

# Alternate Vehicle Detector

Real-Time Object Detection Using Tiny YOLO

Santa Clara University

ELEN 645

By Mason Lee, Jackson Liu and Katherine Lu

# The Premise

We planned to implement a neural network able to identify “non-standard” vehicles which would appear on the road. This network would detect objects, label them with a bounding box, and work in real-time (i.e. 30 fps at a minimum). The work required included obtaining a pre-trained model, modifying code as needed to accommodate new classification labels, obtaining datasets, modifying data for training, testing our trained model, and real-time implementation for live video object detection.

## Preliminary Work

When starting off the project, we needed to get a better understanding of how CNN’s perform under different hyperparameters and architectures. Therefore we ran a series of experiments with MiniVgg, ShallowNet, Vgg6, Vgg8, and Vgg19 to study their outputs and understand how to tweak the parameters to our benefit.

### MiniVgg

```
Epoch 100/100
64/3011 [=====>.....] - ETA: 0s - loss: 0.0696 - acc: 0.968
384/3011 [==>.....] - ETA: 0s - loss: 0.1207 - acc: 0.947
704/3011 [=====>.....] - ETA: 0s - loss: 0.1484 - acc: 0.941
1024/3011 [=====>.....] - ETA: 0s - loss: 0.1450 - acc: 0.943
1344/3011 [=====>.....] - ETA: 0s - loss: 0.1474 - acc: 0.944
1664/3011 [=====>.....] - ETA: 0s - loss: 0.1384 - acc: 0.946
1984/3011 [=====>.....] - ETA: 0s - loss: 0.1472 - acc: 0.945
2304/3011 [=====>.....] - ETA: 0s - loss: 0.1426 - acc: 0.947
2624/3011 [=====>.....] - ETA: 0s - loss: 0.1394 - acc: 0.948
2880/3011 [=====>.....] - ETA: 0s - loss: 0.1435 - acc: 0.945
3011/3011 [=====>.....] - 1s 215us/step - loss: 0.1444 - acc: 0.9449 - val_loss: 1.2670 - val_acc: 0.6464
[INFO] serializing network...
[INFO] evaluating network...
      precision    recall   f1-score   support
  babaybuggy    0.76     0.57     0.65     327
        bike     0.58     0.75     0.65     311
  motorcycle    0.64     0.63     0.64     366
avg / total    0.66     0.65     0.65    1004
```

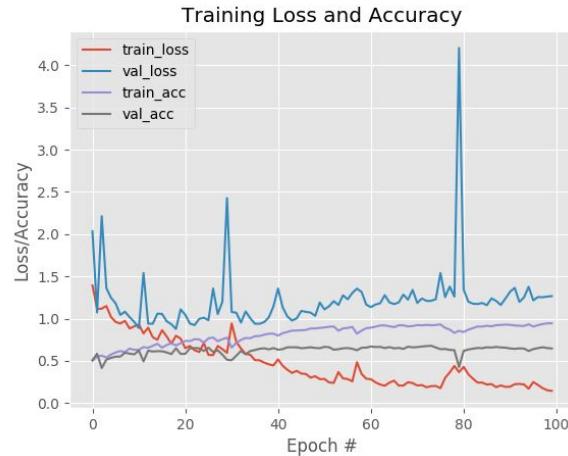


Figure 1. MiniVgg training results.

With a learning rate of 0.01, MiniVgg had was accurate up to around 65%. While the training loss steadily decreased over time, the validation loss was relatively constant. On the other side, training accuracy was much better than our validation accuracy, with some points reaching up to the high 90%'s. This in general shows that our model is overfitted to the training data.

## ShallowNet

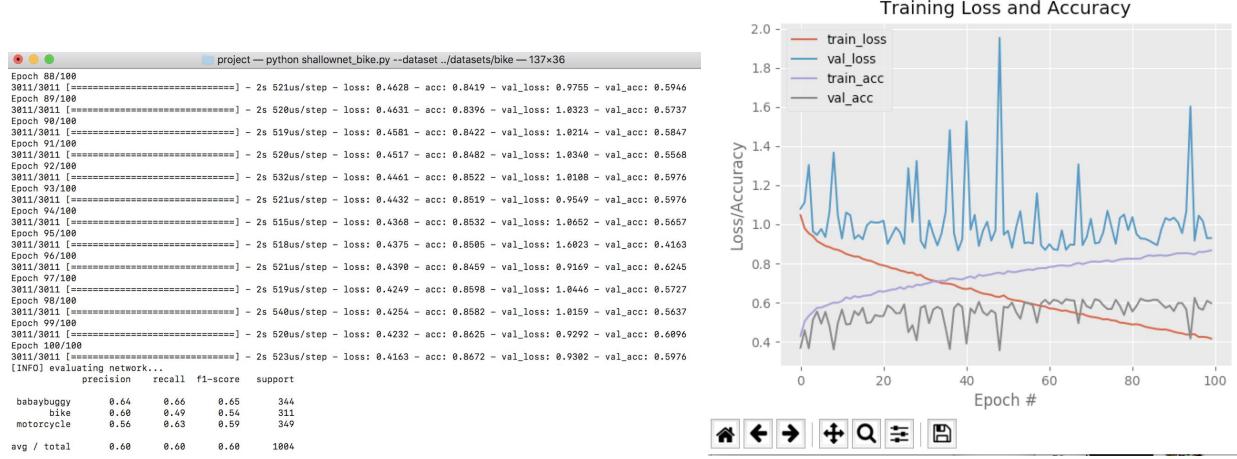


Figure 2. ShallowNet training results.

ShallowNet with a learning rate of 0.01 also showed characteristics of overfitting. If we look at the training loss, it decreases at a steady rate throughout training. However, validation loss is still relatively constant at a high value. When looking at training accuracy vs validation accuracy, we can also see that the two are not as converged as a well trained model would look.

## Vgg6

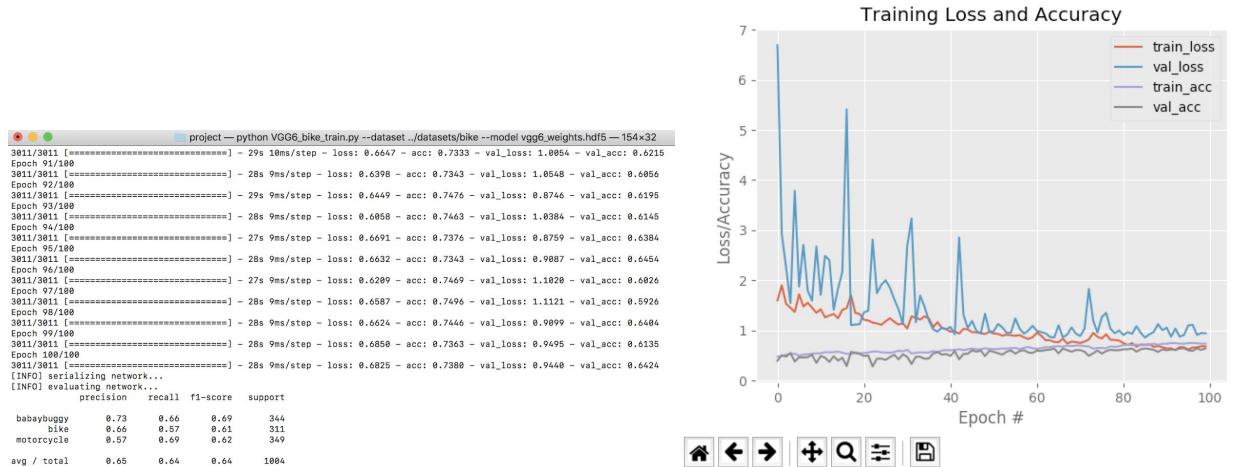


Figure 3. Vgg6 training results with learning rate of 0.01.

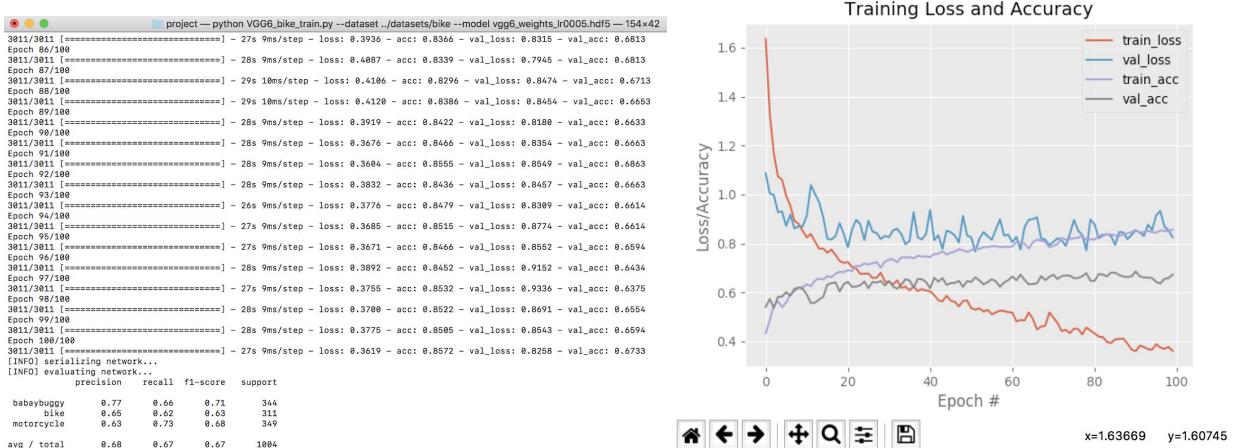


Figure 4. Vgg6 training results with learning rate 0.001.

