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

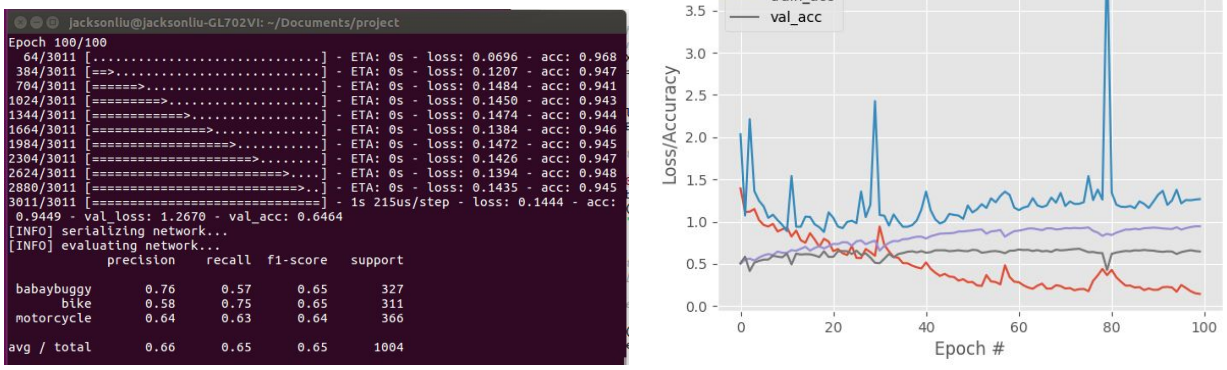


Figure 1. MiniVgg training results.

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

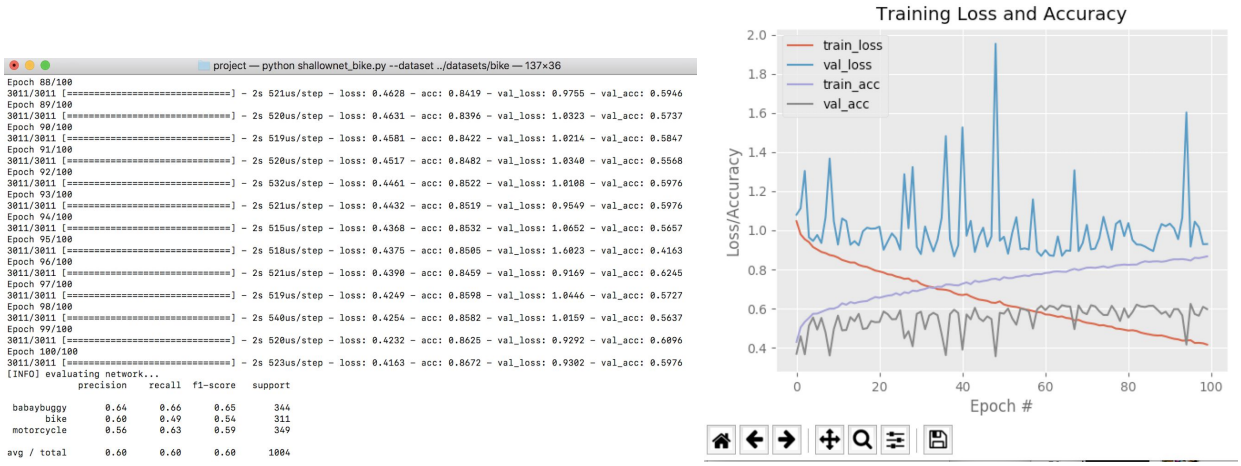


Figure 2. ShallowNet training results.

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

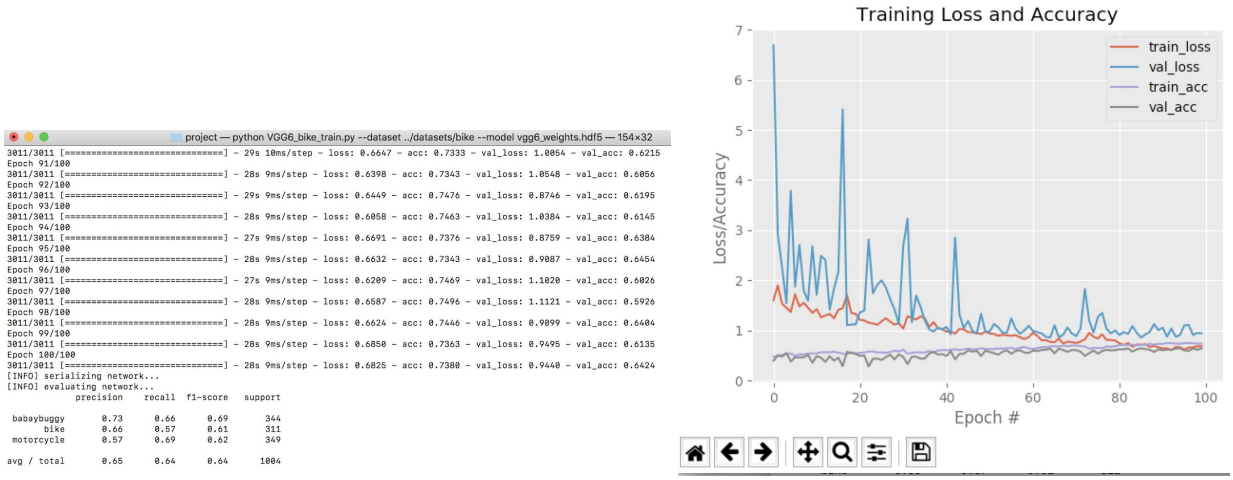


Figure 3. Vgg6 training results with learning rate of 0.01.

