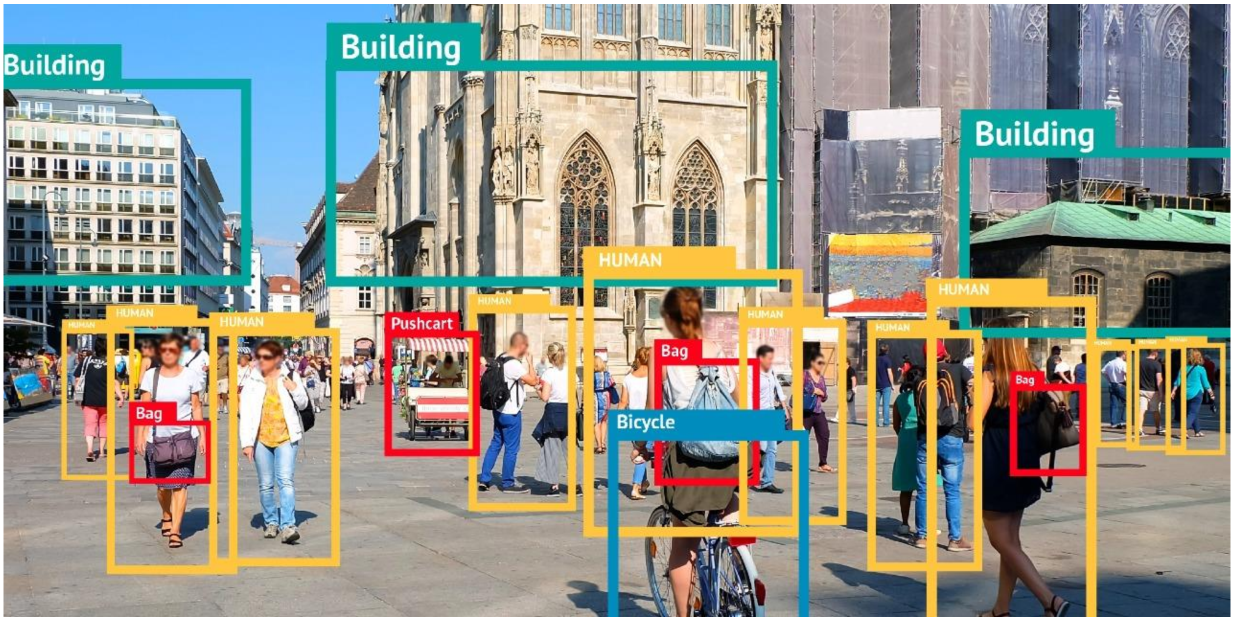
Final Project Report

Exploring Object Detection Its Applications



Ishan Khanka (IK1304)

INDEX

Introduction

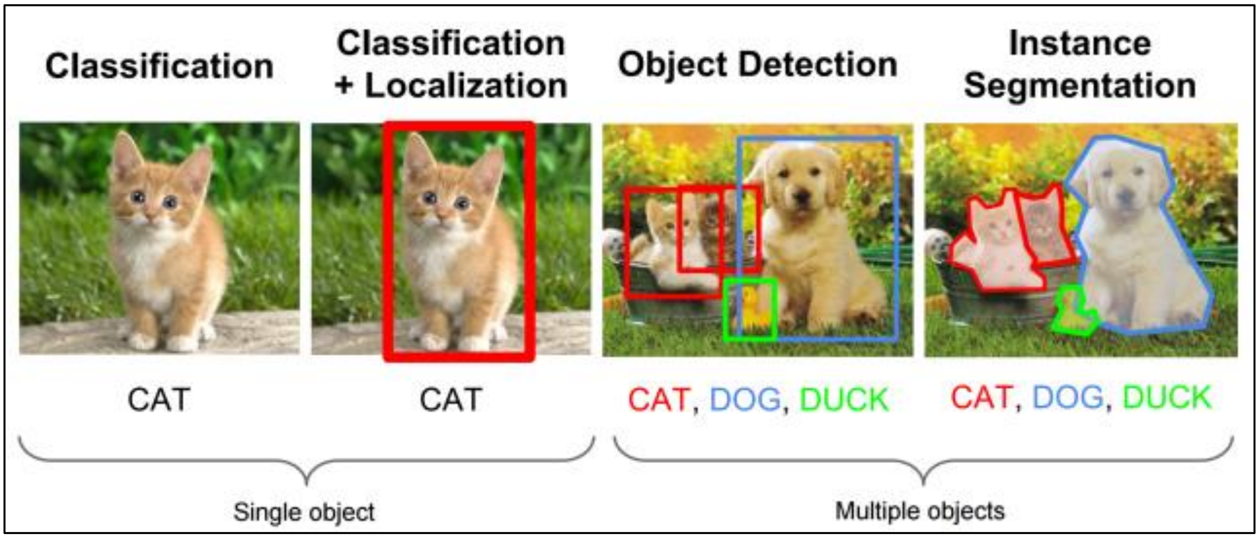
Object detection is one of the domains that has brought about a rapid and revolutionary change in the field of computer vision and deep learning. It deals with identifying and locating objects of certain classes in images. When humans look at images or video, we can recognize and locate objects of interest within a matter of moments. The goal of object detection is to replicate this intelligence using a computer. Given a set of object classes, object detection helps in determining the location and scale of all object instances, if any, that are present in an image.