In the Vgg6 architecture, we implemented six layers in this neural net and saw that tweaking the learning rate from 0.01 to 0.001 helped improve our accuracy by ~3-4%. This could be due to the fact that certain gradients are within a curve of the local minimum, but since our learning rate is 0.01, the point at which we get our gradient jumps back and forth between relatively the same positions. Lowering the learning rate in this case, would allow us to move between gradients at a smaller step, hence allowing us to reach lower in the “local” or “global” minimum to improve our results. As you can also see from the training and validation accuracy, the Vgg6 architecture produces much more correlated results than MiniVgg or Shallownet.

## Vgg8

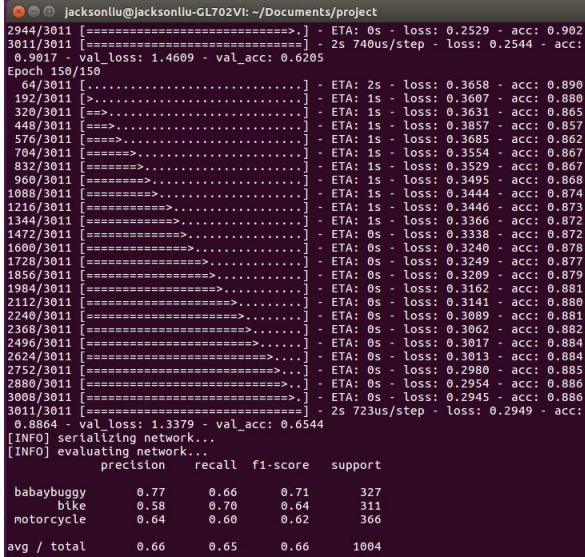
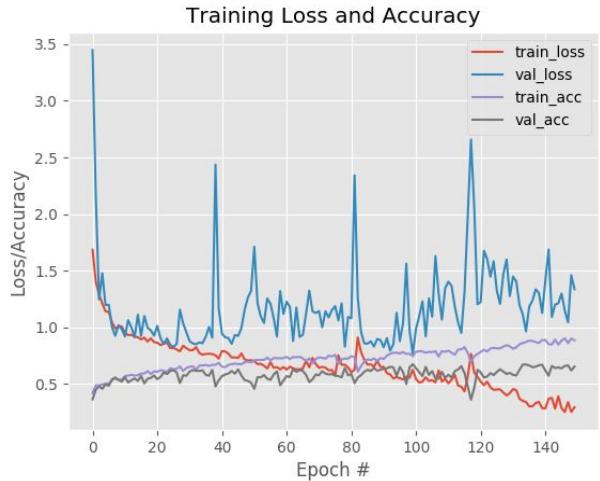


Figure 5. Vgg8 training results with learning rate 0.01.



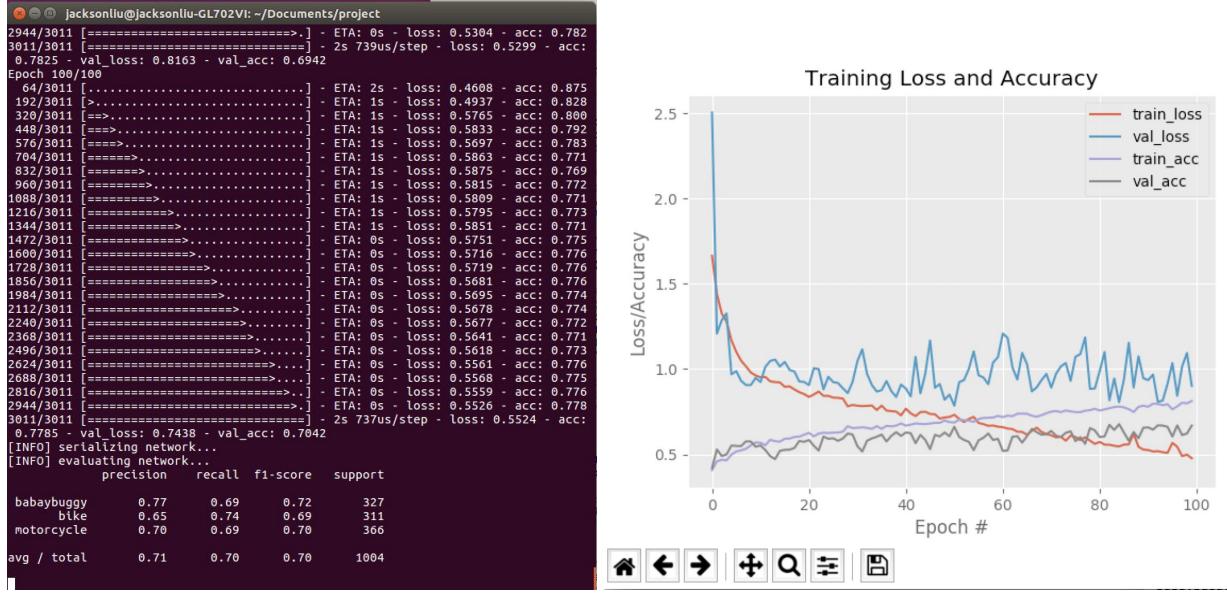


Figure 6. Vgg8 training results with learning rate 0.001.

Using the Vgg8 architecture, the precision of our model is now at ~71%, which is a great improvement from previous architectures tested before. Again, there is a bump in precision as we lower the learning rate. A tradeoff here would be that the lower the learning rate goes, the longer the time it takes to train the CNN. However when comparing the training loss and accuracy graphs above, you can see why sometimes the hit in run time pays off in raising your accuracy.

## Vgg19

```
jacksonliu@jacksonliu-GL702V1: ~/Documents/project
2624/3011 [=====>....] - ETA: 0s - loss: 2.4991 - acc: 0.365
2688/3011 [=====>....] - ETA: 0s - loss: 2.5126 - acc: 0.365
2752/3011 [=====>....] - ETA: 0s - loss: 2.4889 - acc: 0.365
2816/3011 [=====>....] - ETA: 0s - loss: 2.4744 - acc: 0.366
2880/3011 [=====>....] - ETA: 0s - loss: 2.4659 - acc: 0.366
2944/3011 [=====>....] - ETA: 0s - loss: 2.4741 - acc: 0.363
3008/3011 [=====>....] - ETA: 0s - loss: 2.4579 - acc: 0.361
3011/3011 [=====>....] - 3s 1ms/step - loss: 2.4606 - acc: 0
.3610 - val_loss: 2.5367 - val_acc: 0.4004
[INFO] serializing network...
[INFO] evaluating network...
/home/jacksonliu/.virtualenvs/dl4cv/lib/python3.5/site-packages/sklearn/metrics/
classification.py:1135: UndefinedMetricWarning: Precision and F-score are ill-de
fined and being set to 0.0 in labels with no predicted samples.
'precision', 'predicted', average, warn_for)
precision recall f1-score support
babaybuggy 0.64 0.15 0.24 327
bike 0.88 0.88 0.88 311
motorcycle 0.38 0.96 0.55 366
avg / total 0.35 0.40 0.28 1004
```



Figure 7. Vgg19 training results with learning rate 0.01.

After seeing the constant improvement as we increased the number of layers in our CNN, we decided to test out a 19 layer neural network. This did not give us the increase in performance we expected. After increasing our Vgg8 to a Vgg19 architecture, our accuracy went down by

almost 40%! In our opinion, we concluded that this could be due to the vanishing gradient problem, and the fact that our datasets and amount of data that we have do not yield a complex enough problem to justify so many layers within our network. Essentially the networks in the last few layers closest to the Fully Connected Layer were being updated with correct weights and biases, but the first half or more layers were not being updated at all because the updates that it received were almost dismissable by the time they reach these layers, hence yielding terrible results.

## Fine tuning transfer learning with original model

We were not satisfied with the accuracy of any of our models up to this point. Instead we tried to create our own model based on the Vgg architecture.