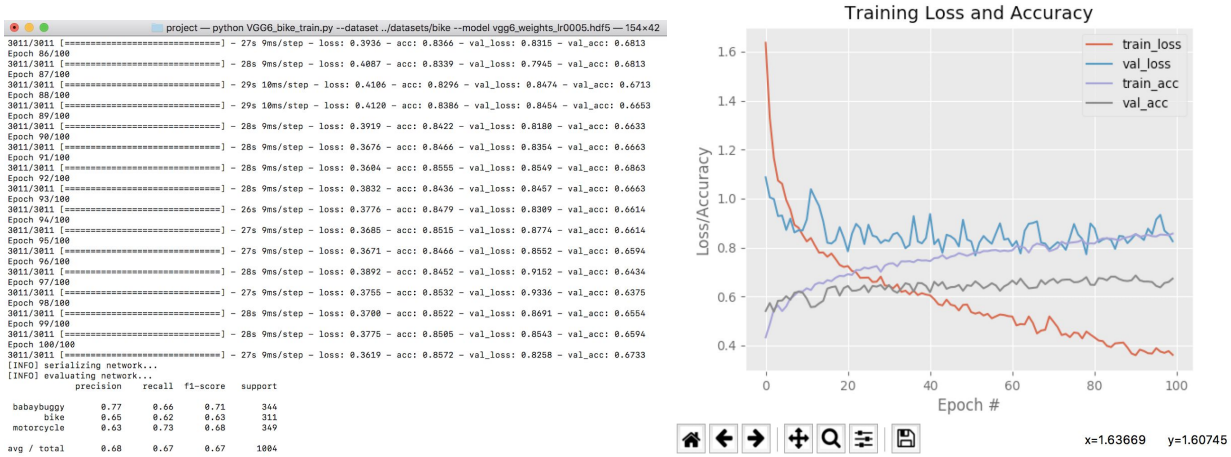


Figure 4. Vgg6 training results with learning rate 0.001.

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

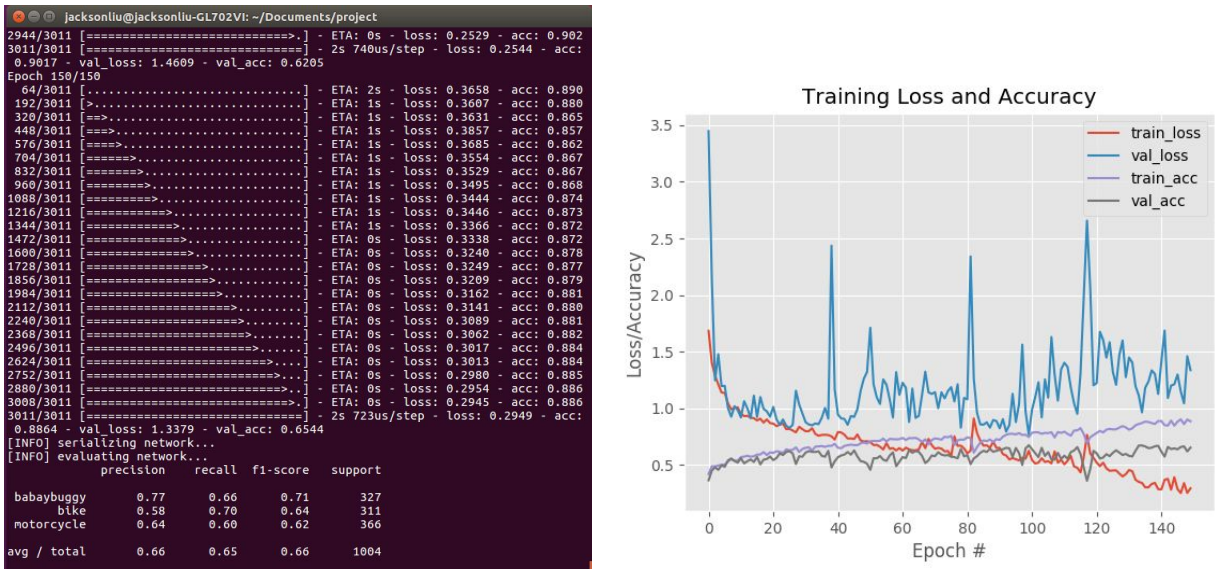


Figure 5. Vgg8 training results with learning rate 0.01.

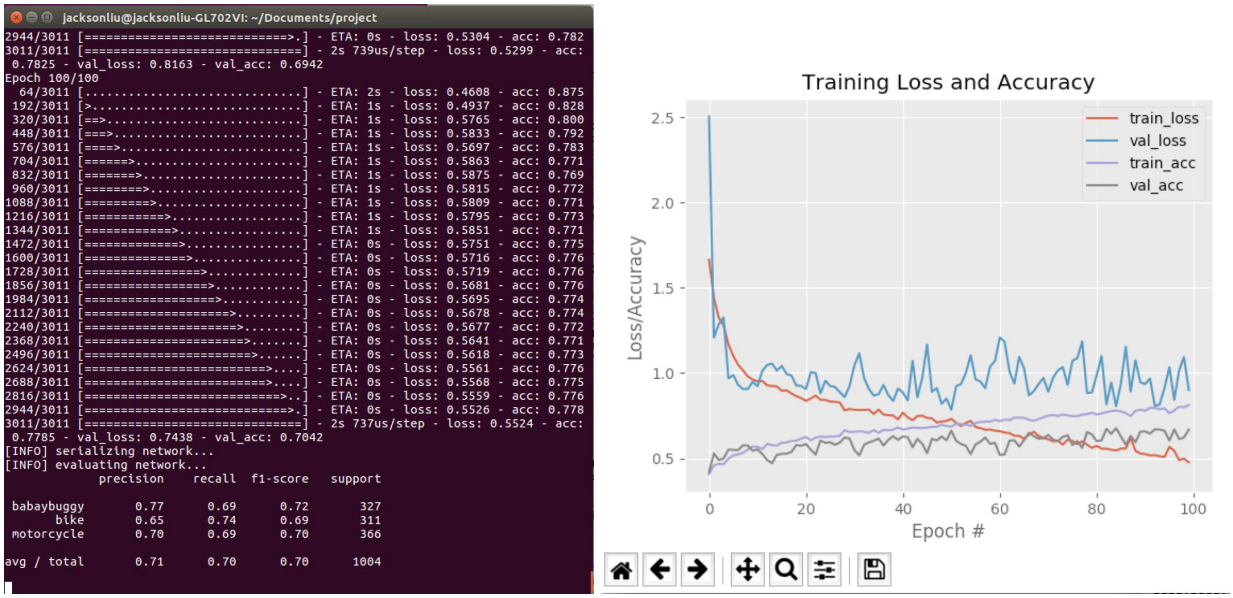


Figure 6. Vgg8 training results with learning rate 0.001.

