5-1-2018

DETERMINING HARDWARE SETUP FOR TRAINING AND TESTING AN OBJECT DETECTION MODEL FOR USE IN AN INDUSTRIAL SETTING

Tanmay Sardesai

DETERMINING HARDWARE SETUP FOR TRAINING AND TESTING AN OBJECT DETECTION MODEL FOR USE IN AN INDUSTRIAL SETTING

by

TANMAY SARDESAI

Presented to the Faculty of the Honors College of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

HONORS BACHELOR OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2018

ACKNOWLEDGMENTS

I would like to thank Dr. Christopher D. McMurrough for his mentorship on this project. I would also like to thank my colleagues—Ashley Quimod, Gowsi Sriskandarajah, Ranjeev Neurekar and Sneha Kalita—for their wonderful collaboration in helping complete the training phase of this project.

This project would have been impossible without the financial support of Cloud 9 Perception. They provided us with the right tools as well as the expensive hardware required.

Lastly I would like to thank my family for all the financial and emotional support that they have provided to help me successfully reach this stage in my collegiate career.

May 04, 2014

ABSTRACT

DETERMINING HARDWARE SETUP FOR TRAINING AND TESTING AN OBJECT DETECTION MODEL FOR USE IN AN INDUSTRIAL SETTING

Tanmay Sardesai, B.S. Computer Science

The University of Texas at Arlington, 2018

Faculty Mentor: Christopher McMurrough

Darknet YOLO is a real-time object detection system that is used in this project. YOLO, which stands for You Only Look Once, uses a single convolutional neural network to detect objects in an image. There are various versions of YOLO, all with their own advantages and disadvantages. For the purpose of this project we will be using Tiny YOLOv2, which is a version of YOLO that is lightweight and performs less calculation, giving us a lower accuracy but higher frame rate. Tiny YOLOv2 applies a single neural network to the full image and divides the image into regions and predicts bounding boxes and probabilities for each region. As part of the Senior Design project our team has trained a seven-class object detection model. The classes that can be detected by this model are people, forklifts, trucks, boxes, pallets, pallet jacks and industrial carts. The goal of this Honors thesis was to run this object detection model on different hardware systems to decide the best option. This model is tested on five different systems: Raspberry Pi 3, Asus Intel i5 4th Gen CPU, Nvidia Jetson TX1, Nvidia Jetson TX2 and Nvidia GeForce GTX 980 Ti. After taking into account the frame rate, accuracy and cost of each of these systems, our recommendation is to use Nvidia Jetson TX2.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Introducing YOLO

Object detection should be fast, accurate and able to recognize a wide variety of classes. Over the past few years object detection has become faster and more accurate using neural networks. We humans glance at an image and can easily see what objects are in the image. Fast and accurate algorithms for object detection will allow computer systems to do the same and make our lives easier. These algorithms can be used to drive cars, automate warehouses and delivery, increase airport security and unlock the potential for general purpose, ubiquitous and responsive robotics system.

YOLO, an abbreviation for You Only Look Once, was created by Joseph Redmon, Ph.D. student at University of Washington working with Dr. Ali Farhadi, who is a Professor at University of Washington. He is one of the 39 students across North America, Europe, and the Middle East to be selected as a 2018 Google Ph.D. Fellow. The paper describing YOLO, which Redmon and Farhadi co-authored with former University of Washington postdoc Santosh Divvala, now a research scientist at the University of Washington Allen School for Artificial Intelligence, and Facebook researcher Ross Girshick, earned the OpenCV People's Choice Award at the Conference on Computer Vision and Pattern Recognition (CVPR 2016). Redmon and Farhadi followed that up with an Honorable Mention at CVPR 2017 for YOLO9000, a new version capable of identifying more than 9,000 different object categories in real time. This is the version of YOLO that is used in the project. YOLO9000 is trained on YOLOv2 to detect 9000. Last month the group released YOLOv3, a newer version of the software. YOLOv3 is much more accurate than a TinyYOLOv2, but when one compares frame rate, it is evident that Tiny YOLO is the correct choice for this project.