Since we were stuck at 70% accuracy by training our own model from scratch, we used a new strategy --- fine tuning transfer learning. This method allowed us using a pretrained model as our kickoff point to train our own model. The way to apply transfer learning is by cutting off the original fully connected layer and replacing it with our own fully connected layer, as shown in Figure 8. We first froze the original convolutional layers and warmed up our own fully connected layers by training on our dataset for 10 epochs. Then we opened up the convolutional layers and allowed back propagation to update the weights in the original convolutional layers. We found this method very powerful, as it boosted our accuracy up to 95% (as shown in Figure 9).

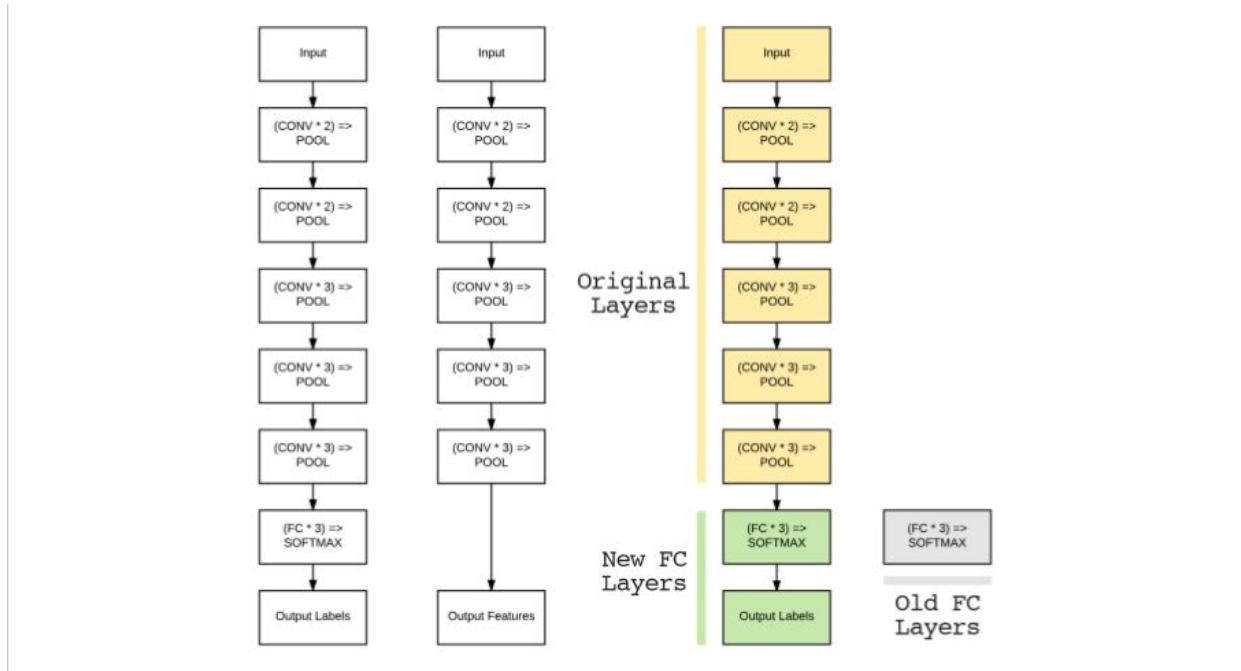


Figure 8. Fine tuning transfer learning method.

```

jacksonliu@jacksonliu-GL702VI: ~/Documents/project
Epoch 81/100
93/93 [=====] - 25s 270ms/step - loss: 0.0386 - acc: 0.9863 - val_loss: 0.2027 - val_acc: 0.9559
Epoch 82/100
93/93 [=====] - 25s 271ms/step - loss: 0.0292 - acc: 0.9899 - val_loss: 0.2077 - val_acc: 0.9569
Epoch 83/100
93/93 [=====] - 25s 270ms/step - loss: 0.0277 - acc: 0.9913 - val_loss: 0.2171 - val_acc: 0.9529
Epoch 84/100
93/93 [=====] - 25s 270ms/step - loss: 0.0279 - acc: 0.9900 - val_loss: 0.2225 - val_acc: 0.9509
Epoch 85/100
93/93 [=====] - 25s 268ms/step - loss: 0.0273 - acc: 0.9906 - val_loss: 0.2295 - val_acc: 0.9569
Epoch 86/100
93/93 [=====] - 25s 270ms/step - loss: 0.0247 - acc: 0.9903 - val_loss: 0.2648 - val_acc: 0.9529
Epoch 87/100
93/93 [=====] - 25s 271ms/step - loss: 0.0230 - acc: 0.9926 - val_loss: 0.2186 - val_acc: 0.9579
Epoch 88/100
93/93 [=====] - 25s 271ms/step - loss: 0.0242 - acc: 0.9913 - val_loss: 0.2377 - val_acc: 0.9559
Epoch 89/100
93/93 [=====] - 25s 265ms/step - loss: 0.0194 - acc: 0.9936 - val_loss: 0.2100 - val_acc: 0.9529
Epoch 90/100
93/93 [=====] - 25s 269ms/step - loss: 0.0181 - acc: 0.9956 - val_loss: 0.2295 - val_acc: 0.9549
Epoch 91/100
93/93 [=====] - 25s 264ms/step - loss: 0.0202 - acc: 0.9926 - val_loss: 0.2470 - val_acc: 0.9539
Epoch 92/100
93/93 [=====] - 25s 266ms/step - loss: 0.0233 - acc: 0.9923 - val_loss: 0.2319 - val_acc: 0.9559
Epoch 93/100
93/93 [=====] - 25s 267ms/step - loss: 0.0307 - acc: 0.9926 - val_loss: 0.2163 - val_acc: 0.9549
Epoch 94/100
93/93 [=====] - 25s 266ms/step - loss: 0.0168 - acc: 0.9953 - val_loss: 0.2489 - val_acc: 0.9559
Epoch 95/100
93/93 [=====] - 25s 272ms/step - loss: 0.0237 - acc: 0.9933 - val_loss: 0.2127 - val_acc: 0.9549
Epoch 96/100
93/93 [=====] - 27s 294ms/step - loss: 0.0252 - acc: 0.9909 - val_loss: 0.2381 - val_acc: 0.9589
Epoch 97/100
93/93 [=====] - 27s 289ms/step - loss: 0.0199 - acc: 0.9943 - val_loss: 0.2351 - val_acc: 0.9559
Epoch 98/100
93/93 [=====] - 28s 297ms/step - loss: 0.0180 - acc: 0.9950 - val_loss: 0.2300 - val_acc: 0.9559
Epoch 99/100
93/93 [=====] - 27s 290ms/step - loss: 0.0250 - acc: 0.9913 - val_loss: 0.2418 - val_acc: 0.9569
Epoch 100/100
93/93 [=====] - 27s 292ms/step - loss: 0.0254 - acc: 0.9929 - val_loss: 0.2094 - val_acc: 0.9539
[INFO] evaluating after fine-tuning...
      precision    recall   fi-score   support
  babybuggy     0.92     0.96     0.94      311
    bicycle      0.96     0.91     0.93      329
  fire_engine    0.98     0.99     0.98      358
avg / total     0.95     0.95     0.95      998

[INFO] serializing model...
(dl4cv) jacksonliu@jacksonliu-GL702VI:~/Documents/project$ 

```

Figure 9. Fine Tuning result.

## Learnings

After running experiments with MiniVgg, ShallowNet, and various-layered Vgg architectures, we learned that since our dataset is rather small and not too complex, that using smaller architectures are actually better than using deeper networks, which eventually was the reason why we chose to use Tiny Yolo, rather than the full Yolo architecture.

Another reason why we chose to use smaller networks, was because of the “vanishing gradient” problem that we encountered in our Vgg19 test run. The reason why we believe that the inaccurate data was caused by the vanishing gradient problem, was because if our network is too deep, with too many layers, and our dataset not as complex, then this would mean that the gradients that are being backpropogated would be a smaller number (in the range of 0-1). In small neural network architectures, this small gradient would not be as big of an issue, because by the time it reaches the first layer in your network, it would still be a meaningful update to the weights and biases for this layer. However, given a deeper network and the fact that each layer’s update on their weights and biases depends on the gradient of the previous layer, each

gradient would be even smaller than the previous gradient (decimals in the range 0->1 multiplied against each other would yield a smaller number). In a deep network, this would cause the gradient to eventually “vanish” and become an extremely small, meaningless update to the weights in the first few layers of your neural net.

## Darknet-YOLO

### What is YOLO?

You only look once (YOLO) is a real-time object detection system which applies a single neural network to the full image. This network divides the image into regions and predicts bounding boxes and probabilities for each region. The bounding boxes are weighted by predicted probabilities. We chose to use this network since it was purported to be both fast and accurate.

YOLO v3 has 106 fully convolutional layers and makes detections at 3 scales. It also has no softmax layer, as it allows multiple labels per item detected.