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

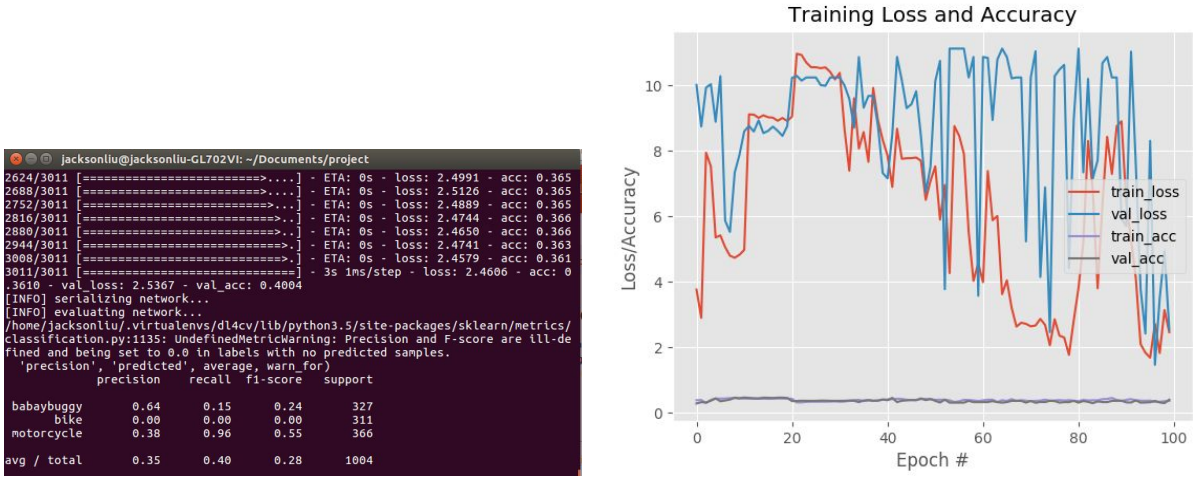


Figure 7. Vgg19 training results with learning rate 0.01.

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).

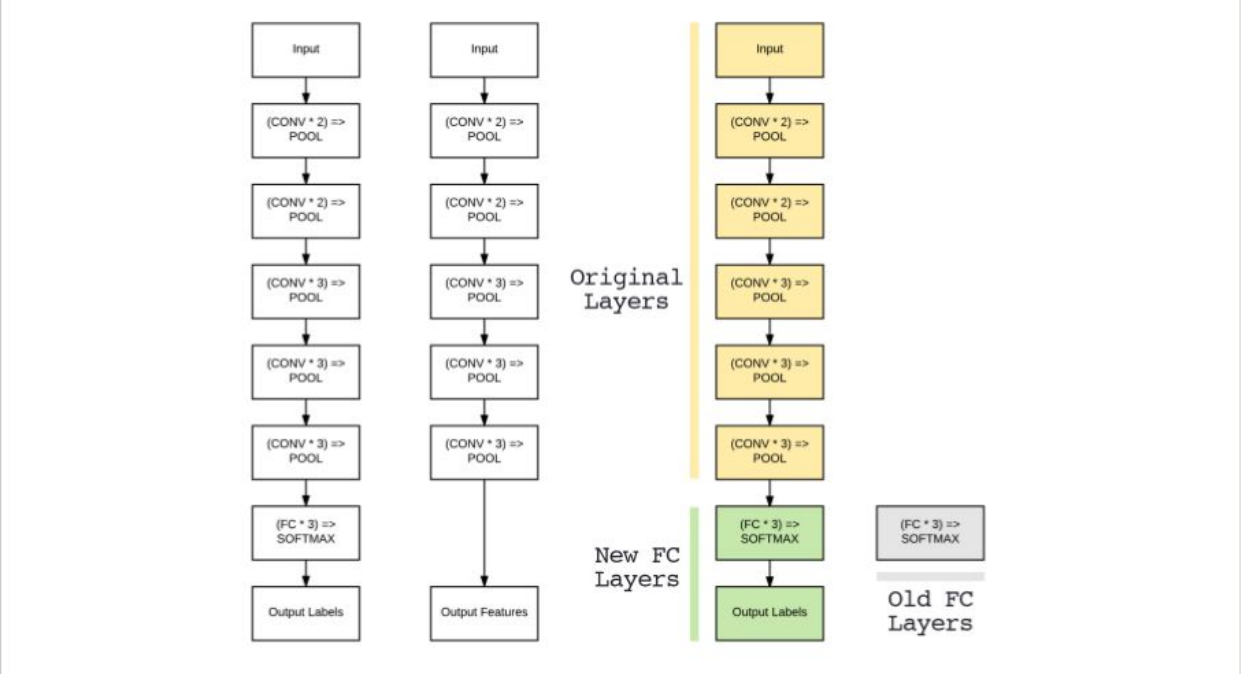


Figure 8. Fine tuning transfer learning method.