Image Classification: Image classification is where a model can analyze an image and identify the 'target class' the image falls under essentially answering the question "What is in this picture?"

Object Localization: It is used to locate the presence of an object in the image and represent it with a bounding box in the form of (position, height, and width) essentially answering the question "What is it and where it is?"

However, in a real real-life scenario, we need to go beyond locating just one object in each image. We need systems that can detect multiple objects in an image and Object Detection provides the means for doing exactly that.

Object Detection: Object Detection algorithms act as a combination of Image classification and Object localization. It takes an image as input and produces one or more bounding boxes with the class label attached to each bounding box. Object detection models are capable enough to deal with multi-class classification, object localization as well as with objects with multiple occurrences.



Problem Statement and Project Overview

- Our objective is to build a model for object detection using image datasets.
- We plan to run the model on GPU's available on cloud platforms and Baremetal.
- We seek to analyze/compare their relative performance on these different platforms based on the execution time as well as the model's loss function thereby accounting for both the cloud and machine learning aspects of this course.
- Furthermore, we will be creating a web application that users could use to detect common objects or different types of blood cells in images uploaded by them.

Google Cloud

Bare Metal

Execution environments

As a part of this project we attempt to train models on the COCO and BCCD dataset using pytorch and run it on both BareMetal and Cloud platforms to access its relative performance on both platforms.

- **BareMetal:** A bare metal server is a single tenant physical server. Since they offer single-tenet environments i.e. a single servers' physical resources may not be shared between two or more tenets, they can be used to run dedicated services without any interruptions for longer durations. Bare-metal servers offer isolation and are free of the "noisy neighbor " effect that plagues virtual environments. Network latency is minimized for better performance, and the tenant enjoys root access. Bare metal is highly customizable, and the tenant may optimize the server based upon their individual needs.

  Bare metal servers do not require the use of several layers of software, unlike the virtual environment, which has at least one additional layer of software – a Type 1 hypervisor. This implies that there is one less layer of software between the user and the physical hardware in everyday use. Hence, we can expect better performance.

  For the Bare Metal setup, I'm using the NYU High Performance Cluster. The cluster comes with just the operating system and minimal tools for development. Any new tools or drivers to be installed can be installed using the "module load" command. There are restrictions placed on the user on installing third-party software. GPUs can be allotted to the user based on availability per the user's requirement and other resources such as memory and CPU cores are also provided when the user requests them. We train the model using **Tesla P40 GPU.**

- **Cloud:** Cloud offers a distributed environment comprised of multi-tenant, virtualized servers. The host machine shares its resources with multiple virtual instances. They each get a portion of CPU, RAM and storage. Cloud Servers allow you to add resources to individual virtual machines (vertical scaling) or add whole new servers (horizontal scaling) at any time, in a matter of minutes. This scalability makes Cloud Servers better suited to variable workloads, where the ability to dynamically scale performance is more important than sheer horsepower. Cloud computing environments are more prone to latency for various reasons. For example, if VMs are on separate networks, it can lead to packet delays. With cloud environments, you do not have a direct connection with the physical hardware, as there is a hypervisor layer between your app and physical resources. Thus, the chances are that VMs will suffer from a higher latency than if you were running apps directly on a bare metal server. Furthermore, performance bottlenecks may occur due to the sheer number of tenants. If you have noisy neighbors who like to run resource-intensive workloads on their share of the server, they may very well impact you leading to degraded performance.

  For the Bare Metal setup, I'm using the Google Colab. The Colab setup contrasts to the Bare Metal setup since it is provided as a Platform as a Service. The user is automatically allotted one **Tesla T4 GPU.**

Related work

For our project, we found the following resources to be extremely helpful:

Yolo v3:

https://pjreddie.com/darknet/yolo/

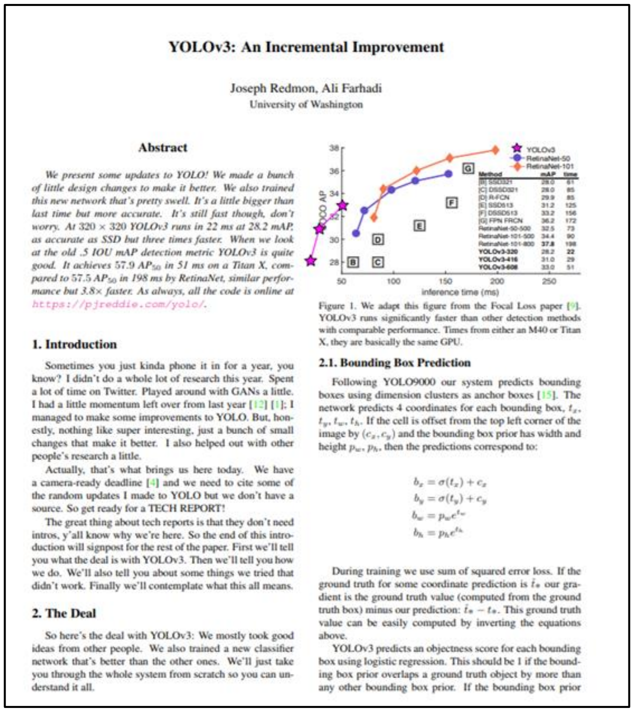
Papers With Code:

https://paperswithcode.com/method/yolov3

YOLO v3-Tiny: Object Detection and Recognition using one stage improved model
https://ieeexplore.ieee.org/document/9074315

Towards DataScience

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



YOLOv3: An Incremental Improvement

Joseph Redmon, Ali Farhadi
University of Washington

Abstract

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

1. Introduction

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

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

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

2. The Deal

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

Method	mAP	time
[B] SSD321	28.0	61
[C] DSSD321	28.0	85
[D] R-FCN	29.9	85
[E] SSD513	31.2	125
[F] DSSD513	33.2	156
[G] FPN FRCN	36.2	172
RetinaNet-50-500	32.5	73
RetinaNet-101-500	34.4	90
RetinaNet-101-800	**37.8**	198
YOLOv3-320	28.2	**22**
YOLOv3-416	31.0	29
YOLOv3-608	33.0	51

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

2.1. Bounding Box Prediction

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

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

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

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

Models Used

YOLO (“You Only Look Once”) is an effective real-time object recognition algorithm, first described in the seminal 2015 paper by Joseph Redmon.

Giving us an opportunity to analyze how the number of layers impacts the performance of the model, we used the following two versions of YOLO for the experiment design:

- Yolo v3
- Tiny-Yolo v3

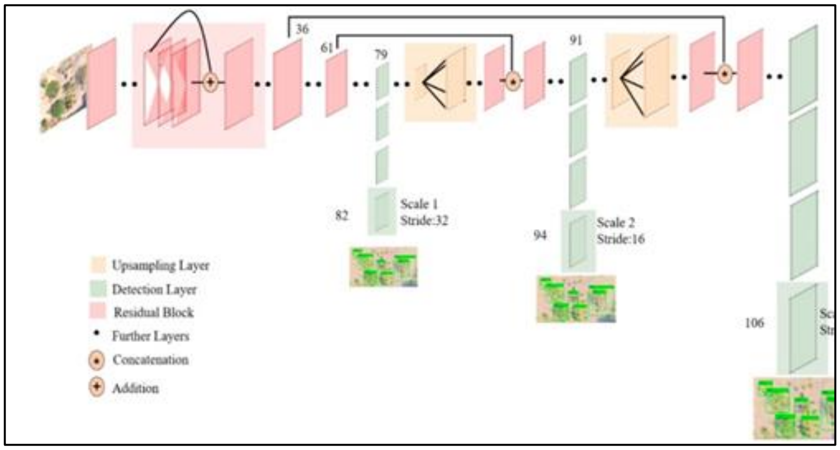
1. Yolov3

Yolov3 Network Architecture

YOLO v3 uses a variant of Darknet, which originally has a 53-layer network trained on ImageNet. For the task of detection, 53 more layers are stacked onto it, giving us a 106 layer fully convolutional underlying architecture for YOLO v3.

The most salient feature of v3 is that it makes detections at three different scales by down sampling the dimensions of the input image by 32, 16 and 8 respectively.

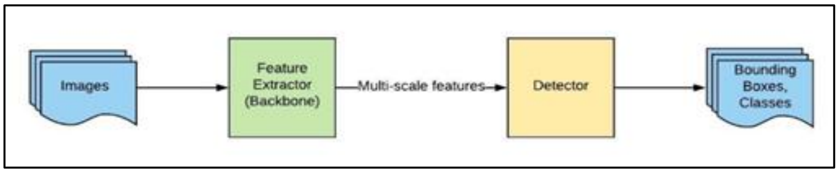
The first detection is made by the 82nd layer. Then, the second detection is made by the 94th layer and the final at the 106th layer.



Feature Extraction and Detection

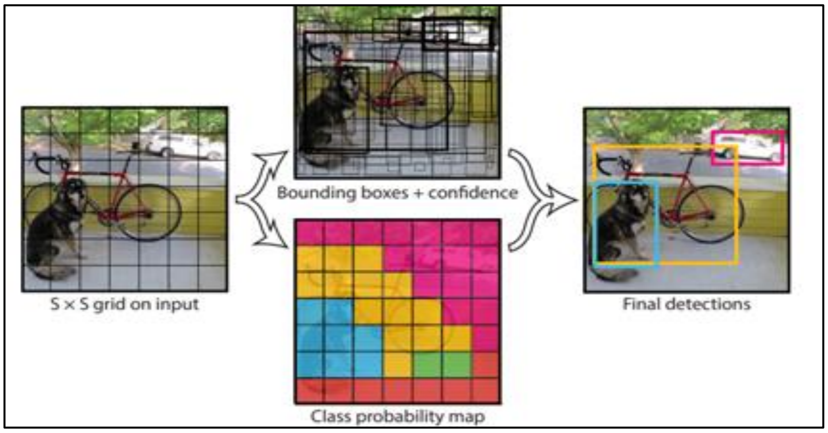
The whole system can be divided into two major components:

- Feature Extractor
- Detector



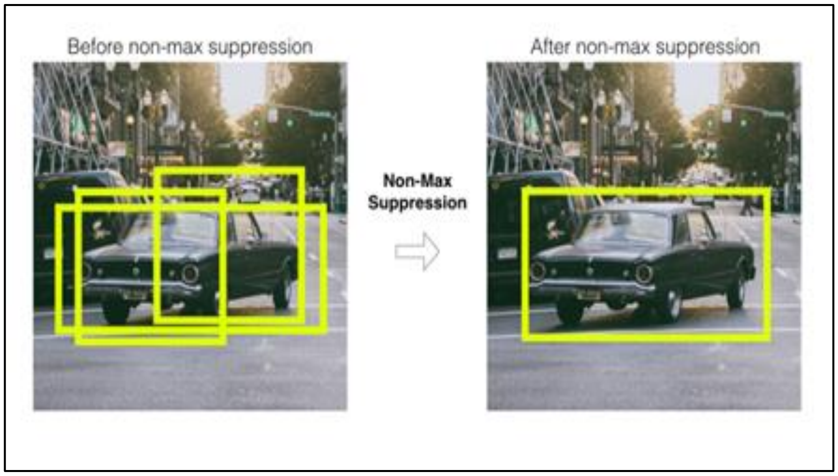
When a new image comes in, it goes through the feature extractor first so that we can obtain feature embeddings at three (or more) different scales. Then, these features are fed into three (or more) branches of the detector to get bounding boxes and class information.

We take an image and split it into an SxS grid, within each of the grid we take m bounding boxes. For each of the bounding box, the network outputs a class probability and offset values for the bounding box. The bounding box



having the class probability above a threshold value is selected and used to locate the object within the image.

With the YOLO algorithm we are not searching for interesting regions in our input image that could potentially contain an object. Instead, we are splitting our image into cells. Each cell is responsible for predicting bounding boxes. Therefore, we arrive at many bounding boxes for one image.
Most of these cells and bounding boxes will not contain an object.
Therefore, we use the value of pc- the probability that there is an object in the bounding box to remove/prune bounding boxes with low object probability and with the highest shared area. This process is called **non-max suppression.**

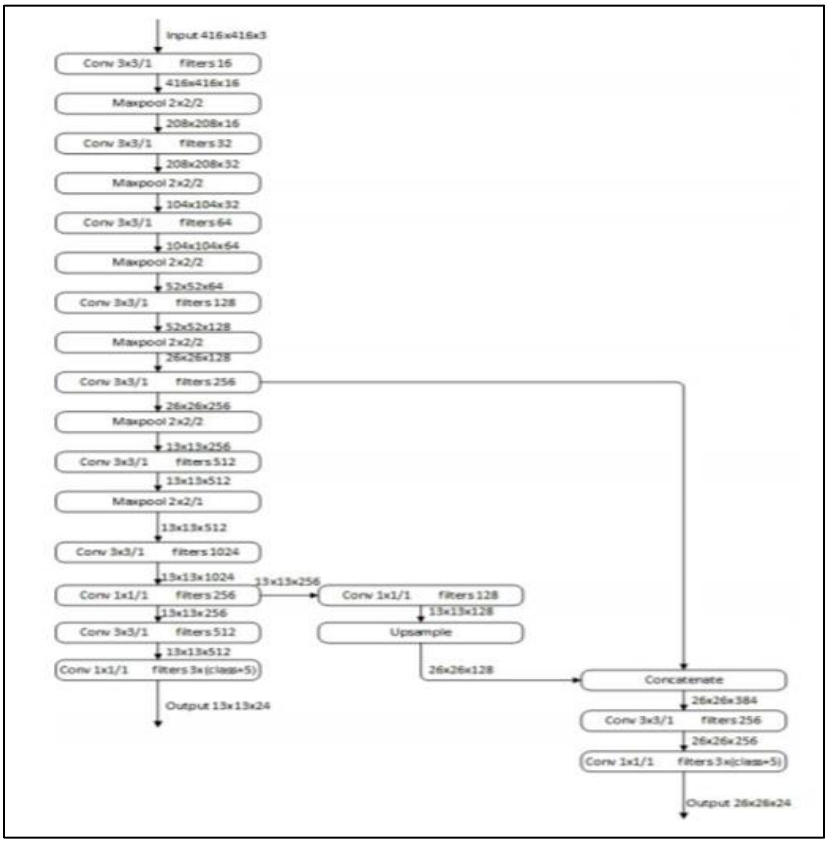


2. Tiny-Yolov3

The Tiny-yolov3 model is a simplified version of the YOLOV3 model.