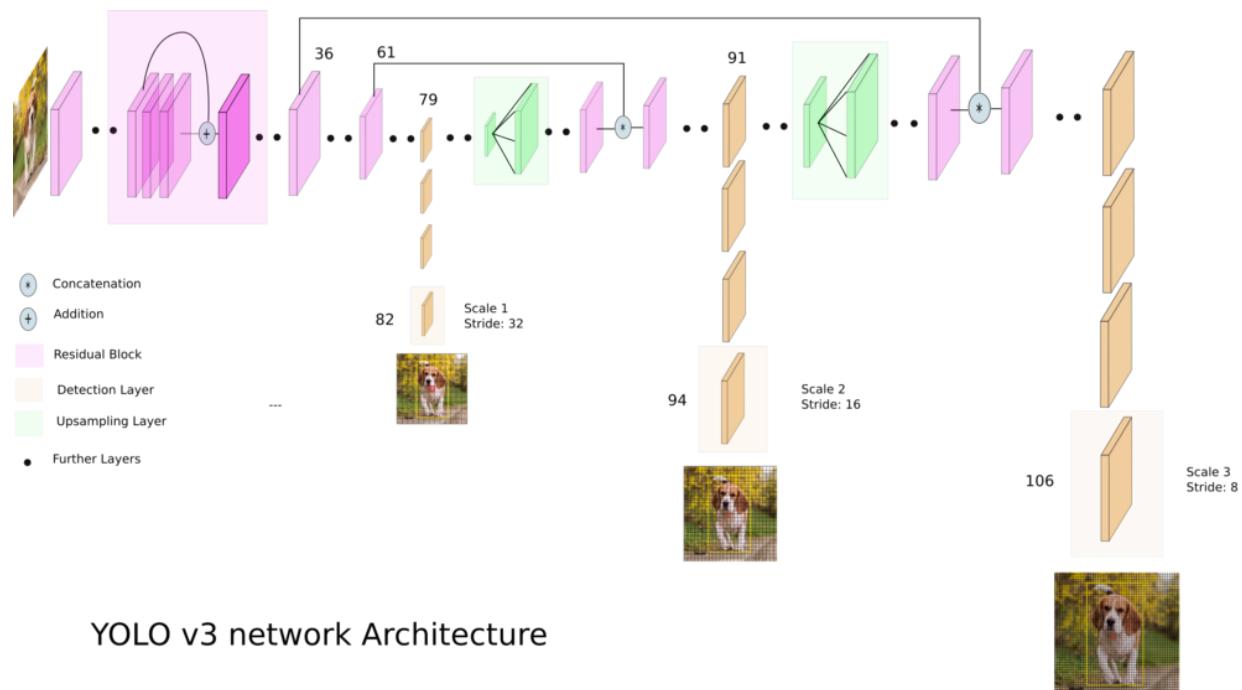


Figure 10: YOLO v3 network architecture overview. (Source: <https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b>)

```
[d14cv] jacksonliu@jacksonliu-GL702VI:~/darknet$ ./darknet detector train cfg/obj6.data cfg/yolo_v3_obj.cfg
layer  filters size input output
0 conv  32 3 x 3 / 1 416 x 416 x 3 -> 416 x 416 x 32 0.299 BFLOPs
1 conv  64 3 x 3 / 2 416 x 416 x 32 -> 208 x 208 x 64 1.595 BFLOPs
2 conv  32 3 x 3 / 1 208 x 208 x 64 -> 208 x 208 x 32 0.299 BFLOPs
3 conv  64 3 x 3 / 1 208 x 208 x 32 -> 208 x 208 x 64 1.595 BFLOPs
4 res 1 208 x 208 x 64 -> 208 x 208 x 64
5 conv  128 3 x 3 / 2 208 x 208 x 64 -> 104 x 104 x 128 0.177 BFLOPs
6 conv  64 1 x 1 / 1 104 x 104 x 128 -> 104 x 104 x 64 0.177 BFLOPs
7 conv  128 3 x 3 / 1 104 x 104 x 64 -> 104 x 104 x 128 1.595 BFLOPs
8 res 5 104 x 104 x 128 -> 104 x 104 x 128
9 conv  64 1 x 1 / 1 104 x 104 x 128 -> 104 x 104 x 64 0.177 BFLOPs
10 conv  128 3 x 3 / 1 104 x 104 x 64 -> 104 x 104 x 128 1.595 BFLOPs
11 res 8 104 x 104 x 128 -> 104 x 104 x 128
12 conv  256 3 x 3 / 1 104 x 104 x 128 -> 52 x 52 x 256 1.595 BFLOPs
13 conv  128 1 x 1 / 1 52 x 52 x 256 -> 52 x 52 x 128 0.177 BFLOPs
14 conv  256 3 x 3 / 1 52 x 52 x 128 -> 52 x 52 x 256 1.595 BFLOPs
15 res 12 52 x 52 x 256 -> 52 x 52 x 256
16 conv  128 3 x 3 / 1 52 x 52 x 256 -> 52 x 52 x 128 0.177 BFLOPs
17 conv  256 3 x 3 / 1 52 x 52 x 128 -> 52 x 52 x 256 1.595 BFLOPs
18 res 15 52 x 52 x 256 -> 52 x 52 x 256
19 conv  128 1 x 1 / 1 52 x 52 x 256 -> 52 x 52 x 128 0.177 BFLOPs
20 conv  256 3 x 3 / 1 52 x 52 x 128 -> 52 x 52 x 256 1.595 BFLOPs
21 res 18 52 x 52 x 256 -> 52 x 52 x 256
22 conv  128 3 x 3 / 1 52 x 52 x 256 -> 52 x 52 x 128 0.177 BFLOPs
23 conv  256 3 x 3 / 1 52 x 52 x 128 -> 52 x 52 x 256 1.595 BFLOPs
24 res 21 52 x 52 x 256 -> 52 x 52 x 256
25 conv  128 1 x 1 / 1 52 x 52 x 256 -> 52 x 52 x 128 0.177 BFLOPs
26 conv  256 3 x 3 / 1 52 x 52 x 128 -> 52 x 52 x 256 1.595 BFLOPs
27 res 24 52 x 52 x 256 -> 52 x 52 x 256
28 conv  128 1 x 1 / 1 52 x 52 x 256 -> 52 x 52 x 128 0.177 BFLOPs
29 conv  256 3 x 3 / 1 52 x 52 x 128 -> 52 x 52 x 256 1.595 BFLOPs
30 res 27 52 x 52 x 256 -> 52 x 52 x 256
31 conv  128 1 x 1 / 1 52 x 52 x 256 -> 52 x 52 x 128 0.177 BFLOPs
32 conv  256 3 x 3 / 1 52 x 52 x 128 -> 52 x 52 x 256 1.595 BFLOPs
33 res 30 52 x 52 x 256 -> 52 x 52 x 256
34 conv  128 1 x 1 / 1 52 x 52 x 256 -> 52 x 52 x 128 0.177 BFLOPs
35 conv  256 3 x 3 / 1 52 x 52 x 128 -> 52 x 52 x 256 1.595 BFLOPs
36 res 33 52 x 52 x 256 -> 52 x 52 x 256
37 conv  512 3 x 3 / 2 52 x 52 x 256 -> 26 x 26 x 512 1.595 BFLOPs
38 conv  256 3 x 3 / 1 26 x 26 x 512 -> 26 x 26 x 256 0.177 BFLOPs
39 conv  512 3 x 3 / 1 26 x 26 x 256 -> 26 x 26 x 512 1.595 BFLOPs
40 res 37 26 x 26 x 512 -> 26 x 26 x 512
41 conv  256 1 x 1 / 1 26 x 26 x 512 -> 26 x 26 x 256 0.177 BFLOPs
42 conv  512 3 x 3 / 1 26 x 26 x 256 -> 26 x 26 x 512 1.595 BFLOPs
43 res 40 26 x 26 x 512 -> 26 x 26 x 512
44 conv  256 1 x 1 / 1 26 x 26 x 512 -> 26 x 26 x 256 0.177 BFLOPs
45 conv  512 3 x 3 / 1 26 x 26 x 256 -> 26 x 26 x 512 1.595 BFLOPs
46 res 43 26 x 26 x 512 -> 26 x 26 x 512
47 conv  256 1 x 1 / 1 26 x 26 x 512 -> 26 x 26 x 256 0.177 BFLOPs
48 conv  512 3 x 3 / 1 26 x 26 x 256 -> 26 x 26 x 512 1.595 BFLOPs
49 conv  256 1 x 1 / 1 26 x 26 x 512 -> 26 x 26 x 256 0.177 BFLOPs
50 conv  512 3 x 3 / 1 26 x 26 x 256 -> 26 x 26 x 512 1.595 BFLOPs
51 conv  256 1 x 1 / 1 26 x 26 x 512 -> 26 x 26 x 256 0.177 BFLOPs
52 res 49 26 x 26 x 512 -> 26 x 26 x 512
53 conv  512 1 x 1 / 1 26 x 26 x 512 -> 26 x 26 x 256 0.177 BFLOPs
54 conv  256 1 x 1 / 1 26 x 26 x 256 -> 26 x 26 x 512 1.595 BFLOPs
55 res 52 26 x 26 x 512 -> 26 x 26 x 512
```

Figure 11. YOLO v3 network architecture as displayed when run on the command console.