1.2 Project Requirements

The sponsor had a list of requirements that we had to satisfy for the completion of this project. First and foremost the system had to be real-time. This means that the program is run on a video input from the webcam and should detect objects as things in the video feed are moving around. Secondly, the model has to detect seven different classes. When the requirements were set, the difficulty of the project was unclear, so each class had different priority. People and forklifts had the highest priority, followed by boxes, pallets, pallet jacks and industrial carts. Trucks had the lowest priority. The requirement for frame rate was minimum 10 frames per second. A standard video input is 30 frames per seconds. Therefore, with a minimum frame rate of 10 frames, one would drop one third of the frames. Lastly, we had to find the cheapest way to satisfy these requirements so that it is easy and cheap to deploy the product at the consumer location.

Detecting industrial objects in real time is the first step towards automating warehouses. With automation companies can increase the output that they produce, reduce the number of accidents and the cost of manufacturing. The most common objects in a warehouse are people and forklifts which is why they were high priority. Object detection for warehouses is currently not done by any company. It is a largely unexplored industry. Therefore, working on this will help the sponsors in their computer vision and machine learning applications in warehouses.

CHAPTER 2

CREATING A DATASET

The first step to any machine learning application is creating a dataset of annotated images. For this project as we wanted to detect objects in an image, the dataset was composed of images of all the classes. This section discusses how to download the images and problems that were tackled during this step.

2.1 Downloading Images

As the training phase required more than 500 images for each of the classes, this task was divided among all the team members. Everyone on the team used a variety of tools to download images. One such open source tool was https://github.com/hardikvasa/google-images-download. Google_images_download can be installed directly via Python-pip. It is an easy way to download bulk images directly from Google images. To download 500 images of forklift, one has to simply type *googleimagesdownload --keywords "forklift" --limit 500* in the terminal.

2.2 Refining Images

2.2.1 Deleting Impurities

As Google image search is not perfect, it always leads to a plethora of images which are completely unrelated. As we are download 500 images, towards the end some of the images are unrelated to the keyword. For example, if one types “pallet” in a Google image search then they will come across images of furniture (made using pallets). Such images cannot be used during the training phase. Therefore, it is necessary to go through all the images cannot be used during the training phase. Therefore, it is necessary to go through all the images and delete such impurities.

2.2.2 Deleting Duplicates

Another issue that Google images cause is that it has multiple duplicates in the same search list. This leads to us downloading multiple copies of the same image. This is bad for training a model as it cannot learn from multiple copies of the same object. In fact, it also causes overfitting. This is why it is required to go through all the images and delete duplicates.

2.3 Re-Download and Repeat

Following the steps described in section 2.2 will lead to a decrease in the number of images gathered, as we are deleting unwanted images. This is why the above two steps have to be repeated until the total number of unique good quality images are above 500. Once we have 500 or more images we can proceed to the next step of annotating images.

CHAPTER 3

ANNOTATION

Most machine learning algorithms are split into two group—supervised learning and unsupervised learning. Supervised learning is when the model is given a data point and its target value, and the algorithm trains on it, minimizing the number of false positives. Unsupervised learning is when the system has no idea about the target value and has to learn on its own. As YOLO is a supervised learning algorithm, the next step in training is to provide annotations for all 500 images for all the classes.

3.1 BBox Tool

For labeling all of the images our team used the BBox Tool. BBox Tool is an open source project that is frequently used in annotating images for machine learning applications. For annotating images one has to load all the images in the tool and then use a cursor to draw rectangular boxes around all objects of interest. This should be done for all the objects in all the images. After the image is labeled a corresponding text file is created with number of objects and then locations of all the objects in this format: "[bounding box left X] [bounding box top Y] [bounding box right X] [bounding box bottom Y]". These labels are what we called annotations for the image.

3.2 Converting Labels

As with most opens source technologies there is no proper integration between the output from the BBox Tool and YOLO training input. The input for YOLO is "[class number] [object center in X] [object center in Y] [object width in X] [object width in Y]".

For this we use a script to convert the labels for each of the files. It involves basic file reading and arithmetic to solve this problem. Once all the files are converted we can proceed to train the model.

CHAPTER 4

TRAINING

In this step we start the training phase as the dataset is ready. Two steps are required to start training.