YOLOV3 uses the architecture of darknet53, and then uses many 1x1 and 3x3 convolution kernels to extract features. Tiny-Yolov3 reduces the number of convolutional layers, its basic structure has only 7 convolutional layers, and then features are extracted by using a small number of 1x1 and 3x3 convolutional layers.

Tiny-Yolov3 uses the pooling layer instead of YOLO V3's convolutional layer with a step size of 2 to achieve dimensionality reduction.



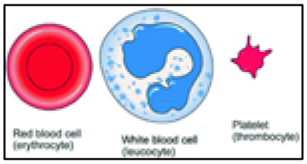
Datasets

As accuracy was not the focus for the assignment, for testing and performance two different datasets were used, one for common object detection and one for blood cell detection.

- **COCO**
  - Dataset Description
    - COCO stands for Common Objects in Context
    - There are 330K images (>200K labeled)
    - About 1.5 million object instances
    - 80 object categories
    - 5 captions per image
  - Object detection for COCO

Object detection is breaking into a wide range of industries, with use cases ranging from personal security to productivity in the workplace. Object detection and recognition is applied in many areas of computer vision, including image retrieval, security, surveillance, automated vehicle systems and machine inspection.

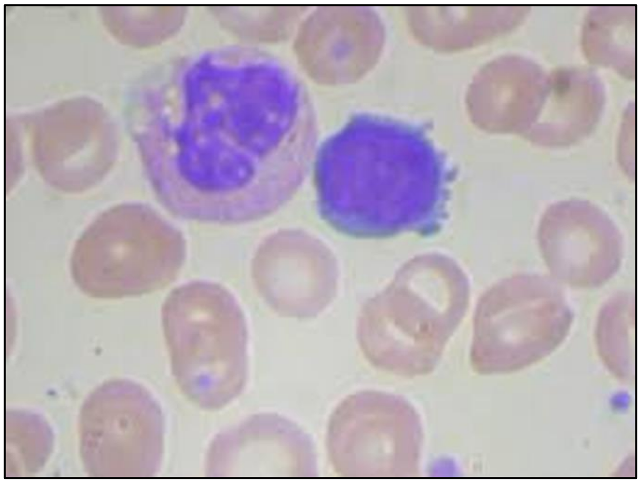
- **BCCD**
  - Dataset Description



- BCCD stands for Blood Cell Count and Detection
- There are 364 images across three classes:
  - WBC (white blood cells)
  - RBC (red blood cells),
  - Platelets.
- In total, there are 4888 labels across 3 classes

  - Object Detection for BCCD

Counting the number of white blood cells(WBC's),red blood cells(RBC's) and platelets in the blood is an important test for evaluating the health of an individual. The lack or excess of any one of these could be the sign of a potential disease and could lead to severe health problems. Traditionally blood cells are counted manually using hemocytometer along with other laboratory equipment and chemical compounds, which is a time-consuming and tedious task. Hence, the use of deep learning-based systems for detection and counting enables us to count blood cells from smear images in less than a second, which is useful for practical applications.



Steps and Execution

1. **Steps for Bare metal**

- `ssh ik1304@gw.hpc.nyu.edu`
- `ssh ik1304@prince.hpc.nyu.edu`
- Copy code from github to /scratch/ik1304 via winscp
- Get an Interactive node
  ```
  srun -t10:00:00 --gres=gpu:1 --mem 102400 --pty /bin/bash
  ```
- Load module. We are using the stable one
  ```
  module load python3/intel/3.6.3 cuda/9.0.176 nccl/cuda9.0/2.4.2
  ```
- Set up the python virtual environment
  ```
  mkdir pytorch_env
  cd pytorch_env
  virtualenv --system-site-packages py3.6.3
  source py3.6.3/bin/activate
  pip3 install http://download.pytorch.org/whl/cu92/torch-0.4.1-cp36-cp36mlinux_x86_64.whl
  pip3 install torchvision
  cd..
  pip3 install -r requirements.txt
  ```
- Activate the virtual environment
  ```
  source pytorch_env/py3.6.3/bin/activate
  ```
- Execute job with different variations of hyperparameters, for e.g.
  1. `/usr/bin/time -v nvprof python train.py --img 416 --batch 4 --epochs 1 --data bccd.yaml --weights yolov3.pt &> bare_metal_bs4_ep1_lr1.txt`
  2. `/usr/bin/time -v nvprof python train.py --img 416 --batch 8 --epochs 1 --data bccd.yaml --weights yolov3.pt &> bare_metal_bs8_ep1_lr1.txt`
  3. `/usr/bin/time -v nvprof python train.py --img 416 --batch 16 --epochs 1 --data bccd.yaml --weights yolov3.pt &> bare_metal_bs16_ep1_lr1.txt`
  4. `/usr/bin/time -v nvprof python train.py --img 416 --batch 32 --epochs 1 --data bccd.yaml --weights yolov3.pt &> bare_metal_bs32_ep1_lr1.txt`