## YOLO Detection

YOLOv3 predicts boxes at 3 different scales. The YOLO system extracts features from those scales using a similar concept to feature pyramid networks [2]. From the base feature extractor the authors added several convolutional layers. The last of these predicts a 3-d tensor encoding bounding box, objectness, and class predictions. In official YOLO experiments with COCO [3] they 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. In our experiments, we have 6 class predictions, thus total number filter is  $N \times N \times [3 * (4 + 1 + 6)]$ .

Next YOLO takes the feature map from 2 layers previous and upsample it by 2x. The authors also take a feature map from earlier in the network and merge it with their upsampled features using concatenation. This method allows the authors to get more meaningful semantic information from the upsampled features and finer-grained information from the earlier feature map. The authors then add a few more convolutional layers to process this combined feature map, and eventually predict a similar tensor, although now twice the size. They perform the same design one more time to predict boxes for the final scale. Thus their predictions for the 3rd scale benefit from all the prior computation as well as fine-grained features from early on in the network.

They still use k-means clustering to determine their bounding box priors. In the full version of YOLOv3, they choose 9 clusters and 3 scales arbitrarily and then divide up the clusters evenly across scales [1]. On the COCO dataset the 9 clusters were:

(10×13),(16×30),(33×23),(30×61),(62×45),(59×119),(116 × 90),(156 × 198),(373 × 326).

For tiny YOLO, we used 6 clusters which were:  
 $(10 \times 14)$ ,  $(23 \times 27)$ ,  $(37 \times 58)$ ,  $(81 \times 82)$ ,  $(135 \times 169)$ ,  $(344 \times 319)$

For further discussion of how YOLO works, you may want to check out the following papers and articles:

<https://pjreddie.com/media/files/papers/YOLOv3.pdf>

<http://machinethink.net/blog/object-detection-with-yolo/>

<https://blog.paperspace.com/how-to-implement-a-yolo-object-detector-in-pytorch/>

<https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b>

<https://hackernoon.com/understanding-yolo-f5a74bbc7967>

## Tiny YOLO

Tiny YOLOv3, meant to be used for resource-constrained environments, has 23 layers in a similar architecture to YOLOv3. This is the network we used, as it is faster (having fewer layers) and trains better given our small datasets.

Layer	filters	size	input		output	
0 conv	16	$3 \times 3 / 1$	$416 \times 416 \times 3$	->	$416 \times 416 \times 16$	0.150 BFLOPs
1 max		$2 \times 2 / 2$	$416 \times 416 \times 16$	->	$208 \times 208 \times 16$	
2 conv	32	$3 \times 3 / 1$	$208 \times 208 \times 16$	->	$208 \times 208 \times 32$	0.399 BFLOPs
3 max		$2 \times 2 / 2$	$208 \times 208 \times 32$	->	$104 \times 104 \times 32$	
4 conv	64	$3 \times 3 / 1$	$104 \times 104 \times 32$	->	$104 \times 104 \times 64$	0.399 BFLOPs
5 max		$2 \times 2 / 2$	$104 \times 104 \times 64$	->	$52 \times 52 \times 64$	
6 conv	128	$3 \times 3 / 1$	$52 \times 52 \times 64$	->	$52 \times 52 \times 128$	0.399 BFLOPs
7 max		$2 \times 2 / 2$	$52 \times 52 \times 128$	->	$26 \times 26 \times 128$	
8 conv	256	$3 \times 3 / 1$	$26 \times 26 \times 128$	->	$26 \times 26 \times 256$	0.399 BFLOPs
9 max		$2 \times 2 / 2$	$26 \times 26 \times 256$	->	$13 \times 13 \times 256$	
10 conv	512	$3 \times 3 / 1$	$13 \times 13 \times 256$	->	$13 \times 13 \times 512$	0.399 BFLOPs
11 max		$2 \times 2 / 1$	$13 \times 13 \times 512$	->	$13 \times 13 \times 512$	
12 conv	1024	$3 \times 3 / 1$	$13 \times 13 \times 512$	->	$13 \times 13 \times 1024$	1.595 BFLOPs
13 conv	256	$1 \times 1 / 1$	$13 \times 13 \times 1024$	->	$13 \times 13 \times 256$	0.089 BFLOPs
14 conv	512	$3 \times 3 / 1$	$13 \times 13 \times 256$	->	$13 \times 13 \times 512$	0.399 BFLOPs
15 conv	33	$1 \times 1 / 1$	$13 \times 13 \times 512$	->	$13 \times 13 \times 33$	0.006 BFLOPs
16 detection						
17 route	13					
18 conv	128	$1 \times 1 / 1$	$13 \times 13 \times 256$	->	$13 \times 13 \times 128$	0.011 BFLOPs
19 upsample		2x	$13 \times 13 \times 128$	->	$26 \times 26 \times 128$	
20 route	19 8					
21 conv	256	$3 \times 3 / 1$	$26 \times 26 \times 384$	->	$26 \times 26 \times 256$	1.196 BFLOPs
22 conv	33	$1 \times 1 / 1$	$26 \times 26 \times 256$	->	$26 \times 26 \times 33$	0.011 BFLOPs
23 detection						

Loading weights from backup/yolov3-tiny6\_410000.weights...Done!

Figure 12: Tiny YOLOv3 network architecture.

## Alternate Vehicle Detector (AVD) with Tiny YOLO

### Equipment

We used the following equipment and environments:

- OS: Ubuntu 16.04

- CUDA 9.0
- cuDNN 7.0
- GPU:Nvidia Geforce GTX 1080
- Tensorflow 1.8.0
- OpenCV 3.3.0
- Keras 2.1.6

## Classes

Our network detects 6 classes: bicycle, baby buggy, fire engine, motorcycle, scooter, and motor-scooter.

## Data Annotation

During our preliminary exploration we used Keras to build a classification model on those vehicles; the data then was labeled by identifying whole images under a specific label. However, YOLO requires training data which includes location information within the image as well as a classification label. We created this training label data manually.

The data annotation tool we used is Yolo-mark: [https://github.com/AlexeyAB/Yolo\\_mark](https://github.com/AlexeyAB/Yolo_mark). The UI of this tool is shown below, in Figure 13. It allowed users to define a bounding box with a label and would generate a text file associated with the image containing the relevant label data.

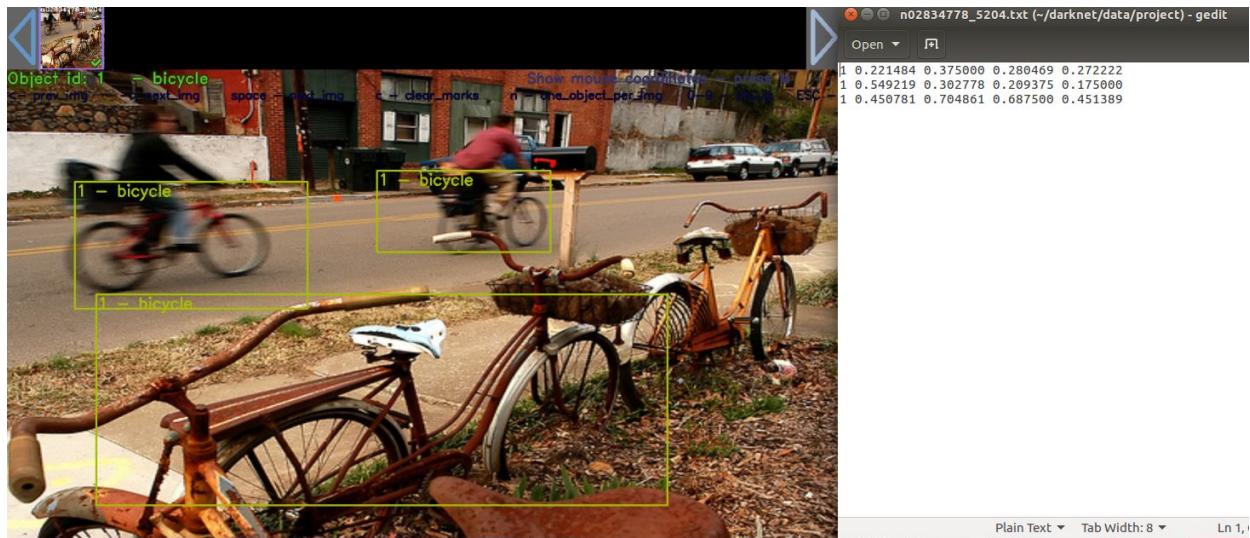


Figure 13. Yolo-mark UI and output.