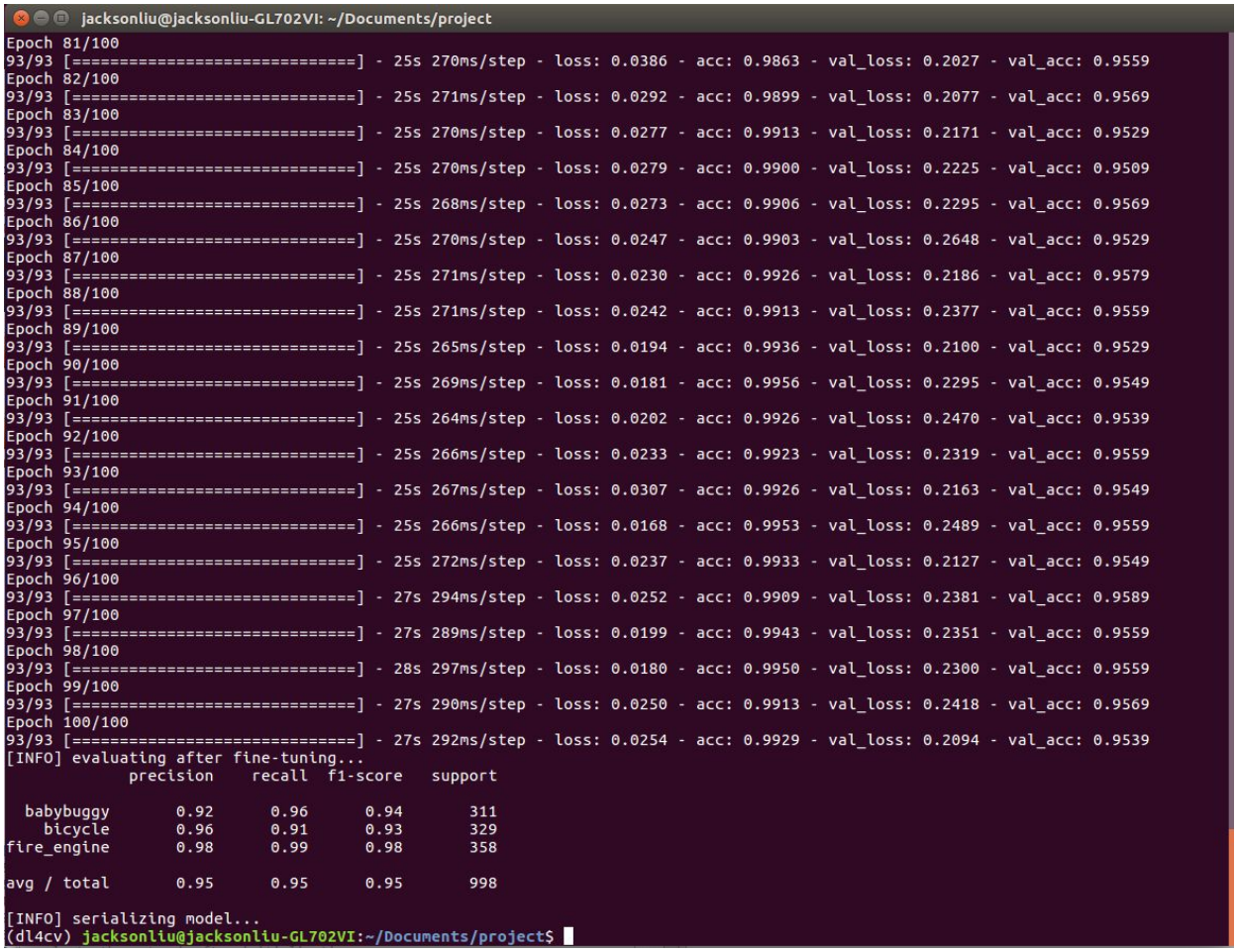


Figure 9. Fine Tuning result.

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.

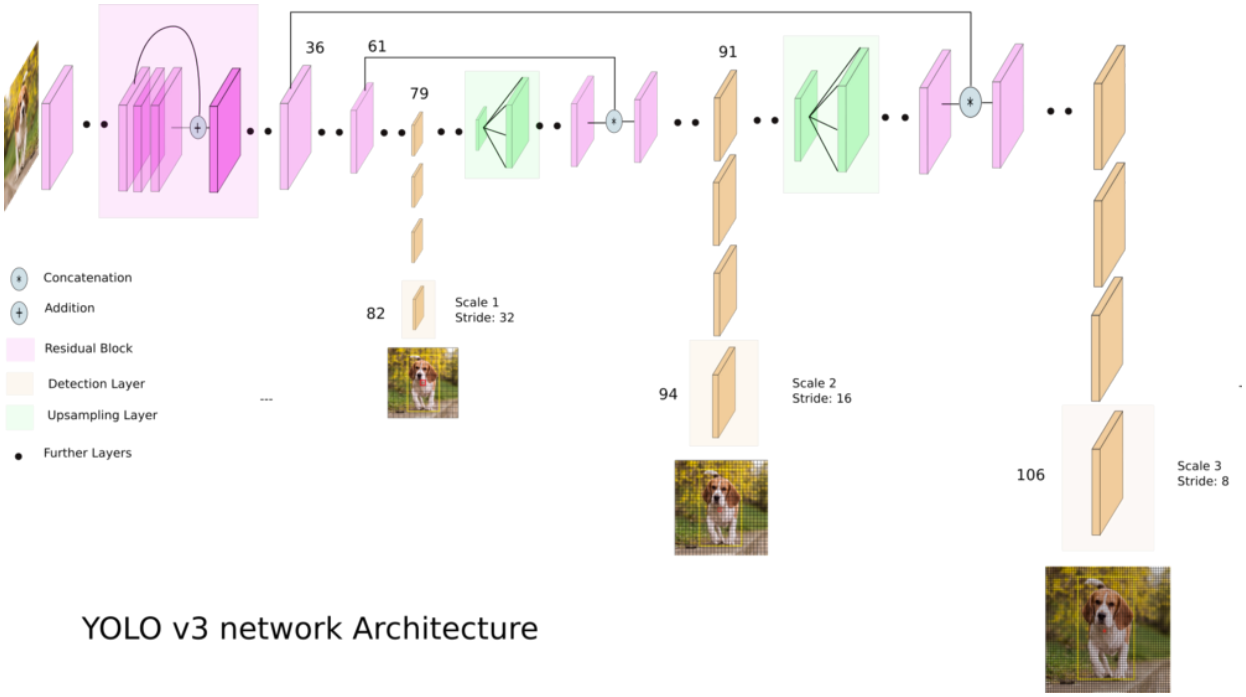


Figure 10: YOLO v3 network architecture overview. (Source: https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b)