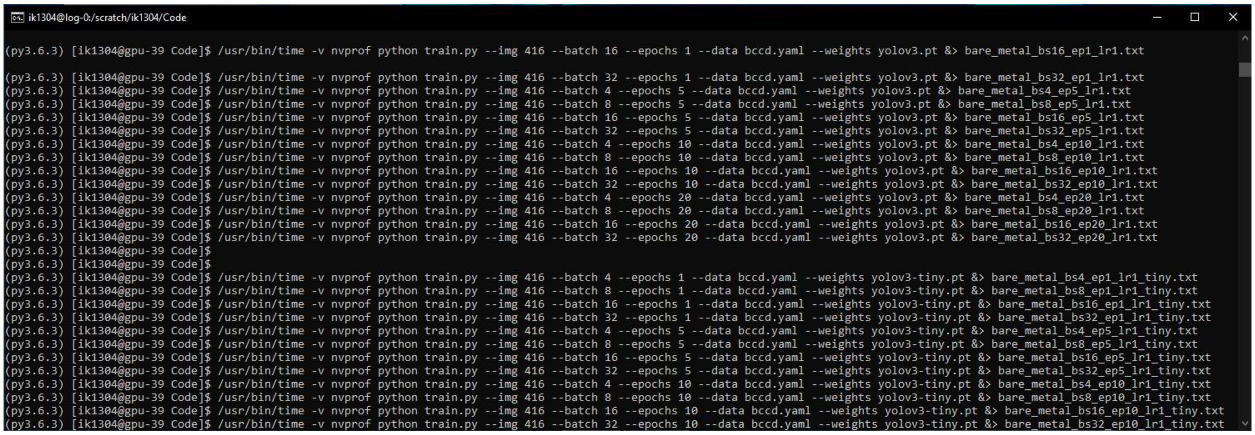
  .

  .

  .