The first column in the generated text file is the class index according to the index of names the *names.txt* file. The following columns are relative x-center, y-center, width and height. Here is the source code of calculating those four location information, shown in Figure 14. Since our data points have different dimensions, the yolo-mark UI would resize the input images and

display the resized version of the image. The height and width values are a relative ratio instead of absolute location information to avoid such problem.

```
for (auto &i : current_coord_vec)
{
    float const relative_center_x = (float)(i.abs_rect.x + i.abs_rect.width / 2) / full_image_roi.cols;
    float const relative_center_y = (float)(i.abs_rect.y + i.abs_rect.height / 2) / full_image_roi.rows;
    float const relative_width = (float)i.abs_rect.width / full_image_roi.cols;
    float const relative_height = (float)i.abs_rect.height / full_image_roi.rows;

    if (relative_width <= 0) continue;
    if (relative_height <= 0) continue;
    if (relative_center_x <= 0) continue;
    if (relative_center_y <= 0) continue;
```

Figure 14. Source code of Yolo-mark localization.

After annotating 7800+ images, we had a dataset as shown in Figure 15. Each image has a corresponding .txt file containing the class and localization information.

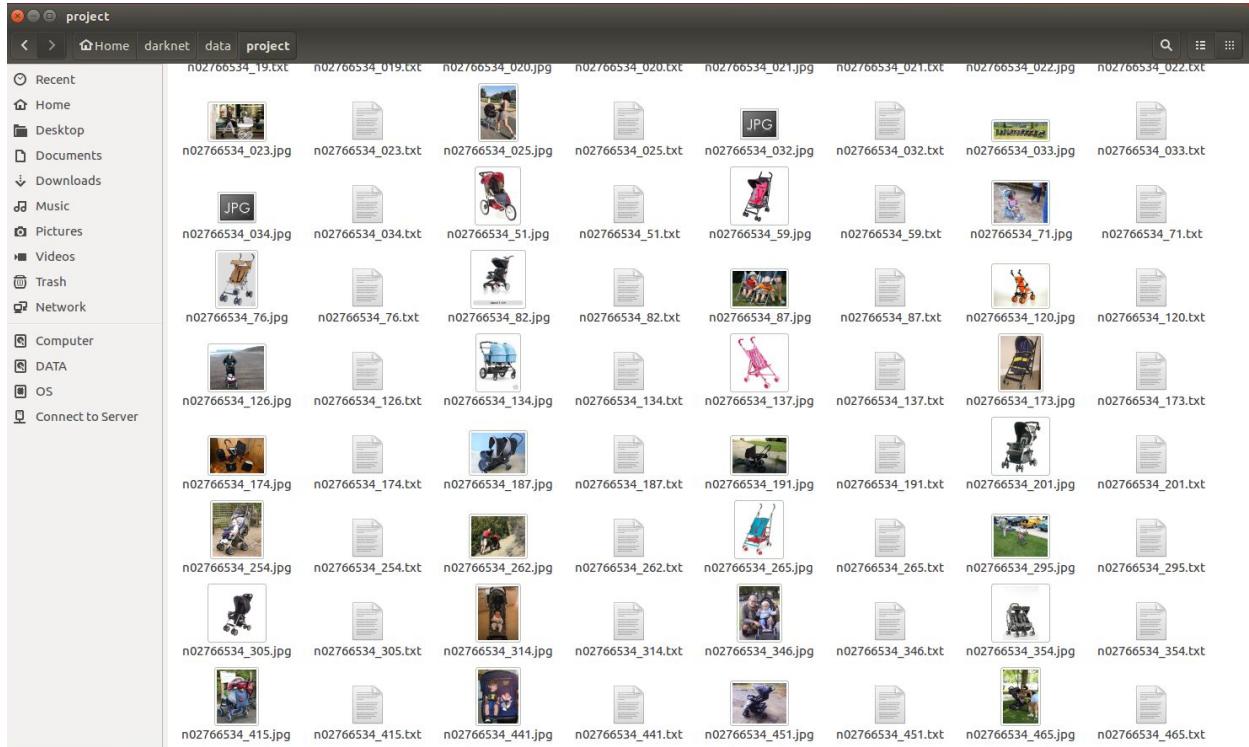


Figure 15. Data annotation file

## Feature extraction

The training strategy we used for YOLO was feature extraction. We cut a pre-trained model called darknet53 at 74th layer. The logic behind feature extraction is that by cutting a pre-trained model at any arbitrary layer, we can use the trained weights at that layer to be our CNN's feature extractor. Even though the pre-trained model may applied on a completely different

datasets, we are still able to obtain a high performance model by using the weights from the pre-trained model to train our own datasets.

## Training

We first tried to build our model according to <https://github.com/AlexeyAB/darknet>. Alexey suggests using the YOLOv3 neural network architecture to train on our customized object detection task. It did not work out well. We were running into two problems: GPU out of memory if we set the validation batch size greater than or equals to 24; or the validation accuracy becoming very low after training for a long time due to the vanishing gradient problem.

At this point, we changed our model from YOLOv3 to tiny-YOLOv3 to avoid the vanishing gradient problem.