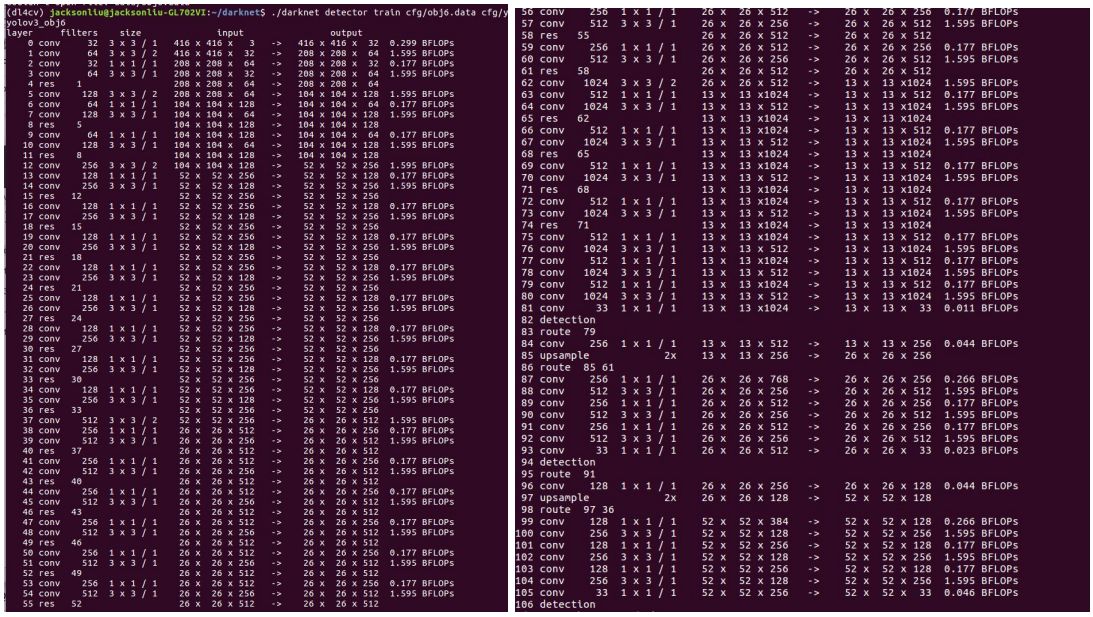


Figure 11. YOLO v3 network architecture as displayed when run on the command console.

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:
(10×14), (23×27), (37×58), (81×82), (135×169), (344×319)

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 x 3 / 1   416 x 416 x   3   ->   416 x 416 x  16  0.150 BFLOPs
    1 max          2 x 2 / 2   416 x 416 x  16   ->   208 x 208 x  16
    2 conv     32  3 x 3 / 1   208 x 208 x  16   ->   208 x 208 x  32  0.399 BFLOPs
    3 max          2 x 2 / 2   208 x 208 x  32   ->   104 x 104 x  32
    4 conv     64  3 x 3 / 1   104 x 104 x  32   ->   104 x 104 x  64  0.399 BFLOPs
    5 max          2 x 2 / 2   104 x 104 x  64   ->    52 x  52 x  64
    6 conv    128  3 x 3 / 1    52 x  52 x  64   ->    52 x  52 x 128  0.399 BFLOPs
    7 max          2 x 2 / 2    52 x  52 x 128   ->    26 x  26 x 128
    8 conv    256  3 x 3 / 1    26 x  26 x 128   ->    26 x  26 x 256  0.399 BFLOPs
    9 max          2 x 2 / 2    26 x  26 x 256   ->    13 x  13 x 256
   10 conv    512  3 x 3 / 1    13 x  13 x 256   ->    13 x  13 x 512  0.399 BFLOPs
   11 max          2 x 2 / 1    13 x  13 x 512   ->    13 x  13 x 512
   12 conv   1024  3 x 3 / 1    13 x  13 x 512   ->    13 x  13 x1024  1.595 BFLOPs
   13 conv    256  1 x 1 / 1    13 x  13 x1024   ->    13 x  13 x 256  0.089 BFLOPs
   14 conv    512  3 x 3 / 1    13 x  13 x 256   ->    13 x  13 x 512  0.399 BFLOPs
   15 conv     33  1 x 1 / 1    13 x  13 x 512   ->    13 x  13 x  33  0.006 BFLOPs
   16 detection
   17 route  13
   18 conv    128  1 x 1 / 1    13 x  13 x 256   ->    13 x  13 x 128  0.011 BFLOPs
   19 upsample           2x    13 x  13 x 128   ->    26 x  26 x 128
   20 route  19 8
   21 conv    256  3 x 3 / 1    26 x  26 x 384   ->    26 x  26 x 256  1.196 BFLOPs
   22 conv     33  1 x 1 / 1    26 x  26 x 256   ->    26 x  26 x  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 . 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.

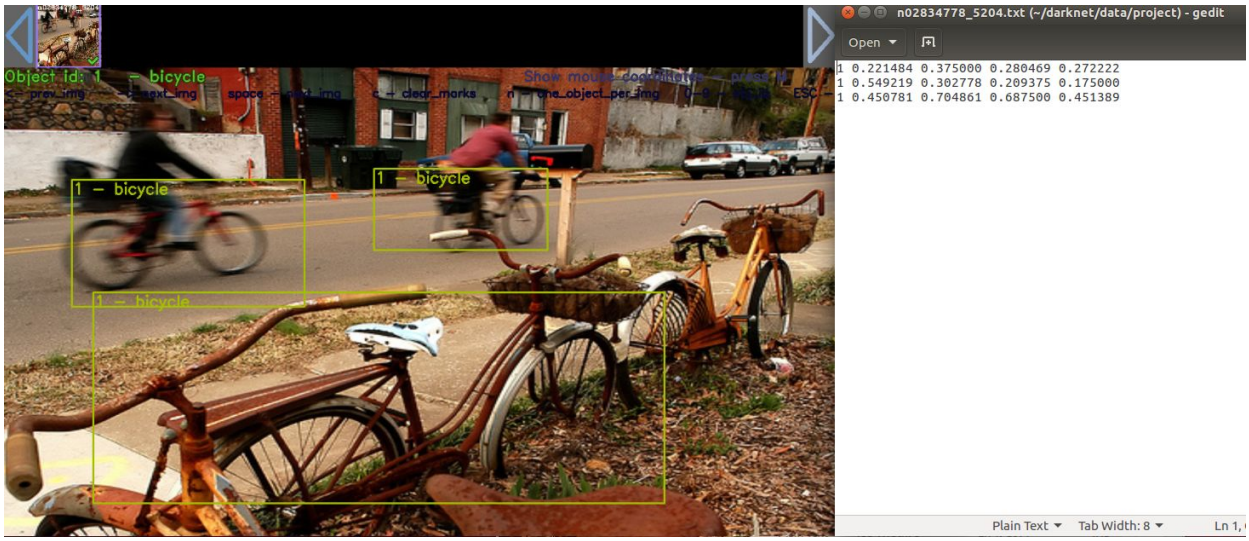


Figure 13. Yolo-mark UI and output.

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.