4.1 Create Train File, Test File and Cfg Files

Before starting training one is required to create a train and test file from the dataset. These file are used by YOLO to test the status of its training. Along with this we also create idc*.data* file which has number of classes being trained, location of train file, location of test file, location of idc.names file and location of directory to be used to store backup weights. The idc.names file has a list of all the classes ordered based on the class number used while annotating. Lastly we use the standard tiny-yolo-v2*.cfg* file and edit the number of classes and filter value for that number. Save this renamed file as idc.cfg

4.2 Start Training

Once all the steps above are carried out we type *./darknet detector train cfg/idc.data cfg/idc.cfg tiny_yolo.conv.13* where tiny_yolo.conv.13 is the pre-trained starting point for YOLO. After the training has started, one has to monitor the output.log.

4.2.1 Issues While Training on Nvidia Jetson TX2

The first training was done on Nvidia Jetson TX2. While training, the system would crash every other hour due to out of memory error. This meant that one person had to stay the entire time and restart the process after every crash. Along with this because of the low computation power of the TX2 it took three days to finish 20,000 of the first single class training. After one set of training it was decided to ask for better hardware from the sponsors.

4.2.2 Training on Nvidia GeForce GTX 980 Ti

After we started training on Nvidia GeForce GTX 980 Ti, the process went very smoothly. As a result of high computing power and access to two Nvidia GeForce GTX 980 Ti, training with 40200 iterations took six to seven hours, which is exponentially faster than the Jetson TX2.

4.2.3 Conclusions About Training

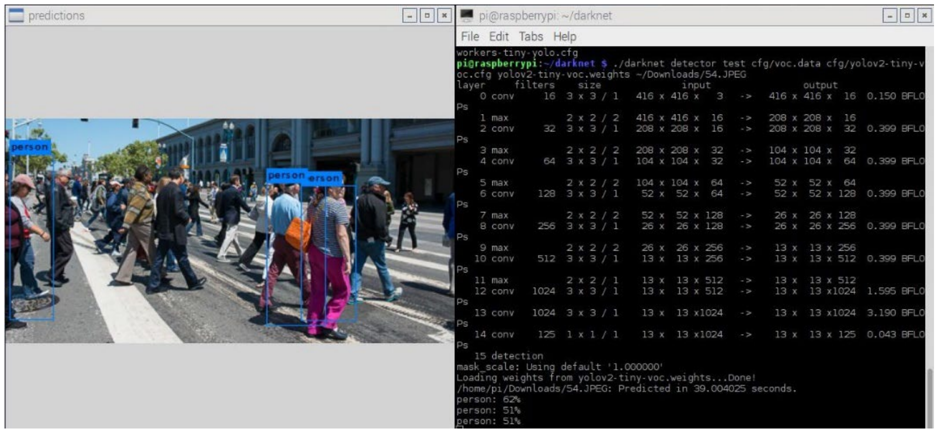
At this point we had decided that training should always be done on a system that is more powerful than a Jetson TX2. For this project our best option was to use Nvidia GeForce GTX 980 Ti.

CHAPTER 5

TESTING

This section discusses the independent work done for the Honors thesis by the author. In this section we will compare the trained model for the people class on different hardware systems. At the end of the section we will compare frame rate, cost and accuracy to reach a decision.