We have some screenshots of training process. If you look closer in Figure 16, in the first 200 iterations the “Class: 0.500382” which represents the training accuracy is 50.0382%, and the “391.753906” which represents the training loss is 391.753906%. We have a low training accuracy and high training loss initially, but as the training process progressed, as shown in Figure 17, most training accuracies were over 99% and the training losses were about 0.95. In our final model, the training loss is about 0.35%.

```
jacksonliu@jacksonliu-GL702VI:~/darknet
Region 23 Avg IOU: 0.689773, Class: 0.497174, Obj: 0.444755, No Obj: 0.444861, .5R: 1.000000, .75R: 0.333333, count: 3
189: 348.126068, 396.774384 avg, 0.000001 rate, 0.287893 seconds, 4536 images
Loaded: 0.000045 seconds
Region 16 Avg IOU: 0.563869, Class: 0.500382, Obj: 0.484603, No Obj: 0.484387, .5R: 0.750000, .75R: 0.000000, count: 4
Region 23 Avg IOU: 0.564091, Class: 0.496128, Obj: 0.443318, No Obj: 0.443431, .5R: 1.000000, .75R: 0.000000, count: 2
Region 16 Avg IOU: 0.571973, Class: 0.501192, Obj: 0.484391, No Obj: 0.484387, .5R: 1.000000, .75R: 0.000000, count: 3
Region 23 Avg IOU: 0.560089, Class: 0.495958, Obj: 0.443231, No Obj: 0.443429, .5R: 1.000000, .75R: 0.000000, count: 1
Region 16 Avg IOU: 0.522983, Class: 0.501563, Obj: 0.484004, No Obj: 0.484386, .5R: 0.500000, .75R: 0.000000, count: 4
Region 23 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.443441, .5R: -nan, .75R: -nan, count: 0
Region 16 Avg IOU: 0.535575, Class: 0.499333, Obj: 0.484135, No Obj: 0.484387, .5R: 0.666667, .75R: 0.000000, count: 3
Region 23 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.443444, .5R: -nan, .75R: -nan, count: 0
Region 16 Avg IOU: 0.527243, Class: 0.499849, Obj: 0.485095, No Obj: 0.484387, .5R: 0.666667, .75R: 0.000000, count: 3
Region 23 Avg IOU: 0.536569, Class: 0.496834, Obj: 0.443330, No Obj: 0.443433, .5R: 0.666667, .75R: 0.000000, count: 3
Region 16 Avg IOU: 0.688669, Class: 0.500556, Obj: 0.484060, No Obj: 0.484388, .5R: 1.000000, .75R: 0.500000, count: 2
Region 23 Avg IOU: 0.600125, Class: 0.499300, Obj: 0.443554, No Obj: 0.443441, .5R: 1.000000, .75R: 0.000000, count: 2
Region 16 Avg IOU: 0.639077, Class: 0.500306, Obj: 0.484287, No Obj: 0.484387, .5R: 1.000000, .75R: 0.000000, count: 4
Region 23 Avg IOU: 0.703628, Class: 0.498849, Obj: 0.443648, No Obj: 0.443428, .5R: 1.000000, .75R: 0.500000, count: 2
Region 16 Avg IOU: 0.589364, Class: 0.500461, Obj: 0.484022, No Obj: 0.484387, .5R: 0.750000, .75R: 0.000000, count: 4
Region 23 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.443432, .5R: -nan, .75R: -nan, count: 0
190: 346.569702, 391.753906 avg, 0.000001 rate, 0.283308 seconds, 4560 images
Resizing
416
Loaded: 0.001242 seconds
Region 16 Avg IOU: 0.479336, Class: 0.499920, Obj: 0.484057, No Obj: 0.483996, .5R: 0.333333, .75R: 0.000000, count: 3
Region 23 Avg IOU: 0.461412, Class: 0.498690, Obj: 0.441732, No Obj: 0.442085, .5R: 0.500000, .75R: 0.000000, count: 2
Region 16 Avg IOU: 0.639757, Class: 0.501122, Obj: 0.483671, No Obj: 0.483995, .5R: 0.666667, .75R: 0.000000, count: 3
Region 23 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.442003, .5R: -nan, .75R: -nan, count: 0
Region 16 Avg IOU: 0.550510, Class: 0.499141, Obj: 0.484038, No Obj: 0.483997, .5R: 0.333333, .75R: 0.000000, count: 3
Region 23 Avg IOU: 0.564738, Class: 0.498090, Obj: 0.442024, No Obj: 0.442011, .5R: 1.000000, .75R: 0.000000, count: 1
Region 16 Avg IOU: 0.683012, Class: 0.501956, Obj: 0.483899, No Obj: 0.483995, .5R: 1.000000, .75R: 0.333333, count: 3
Region 23 Avg IOU: 0.570952, Class: 0.495622, Obj: 0.441918, No Obj: 0.442008, .5R: 1.000000, .75R: 0.000000, count: 1
Region 16 Avg IOU: 0.713837, Class: 0.500096, Obj: 0.483738, No Obj: 0.483995, .5R: 1.000000, .75R: 0.666667, count: 3
Region 23 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.442001, .5R: -nan, .75R: -nan, count: 0
Region 16 Avg IOU: 0.695504, Class: 0.500684, Obj: 0.484163, No Obj: 0.483996, .5R: 1.000000, .75R: 0.500000, count: 4
Region 23 Avg IOU: 0.778365, Class: 0.496702, Obj: 0.441843, No Obj: 0.442002, .5R: 1.000000, .75R: 1.000000, count: 2
Region 16 Avg IOU: 0.541589, Class: 0.500593, Obj: 0.483568, No Obj: 0.483996, .5R: 0.750000, .75R: 0.000000, count: 4
Region 23 Avg IOU: 0.2255391, Class: 0.500187, Obj: 0.442183, No Obj: 0.442006, .5R: 0.000000, .75R: 0.000000, count: 2
Region 16 Avg IOU: 0.641378, Class: 0.501509, Obj: 0.483898, No Obj: 0.483994, .5R: 1.000000, .75R: 0.000000, count: 3
Region 23 Avg IOU: 0.732253, Class: 0.496593, Obj: 0.442091, No Obj: 0.442005, .5R: 1.000000, .75R: 0.000000, count: 1
191: 259.400696, 378.518585 avg, 0.000001 rate, 0.243922 seconds, 4584 images
Loaded: 0.000045 seconds
Region 16 Avg IOU: 0.660456, Class: 0.500171, Obj: 0.483456, No Obj: 0.483608, .5R: 1.000000, .75R: 0.333333, count: 6
Region 23 Avg IOU: 0.593410, Class: 0.496879, Obj: 0.440245, No Obj: 0.440594, .5R: 1.000000, .75R: 0.000000, count: 1
Region 16 Avg IOU: 0.579343, Class: 0.498591, Obj: 0.484065, No Obj: 0.483607, .5R: 1.000000, .75R: 0.000000, count: 4
Region 23 Avg IOU: 0.595838, Class: 0.498654, Obj: 0.440775, No Obj: 0.440590, .5R: 0.857143, .75R: 0.142857, count: 7
Region 16 Avg IOU: 0.598298, Class: 0.499262, Obj: 0.483344, No Obj: 0.483607, .5R: 1.000000, .75R: 0.000000, count: 2
Region 23 Avg IOU: 0.492555, Class: 0.499408, Obj: 0.441513, No Obj: 0.440589, .5R: 0.000000, .75R: 0.000000, count: 1
```

Figure 16. First 200 iterations.

```

jacksonliu@jacksonliu-GL702VI: ~/darknet
60086: 0.848015, 0.917569 avg, 0.001000 rate, 0.299766 seconds, 1442064 images
Loaded: 0.00044 seconds
Region 16 Avg IOU: 0.866586, Class: 0.996115, Obj: 0.413956, No Obj: 0.002799, .5R: 1.000000, .75R: 1.000000, count: 3
Region 23 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000004, .5R: -nan, .75R: -nan, count: 0
Region 16 Avg IOU: 0.766979, Class: 0.699246, Obj: 0.490837, No Obj: 0.002007, .5R: 1.000000, .75R: 0.333333, count: 3
Region 23 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000049, .5R: -nan, .75R: -nan, count: 0
Region 16 Avg IOU: 0.770600, Class: 0.993510, Obj: 0.270597, No Obj: 0.003938, .5R: 1.000000, .75R: 0.400000, count: 5
Region 23 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000005, .5R: -nan, .75R: -nan, count: 0
Region 16 Avg IOU: 0.797925, Class: 0.926345, Obj: 0.180344, No Obj: 0.002383, .5R: 1.000000, .75R: 1.000000, count: 3
Region 23 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000009, .5R: -nan, .75R: -nan, count: 0
Region 16 Avg IOU: 0.863707, Class: 0.999331, Obj: 0.351726, No Obj: 0.003413, .5R: 1.000000, .75R: 1.000000, count: 3
Region 23 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000002, .5R: -nan, .75R: -nan, count: 0
Region 16 Avg IOU: 0.778340, Class: 0.705243, Obj: 0.241827, No Obj: 0.002324, .5R: 1.000000, .75R: 0.500000, count: 4
Region 23 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000004, .5R: -nan, .75R: -nan, count: 0
Region 16 Avg IOU: 0.762028, Class: 0.623174, Obj: 0.078610, No Obj: 0.001652, .5R: 1.000000, .75R: 0.666667, count: 3
Region 23 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000005, .5R: -nan, .75R: -nan, count: 0
Region 16 Avg IOU: 0.701622, Class: 0.831950, Obj: 0.134560, No Obj: 0.002550, .5R: 1.000000, .75R: 0.400000, count: 5
Region 23 Avg IOU: 0.751892, Class: 0.999730, Obj: 0.021161, No Obj: 0.000101, .5R: 1.000000, .75R: 1.000000, count: 1
60087: 0.816556, 0.907467 avg, 0.001000 rate, 0.297388 seconds, 1442088 images
Loaded: 0.000447 seconds
Region 16 Avg IOU: 0.754550, Class: 0.997432, Obj: 0.606525, No Obj: 0.002316, .5R: 1.000000, .75R: 0.666667, count: 3
Region 23 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000062, .5R: -nan, .75R: -nan, count: 0
Region 16 Avg IOU: 0.702984, Class: 0.966959, Obj: 0.104345, No Obj: 0.001610, .5R: 1.000000, .75R: 0.333333, count: 3
Region 23 Avg IOU: 0.754739, Class: 0.791228, Obj: 0.025784, No Obj: 0.000049, .5R: 1.000000, .75R: 1.000000, count: 1
Region 16 Avg IOU: 0.723274, Class: 0.867377, Obj: 0.117610, No Obj: 0.001643, .5R: 1.000000, .75R: 0.500000, count: 4
Region 23 Avg IOU: 0.579376, Class: 0.820346, Obj: 0.062316, No Obj: 0.000139, .5R: 0.333333, .75R: 0.000000, count: 3
Region 16 Avg IOU: 0.779586, Class: 0.993921, Obj: 0.218420, No Obj: 0.002952, .5R: 1.000000, .75R: 0.500000, count: 4
Region 23 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000004, .5R: -nan, .75R: -nan, count: 0
Region 16 Avg IOU: 0.850583, Class: 0.971330, Obj: 0.246421, No Obj: 0.001386, .5R: 1.000000, .75R: 1.000000, count: 2
Region 23 Avg IOU: 0.653112, Class: 0.996537, Obj: 0.274175, No Obj: 0.000176, .5R: 1.000000, .75R: 0.000000, count: 2
Region 16 Avg IOU: 0.862736, Class: 0.999370, Obj: 0.420003, No Obj: 0.002494, .5R: 1.000000, .75R: 1.000000, count: 4
Region 23 Avg IOU: 0.894468, Class: 0.999999, Obj: 0.735898, No Obj: 0.000189, .5R: 1.000000, .75R: 1.000000, count: 1
Region 16 Avg IOU: 0.748087, Class: 0.986417, Obj: 0.235765, No Obj: 0.001319, .5R: 1.000000, .75R: 0.666667, count: 3
Region 23 Avg IOU: 0.721922, Class: 0.953849, Obj: 0.000202, No Obj: 0.000017, .5R: 1.000000, .75R: 0.000000, count: 1
Region 16 Avg IOU: 0.793980, Class: 0.997904, Obj: 0.490087, No Obj: 0.002137, .5R: 1.000000, .75R: 1.000000, count: 3
Region 23 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000004, .5R: -nan, .75R: -nan, count: 0
60088: 0.963319, 0.913053 avg, 0.001000 rate, 0.303621 seconds, 1442112 images
Loaded: 0.000446 seconds
Region 16 Avg IOU: 0.758636, Class: 0.993933, Obj: 0.448952, No Obj: 0.003564, .5R: 1.000000, .75R: 0.500000, count: 4
Region 23 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000008, .5R: -nan, .75R: -nan, count: 0
Region 16 Avg IOU: 0.822085, Class: 0.823947, Obj: 0.182590, No Obj: 0.002103, .5R: 1.000000, .75R: 0.800000, count: 5
Region 23 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000005, .5R: -nan, .75R: -nan, count: 0
Region 16 Avg IOU: 0.816583, Class: 0.996314, Obj: 0.659062, No Obj: 0.002335, .5R: 1.000000, .75R: 0.666667, count: 3
Region 23 Avg IOU: 0.701408, Class: 0.997363, Obj: 0.113847, No Obj: 0.000130, .5R: 1.000000, .75R: 0.000000, count: 1
Region 16 Avg IOU: 0.785958, Class: 0.923948, Obj: 0.309430, No Obj: 0.003651, .5R: 1.000000, .75R: 0.800000, count: 5
Region 23 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000048, .5R: -nan, .75R: -nan, count: 0
Region 16 Avg IOU: 0.855148, Class: 0.770859, Obj: 0.219884, No Obj: 0.001511, .5R: 1.000000, .75R: 1.000000, count: 3

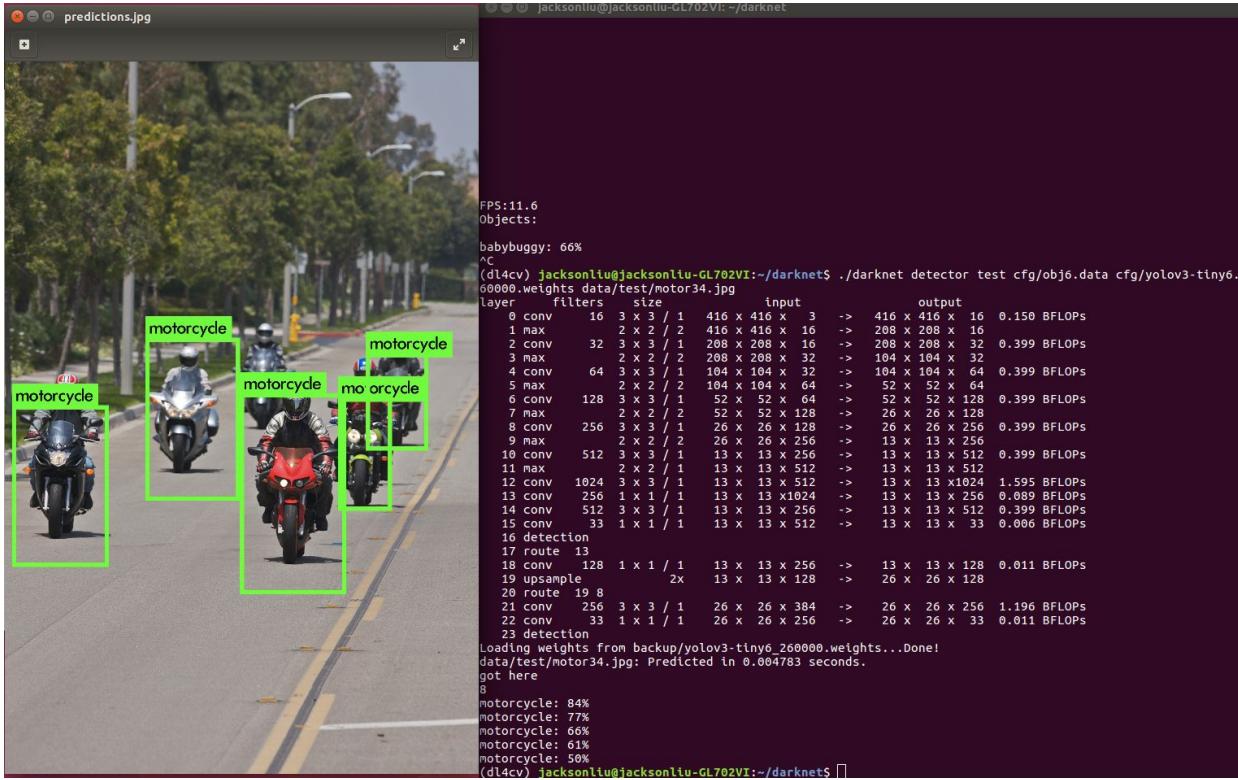
```