  All the different permutations of runs are given in **baremetal run.txt**

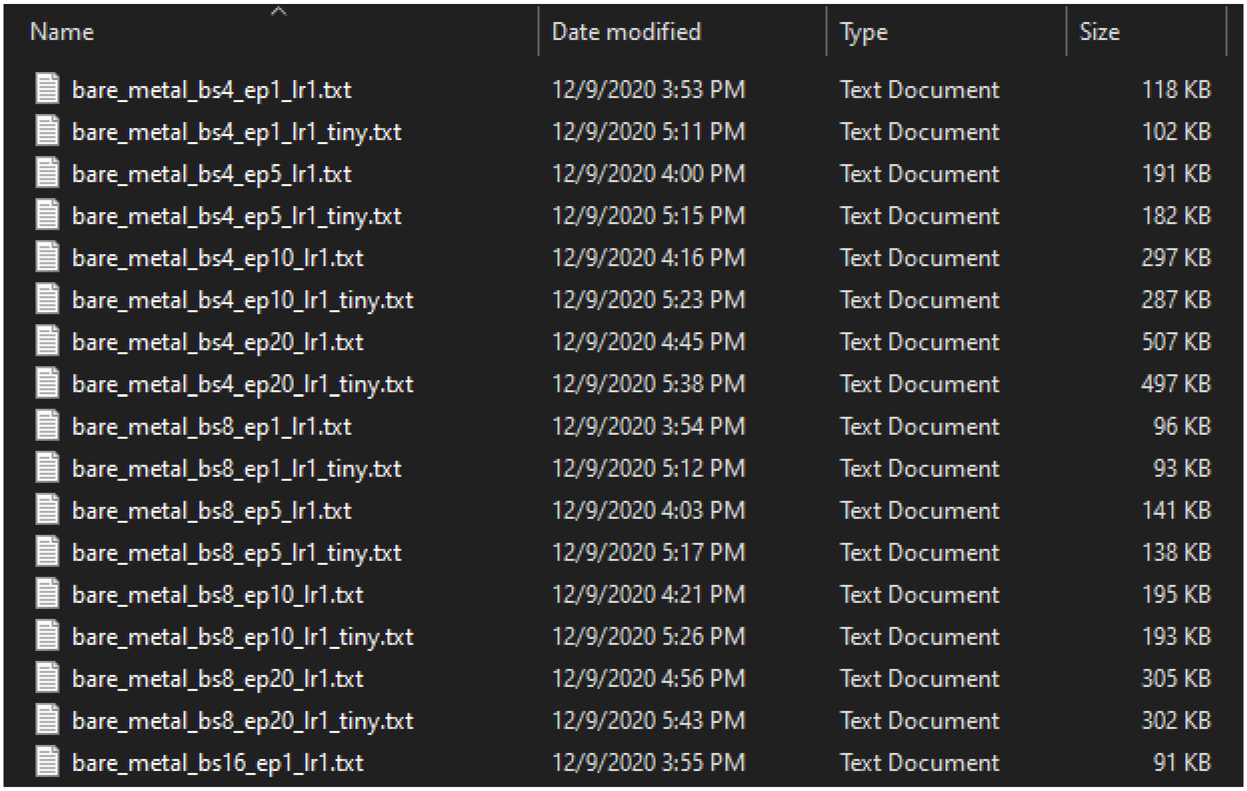
2. **Execution on Baremetal**



```
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 16 --epochs 1 --data bccd.yaml --weights yolov3.pt &> bare_metal_bs16_ep1_lr1.txt

(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 32 --epochs 1 --data bccd.yaml --weights yolov3.pt &> bare_metal_bs32_ep1_lr1.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 4 --epochs 5 --data bccd.yaml --weights yolov3.pt &> bare_metal_bs4_ep5_lr1.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 8 --epochs 5 --data bccd.yaml --weights yolov3.pt &> bare_metal_bs8_ep5_lr1.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 16 --epochs 5 --data bccd.yaml --weights yolov3.pt &> bare_metal_bs16_ep5_lr1.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 32 --epochs 5 --data bccd.yaml --weights yolov3.pt &> bare_metal_bs32_ep5_lr1.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 4 --epochs 10 --data bccd.yaml --weights yolov3.pt &> bare_metal_bs4_ep10_lr1.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 8 --epochs 10 --data bccd.yaml --weights yolov3.pt &> bare_metal_bs8_ep10_lr1.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 16 --epochs 10 --data bccd.yaml --weights yolov3.pt &> bare_metal_bs16_ep10_lr1.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 32 --epochs 10 --data bccd.yaml --weights yolov3.pt &> bare_metal_bs32_ep10_lr1.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 4 --epochs 20 --data bccd.yaml --weights yolov3.pt &> bare_metal_bs4_ep20_lr1.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 8 --epochs 20 --data bccd.yaml --weights yolov3.pt &> bare_metal_bs8_ep20_lr1.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 16 --epochs 20 --data bccd.yaml --weights yolov3.pt &> bare_metal_bs16_ep20_lr1.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 32 --epochs 20 --data bccd.yaml --weights yolov3.pt &> bare_metal_bs32_ep20_lr1.txt
(py3.6.3) [ik1304@gpu-39 Code]$
(py3.6.3) [ik1304@gpu-39 Code]$
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 4 --epochs 1 --data bccd.yaml --weights yolov3-tiny.pt &> bare_metal_bs4_ep1_lr1_tiny.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 8 --epochs 1 --data bccd.yaml --weights yolov3-tiny.pt &> bare_metal_bs8_ep1_lr1_tiny.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 16 --epochs 1 --data bccd.yaml --weights yolov3-tiny.pt &> bare_metal_bs16_ep1_lr1_tiny.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 32 --epochs 1 --data bccd.yaml --weights yolov3-tiny.pt &> bare_metal_bs32_ep1_lr1_tiny.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 4 --epochs 5 --data bccd.yaml --weights yolov3-tiny.pt &> bare_metal_bs4_ep5_lr1_tiny.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 8 --epochs 5 --data bccd.yaml --weights yolov3-tiny.pt &> bare_metal_bs8_ep5_lr1_tiny.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 16 --epochs 5 --data bccd.yaml --weights yolov3-tiny.pt &> bare_metal_bs16_ep5_lr1_tiny.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 32 --epochs 5 --data bccd.yaml --weights yolov3-tiny.pt &> bare_metal_bs32_ep5_lr1_tiny.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 4 --epochs 10 --data bccd.yaml --weights yolov3-tiny.pt &> bare_metal_bs4_ep10_lr1_tiny.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 8 --epochs 10 --data bccd.yaml --weights yolov3-tiny.pt &> bare_metal_bs8_ep10_lr1_tiny.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 16 --epochs 10 --data bccd.yaml --weights yolov3-tiny.pt &> bare_metal_bs16_ep10_lr1_tiny.txt
(py3.6.3) [ik1304@gpu-39 Code]$ /usr/bin/time -v nvprof python train.py --img 416 --batch 32 --epochs 10 --data bccd.yaml --weights yolov3-tiny.pt &> bare_metal_bs32_ep10_lr1_tiny.txt
```

Note – Logs for each of the experiment is stored in the root dir under logs/logs-baremetal