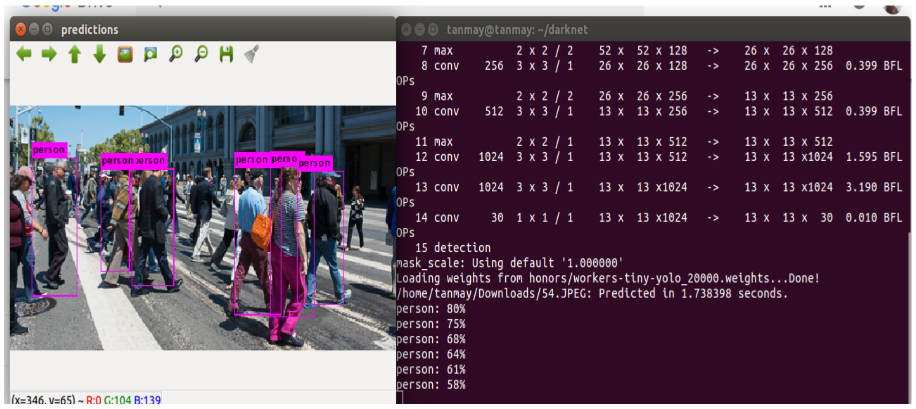
5.1 Test Results on Different Systems



```
workers-tiny-yolo.cfg
pi@raspberrypi:~/darknet $ ./darknet detector test cfg/voc.data cfg/yolov2-tiny-v
oc.cfg yolov2-tiny-voc.weights ~/Downloads/54.JPEG
layer     filters    size              input                output
    0 conv     16  3 x 3 / 1   416 x 416 x   3   ->   416 x 416 x  16  0.150 BFLO
Ps
    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 BFLO
Ps
    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 BFLO
Ps
    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 BFLO
Ps
    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 BFLO
Ps
    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 BFLO
Ps
   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 BFLO
Ps
   13 conv   1024  3 x 3 / 1    13 x  13 x1024   ->    13 x  13 x1024  3.190 BFLO
Ps
   14 conv    125  1 x 1 / 1    13 x  13 x1024   ->    13 x  13 x 125  0.043 BFLO
Ps
   15 detection
mask_scale: Using default '1.000000'
Loading weights from yolov2-tiny-voc.weights...Done!
/home/pi/Downloads/54.JPEG: Predicted in 39.004025 seconds.
person: 62%
person: 51%
person: 51%
```

Figure 5.1: Test Image from Raspberry Pi 3

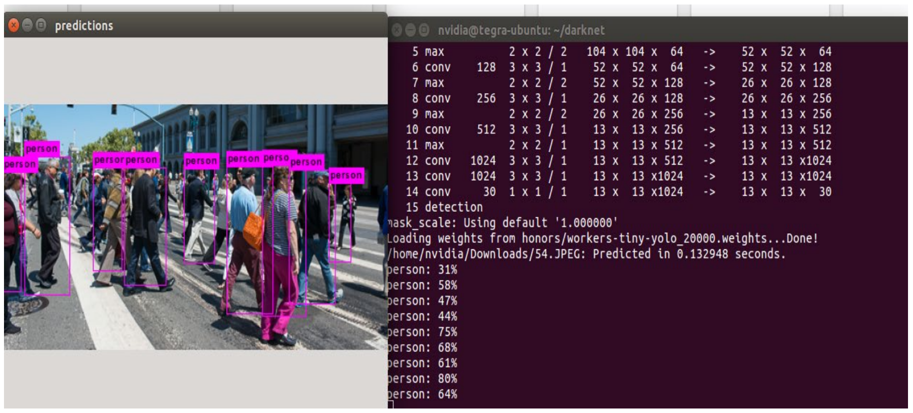
Figure 5.1 shows the output from Raspberry Pi 3. We can see that it took 39.004 seconds to process the image and it detected three people with a confidence in the range of 51%-62%. If we test video input on Raspberry Pi 3, then it crashes. This means that Raspberry Pi 3 cannot handle real-time detection.



```
    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   1024  3 x 3 / 1    13 x  13 x1024   ->    13 x  13 x1024  3.190 BFL
OPs
   14 conv     30  1 x 1 / 1    13 x  13 x1024   ->    13 x  13 x  30  0.010 BFL
OPs
   15 detection
mask_scale: Using default '1.000000'
Loading weights from honors/workers-tiny-yolo_20000.weights...Done!
/home/tanmay/Downloads/54.JPEG: Predicted in 1.738398 seconds.
person: 80%
person: 75%
person: 68%
person: 64%
person: 61%
person: 58%
```