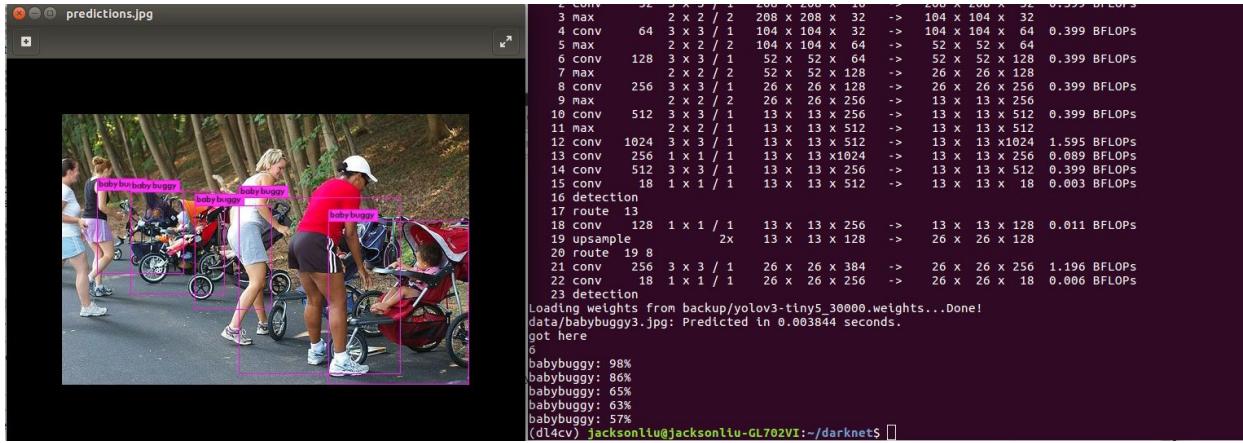
Figure 17. Training after 60000 iterations.

## Results

We were able to implement our network; when run on a laptop with a GPU (see the Equipment section above) the network was able to detect objects on live video up to 120 fps with a fair amount of accuracy.

Demo video: <https://www.youtube.com/watch?v=hNNhUPTL0bg&feature=youtu.be>





## Unchecked Ambition (Things That Didn't Work)

### Integration with Original YOLOv3/Tiny YOLO network

Joseph Redmon (pjreddie) has a github project which contains a pre-trained version of Tiny YOLO which contains many more classes than our AVD. It was originally our hope that we would be able to combine the AVD with the original Tiny YOLOv3 in order to get a network which would accurately detect an inclusive set of objects. After some research we realized that would involve much more time and/or expertise than we had - we would have needed to retrain the network on all the classes, or modified the weight matrix to directly insert the AVD training weights into the original Tiny YOLO weight matrix -- so the idea was descoped out of the project.

### Raspberry Pi

Our goal with the Raspberry Pi was to fully implement our AVD on the Pi as a stand alone implementation. We got as far as attempting to run YOLO and AVD when we hit a snag; the Pi did not have enough memory to run the networks, and would consistently give segfault errors after loading weights. We believe this is due to the 4-core architecture of the Raspberry Pi running Raspbian. It utilizes only 25% of its resources when running the detector.

It is possible we could overcome this by: 1) using a different OS, 2) using a larger SD card, or 3) using an even smaller/shallow architecture than the provided Tiny YOLO. However, we did not explore these options due to time and priority constraints. Implementing in the Raspberry Pi was always somewhat of a stretch goal; we judged that our time was better spent improving the AVD as implemented on a laptop.

## Future Work

While our model performed fairly well, it does not yet match the performance of the supplied, pre-trained model of YOLO v3 or Tiny YOLO. Our model could be improved for detection of obscured/partial objects as well as objects at unusual angles. This could likely be achieved with expanding and improving our datasets.

In addition, implementation on an edge device such as the Raspberry Pi is still an interesting project. We believe it is possible; we just could not complete it due to time constraints.

## References

- [1] Joseph Redmon, Ali Farhadi. YOLOv3: An Incremental Improvement, University of Washington
- [2] 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.
- [3] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollar, and C. L. Zitnick. Microsoft coco: Common objects in context. In European conference on computer vision, pages 740–755. Springer, 2014.



