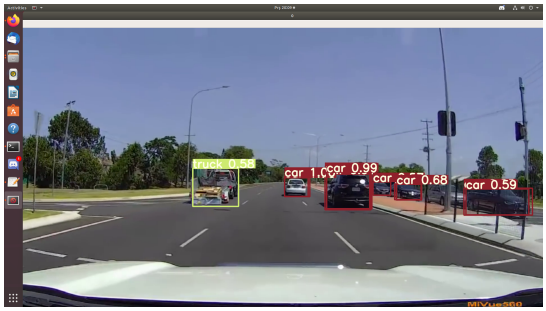


Figure 5.2: Results from second-phase of the training

6 Test Results

We tested our model on a downloaded dashcam video. The results were pretty promising.

(a) Inference example 1



(b) Inference example 2

7 CONCLUSION

We compared our custom model with YOLOv3 that trained on the COCO set and YOLOv3 Tiny that trained on the BDD100K set. All of the YOLOs run at **608x608** input resolution. We calculated their relative score with respect to each other, like in this competition. Our model has the highest score with a very large gap. We calculated these results on **Eatron Validation Set** with **NVIDIA GTX1080TI** and **Intel i9 9900K**.

Model	mAP	FPS	Parameter Count	Relative Score
YOLOv3 (COCO)	0.499	39.11	140.69 Bn	0.558
Tiny YOLOv3 (BDD100K)	0.299	**171.66**	**5.56 Bn**	0.765
Our YOLO	**0.563**	160.57	5.86 Bn	**0.968**

Table 7.1: Comparison between YOLOs

Our model has lost its inference speed a little bit compared to original YOLOv3 Tiny, but it almost **doubled** its mAP score. It even beats the vanilla YOLOv3.

8 REFERENCES

- TF-YOLO: An Improved Incremental Network for Real-Time Object Detection
- https://github.com/ultralytics/yolov3
- https://github.com/AlexeyAB/darknet