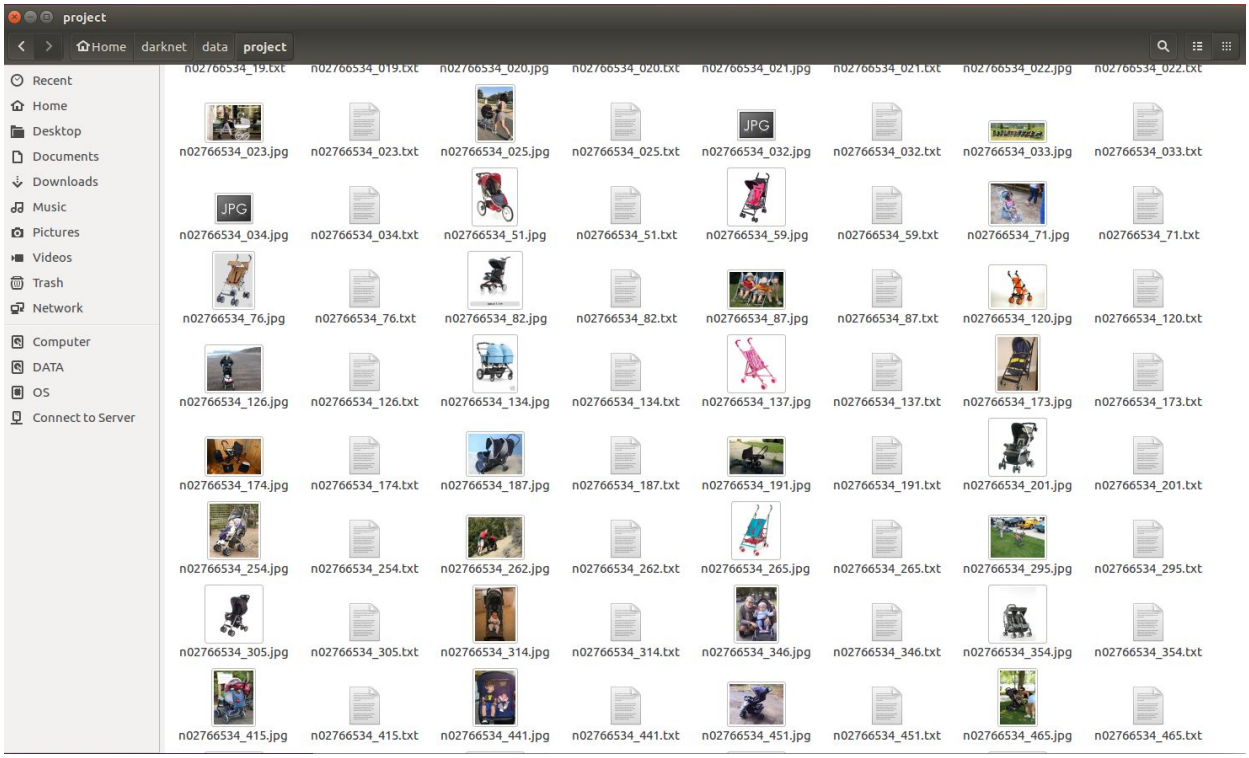


Figure 15. Data annotation file

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%.

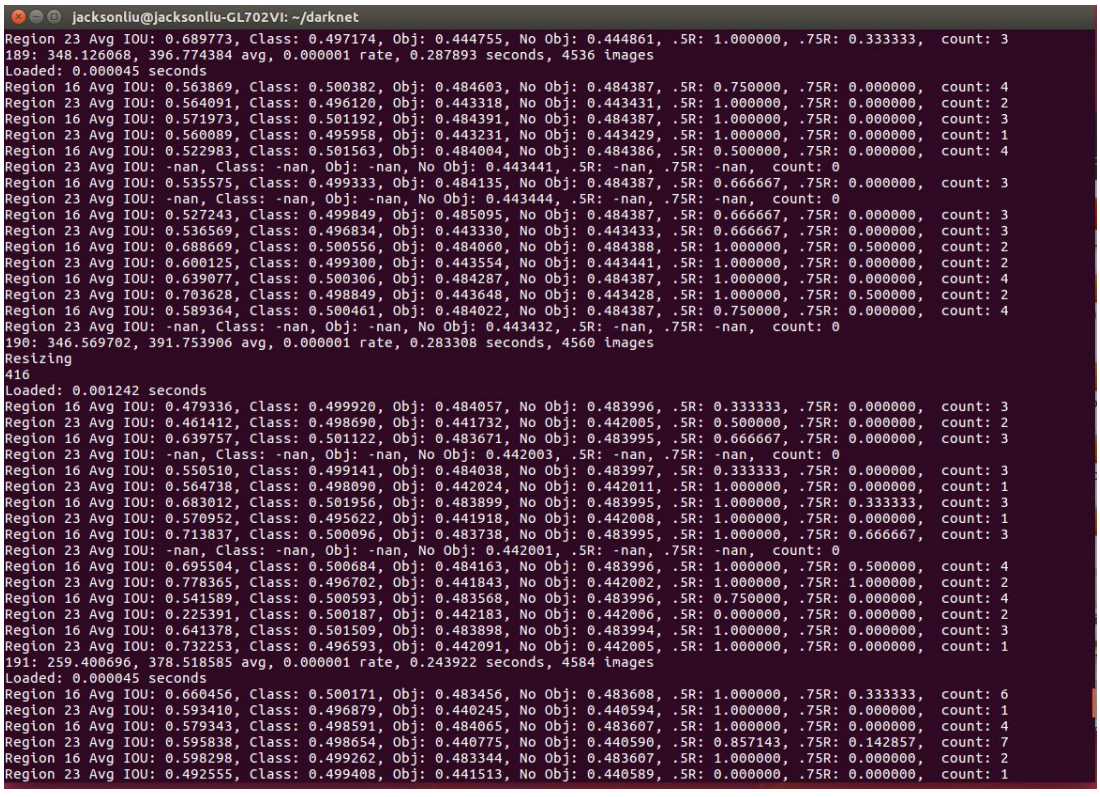


Figure 16. First 200 iterations.