Figure 5.2: Test Image from Intel i5 4th Gen CPU

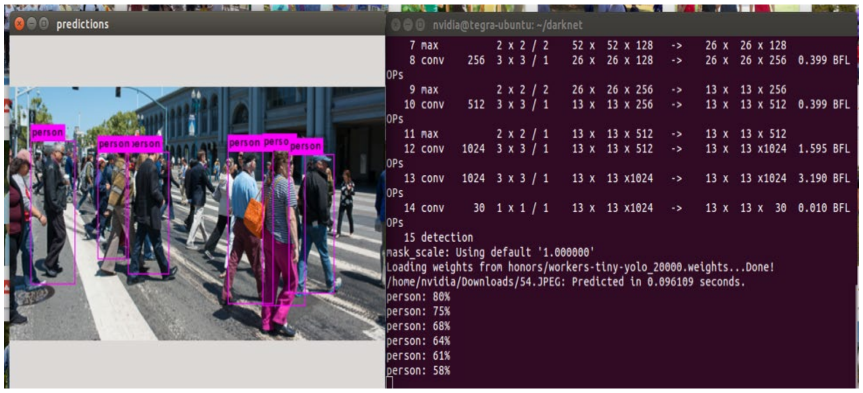
Figure 5.2 shows the output from Intel i5 4th Gen CPU. We see that it took 1.738 seconds and it detects six people with the confidence ranging from 58%-80%. If we use video input on this setup, then we see a frame rate of 0.5 fps, which is really slow.



```
    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
    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
    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
   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
   13 conv   1024  3 x 3 / 1    13 x  13 x1024   ->    13 x  13 x1024
   14 conv     30  1 x 1 / 1    13 x  13 x1024   ->    13 x  13 x  30
   15 detection
mask_scale: Using default '1.000000'
Loading weights from honors/workers-tiny-yolo_20000.weights...Done!
/home/nvidia/Downloads/54.JPEG: Predicted in 0.132948 seconds.
person: 31%
person: 58%
person: 47%
person: 44%
person: 75%
person: 68%
person: 61%
person: 80%
person: 64%
```

Figure 5.3: Test Image from Nvidia Jetson TX1

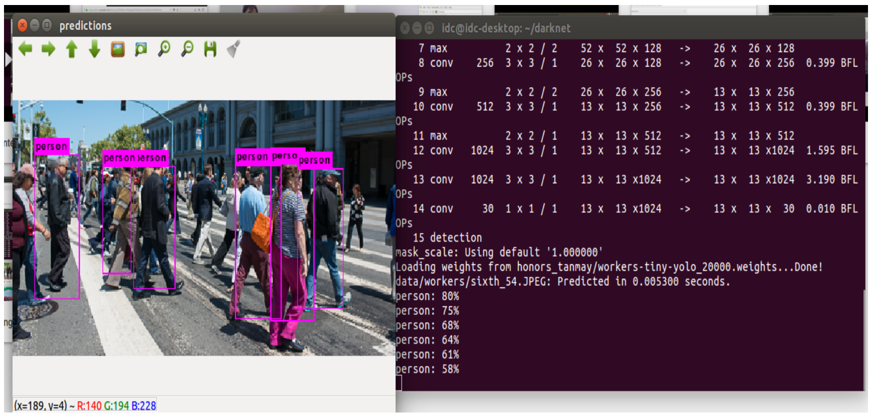
Figure 5.3 shows the output from Nvidia Jetson TX1. We see that it provides the same output as the i5 CPU, but the time required to compute is 0.136, which is about 20 times faster than the CPU. If we use video input on this setup, then we see a frame rate of 10-12 fps.



```
    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   1024  3 x 3 / 1    13 x  13 x1024   ->    13 x  13 x1024  3.190 BFL
OPs
   14 conv     30  1 x 1 / 1    13 x  13 x1024   ->    13 x  13 x  30  0.010 BFL
OPs
   15 detection
mask_scale: Using default '1.000000'
Loading weights from honors/workers-tiny-yolo_20000.weights...Done!
/home/nvidia/Downloads/54.JPEG: Predicted in 0.096109 seconds.
person: 80%
person: 75%
person: 68%
person: 64%
person: 61%
person: 58%
```

Figure 5.4: Test Image from Nvidia Jetson TX2

Figure 5.4 shows the output from Nvidia Jetson Tx2. Time required to compute is 0.096 seconds. If we use video input on this setup, then we see a frame rate of 15-17 fps.



```
    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   1024  3 x 3 / 1    13 x  13 x1024   ->    13 x  13 x1024  3.190 BFL
OPs
   14 conv     30  1 x 1 / 1    13 x  13 x1024   ->    13 x  13 x  30  0.010 BFL
OPs
   15 detection
mask_scale: Using default '1.000000'
Loading weights from honors_tanmay/workers-tiny-yolo_20000.weights...Done!
data/workers/sixth_54.JPEG: Predicted in 0.005300 seconds.
person: 80%
person: 75%
person: 68%
person: 64%
person: 61%
person: 58%
```

Figure 5.5: Test Image from Nvidia GeForce GTX 980 Ti

Lastly we look at Figure 5.5. This is taken from the GeForce GTX 980 Ti. It shows the same output as the TX2 and CPU, but the time required is only 0.0053 seconds. If we use video input on this setup, then we see a frame rate of 60+ fps.