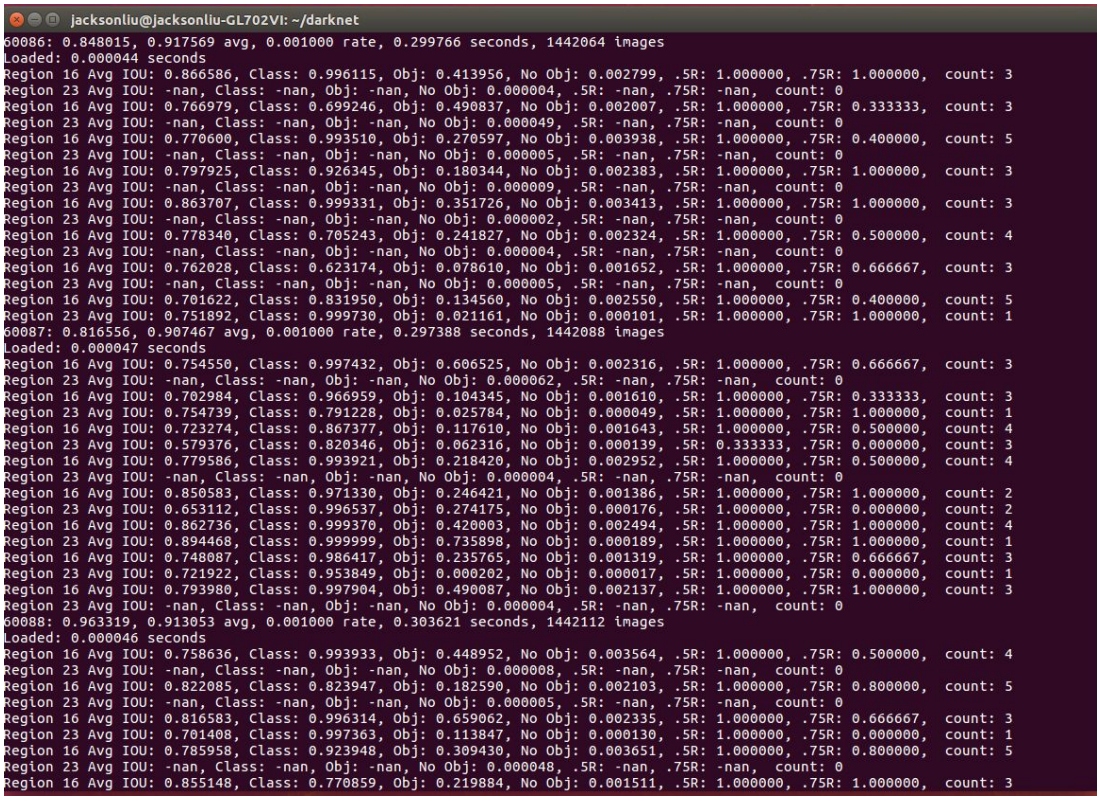
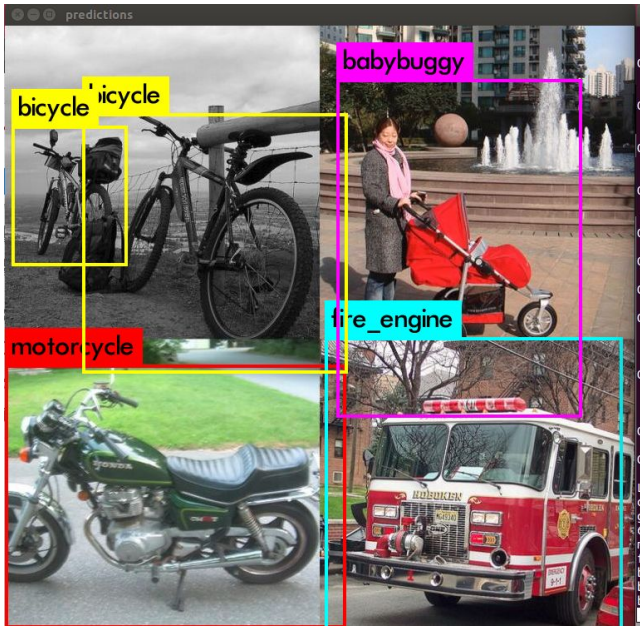


Figure 17. Training after 60000 iterations.

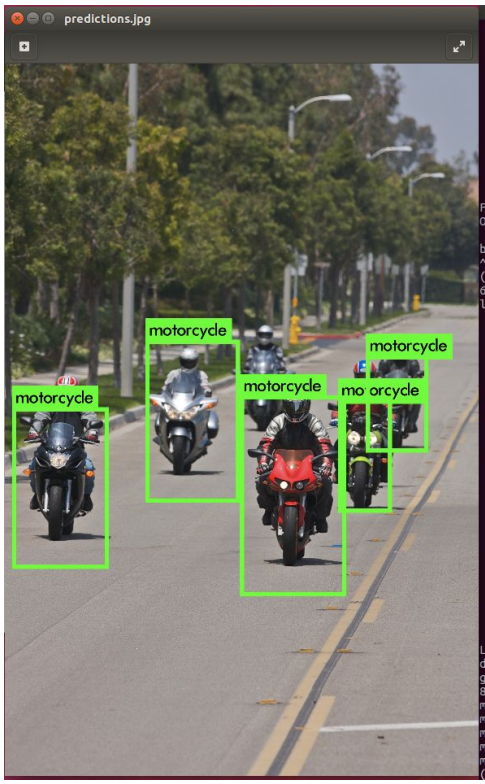
Results

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

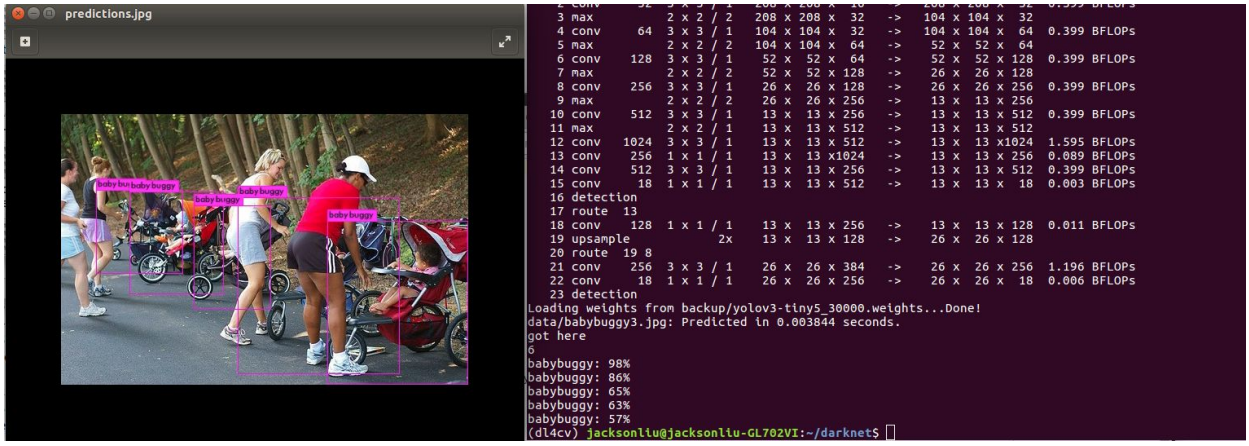



```
jacksonliu@jacksonliu-GL702VI: ~/darknet
   3 max          2 x 2 / 2   208 x 208 x  32   ->   104 x 104 x  32
   4 conv     64  3 x 3 / 1   104 x 104 x  32   ->   104 x 104 x  64  0.399 BFL
OPs
   5 max          2 x 2 / 2   104 x 104 x  64   ->    52 x  52 x  64
   6 conv    128  3 x 3 / 1    52 x  52 x  64   ->    52 x  52 x 128  0.399 BFL
OPs
   7 max          2 x 2 / 2    52 x  52 x 128   ->    26 x  26 x 128
   8 conv    256  3 x 3 / 1    26 x  26 x 128   ->    26 x  26 x 256  0.399 BFL
OPs
   9 max          2 x 2 / 2    26 x  26 x 256   ->    13 x  13 x 256
  10 conv    512  3 x 3 / 1    13 x  13 x 256   ->    13 x  13 x 512  0.399 BFL
OPs
  11 max          2 x 2 / 1    13 x  13 x 512   ->    13 x  13 x 512
  12 conv   1024  3 x 3 / 1    13 x  13 x 512   ->    13 x  13 x1024  1.595 BFL
OPs
  13 conv    256  1 x 1 / 1    13 x  13 x1024   ->    13 x  13 x 256  0.089 BFL
OPs
  14 conv    512  3 x 3 / 1    13 x  13 x 256   ->    13 x  13 x 512  0.399 BFL
OPs
  15 conv     30  1 x 1 / 1    13 x  13 x 512   ->    13 x  13 x  30  0.005 BFL
OPs
  16 detection
  17 route  13
  18 conv    128  1 x 1 / 1    13 x  13 x 256   ->    13 x  13 x 128  0.011 BFL
OPs
  19 upsample            2x    13 x  13 x 128   ->    26 x  26 x 128
  20 route  19 8
  21 conv    256  3 x 3 / 1    26 x  26 x 384   ->    26 x  26 x 256  1.196 BFL
OPs
  22 conv     30  1 x 1 / 1    26 x  26 x 256   ->    26 x  26 x  30  0.010 BFL
OPs
  23 detection
Loading weights from yolov3-tiny5_100000.weights...Done!
data/combo2.jpg: Predicted in 0.003945 seconds.
got here
6
motorcycle: 83%
fire_engine: 71%
bicycle: 81%
bicycle: 54%
babybuggy: 48%
```




```
FPS:11.6
Objects:

babybuggy: 66%
^C
(dl4cv) jacksonliu@jacksonliu-GL702VI:~/darknet$ ./darknet detector test cfg/obj6.data cfg/yolov3-tiny6.c
60000.weights data/test/motor34.jpg
layer     filters    size              input                output
    0 conv     16  3 x 3 / 1   416 x 416 x   3   ->   416 x 416 x  16  0.150 BFLOPs
    1 max          2 x 2 / 2   416 x 416 x  16   ->   208 x 208 x  16
    2 conv     32  3 x 3 / 1   208 x 208 x  16   ->   208 x 208 x  32  0.399 BFLOPs
    3 max          2 x 2 / 2   208 x 208 x  32   ->   104 x 104 x  32
    4 conv     64  3 x 3 / 1   104 x 104 x  32   ->   104 x 104 x  64  0.399 BFLOPs
    5 max          2 x 2 / 2   104 x 104 x  64   ->    52 x  52 x  64
    6 conv    128  3 x 3 / 1    52 x  52 x  64   ->    52 x  52 x 128  0.399 BFLOPs
    7 max          2 x 2 / 2    52 x  52 x 128   ->    26 x  26 x 128
    8 conv    256  3 x 3 / 1    26 x  26 x 128   ->    26 x  26 x 256  0.399 BFLOPs
    9 max          2 x 2 / 2    26 x  26 x 256   ->    13 x  13 x 256
   10 conv    512  3 x 3 / 1    13 x  13 x 256   ->    13 x  13 x 512  0.399 BFLOPs
   11 max          2 x 2 / 1    13 x  13 x 512   ->    13 x  13 x 512
   12 conv   1024  3 x 3 / 1    13 x  13 x 512   ->    13 x  13 x1024  1.595 BFLOPs
   13 conv    256  1 x 1 / 1    13 x  13 x1024   ->    13 x  13 x 256  0.089 BFLOPs
   14 conv    512  3 x 3 / 1    13 x  13 x 256   ->    13 x  13 x 512  0.399 BFLOPs
   15 conv     33  1 x 1 / 1    13 x  13 x 512   ->    13 x  13 x  33  0.006 BFLOPs
   16 detection
   17 route  13
   18 conv    128  1 x 1 / 1    13 x  13 x 256   ->    13 x  13 x 128  0.011 BFLOPs
   19 upsample            2x    13 x  13 x 128   ->    26 x  26 x 128
   20 route  19 8
   21 conv    256  3 x 3 / 1    26 x  26 x 384   ->    26 x  26 x 256  1.196 BFLOPs
   22 conv     33  1 x 1 / 1    26 x  26 x 256   ->    26 x  26 x  33  0.011 BFLOPs
   23 detection
Loading weights from backup/yolov3-tiny6_260000.weights...Done!
data/test/motor34.jpg: Predicted in 0.004783 seconds.
got here
8
motorcycle: 84%
motorcycle: 77%
motorcycle: 66%
motorcycle: 61%
motorcycle: 50%
(dl4cv) jacksonliu@jacksonliu-GL702VI:~/darknet$
```



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/shallower 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.