<u>5.2 Comparing Hardware</u>

Based of the above section we can see that GPUs are better than CPUs for neural network. GPUs are more than 100x faster for training and more than 20x faster for testing neural networks than a CPU. The bulk of our computation is multiplying big matrices (thanks to neural networks), so we want a card with high single precision performance. The fact that a GPU costs more does not necessarily mean it is better. Although in most cases it is better. In the following table we will compare the outputs from all the GPU tests:

Jetson TX1	Jetson TX2	GeForce GTX 980 Ti
NVIDIA Maxwell ™ 256 CUDA cores	NVIDIA Pascal™ 256 CUDA cores	NVIDIA Maxwell ™ 2816 CUDA cores
998 MHz	1300 Mhz	1000-1075 MHz
$499	$599	$569 + $1200
11 fps	16 fps	63 fps

Table 5.1: Comparing Hardware

The final decision of what GPU/CPU to buy is based on the use of the end product. If the user only needs to run it once every one to two minutes and does not mind the low confidence and quality of the result, then using a Raspberry Pi to capture an image every minute to run YOLO will do the job. If the consumer wants a real-time object detection system, then it is recommended to use Nvidia Jetson TX2, as it provides a decent frame rate at a decent price.

REFERENCES

AlexeyAB, "AlexeyAB/darknet," *GitHub*. [Online]. Available: https://github.com/AlexeyAB/darknet. [Accessed: 04-May-2018]

"Allen School News," *Allen School News » Allen School's Joseph Redmon wins Google Ph.D. Fellowship*. [Online]. Available: https://news.cs.washington.edu/2018/04/05/allen-schools-joseph-redmon-wins-google-ph-d-fellowship/. [Accessed: 04-May-2018].

Hardikvasa, "hardikvasa/google-images-download," GitHub. [Online]. Available: https://github.com/hardikvasa/google-images-download. [Accessed: 04-May-2018].

J. Redmon, *YOLO: Real-Time Object Detection*. [Online]. Available: https://pjreddie.com/darknet/yolo/. [Accessed: 04-May-2018].

"Start Training YOLO with Our Own Data," *Guanghan Ning's Blog*, 19-Oct-2016. [Online]. Available: http://guanghan.info/blog/en/my-works/train-yolo/. [Accessed: 04-May-2018].

N. Tijtgat, "How to train YOLOv2 to detect custom objects," *Timebutt.io*, 07-Jun-2017. [Online]. Available: https://timebutt.github.io/static/how-to-train-yolov2-to-detect-custom-objects/. [Accessed: 04-May-2018].

Redmon, Joseph, Farhadi, and Ali, "YOLO9000: Better, Faster, Stronger," *[1402.1128] Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition*, 25-Dec-2016. [Online]. Available: https://arxiv.org/abs/1612.08242. [Accessed: 04-May-2018]

“Redmon, Joseph, Farhadi, and Ali, “YOLOv3: An Incremental Improvement,” *[1402.1128] Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition*, 08-Apr-2018. [Online]. Available: https://arxiv.org/abs/1804.02767. [Accessed: 04-May-2018].

Redmon, Joseph, Divvala, Santosh, Girshick, Ross, Farhadi, and Ali, “You Only Look Once: Unified, Real-Time Object Detection,” *[1402.1128] Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition*, 09-May-2016. [Online]. Available: https://arxiv.org/abs/1506.02640. [Accessed: 04-May-2018].

BIOGRAPHICAL INFORMATION

Tanmay Sardesai is an Honors B.S. in Computer Science student at the University of Texas at Arlington. After completing his undergraduate degree he will pursue an M.S. in Computer Science at the University of California, Los Angeles. There he will specialize in Artificial Intelligence, Machine Learning, and Computer Vision. After completing his M.S., he plans to work in the industry prior to pursuing a Ph.D. in Computer Science.