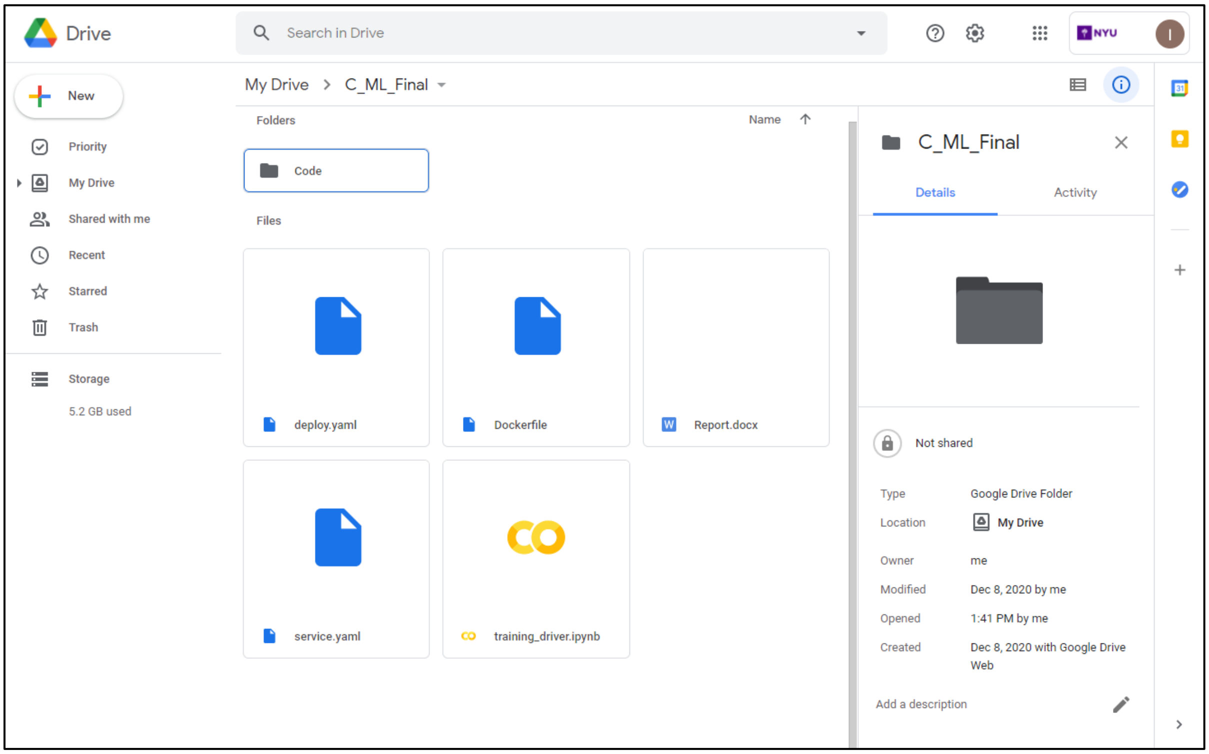


Name	Date modified	Type	Size
bare_metal_bs4_ep1_lr1.txt	12/9/2020 3:53 PM	Text Document	118 KB
bare_metal_bs4_ep1_lr1_tiny.txt	12/9/2020 5:11 PM	Text Document	102 KB
bare_metal_bs4_ep5_lr1.txt	12/9/2020 4:00 PM	Text Document	191 KB
bare_metal_bs4_ep5_lr1_tiny.txt	12/9/2020 5:15 PM	Text Document	182 KB
bare_metal_bs4_ep10_lr1.txt	12/9/2020 4:16 PM	Text Document	297 KB
bare_metal_bs4_ep10_lr1_tiny.txt	12/9/2020 5:23 PM	Text Document	287 KB
bare_metal_bs4_ep20_lr1.txt	12/9/2020 4:45 PM	Text Document	507 KB
bare_metal_bs4_ep20_lr1_tiny.txt	12/9/2020 5:38 PM	Text Document	497 KB
bare_metal_bs8_ep1_lr1.txt	12/9/2020 3:54 PM	Text Document	96 KB
bare_metal_bs8_ep1_lr1_tiny.txt	12/9/2020 5:12 PM	Text Document	93 KB
bare_metal_bs8_ep5_lr1.txt	12/9/2020 4:03 PM	Text Document	141 KB
bare_metal_bs8_ep5_lr1_tiny.txt	12/9/2020 5:17 PM	Text Document	138 KB
bare_metal_bs8_ep10_lr1.txt	12/9/2020 4:21 PM	Text Document	195 KB
bare_metal_bs8_ep10_lr1_tiny.txt	12/9/2020 5:26 PM	Text Document	193 KB
bare_metal_bs8_ep20_lr1.txt	12/9/2020 4:56 PM	Text Document	305 KB
bare_metal_bs8_ep20_lr1_tiny.txt	12/9/2020 5:43 PM	Text Document	302 KB
bare_metal_bs16_ep1_lr1.txt	12/9/2020 3:55 PM	Text Document	91 KB

3. **Steps for Public Cloud Code**

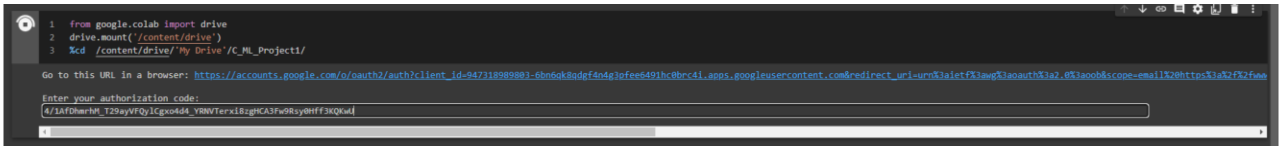
- Login to Google Colaboratory via https://colab.research.google.com/notebooks/welcome.ipynb

- Upload the code containing training_driver.ipynb to the google cloud storage:



Type	Google Drive Folder
Location	My Drive
Owner	me
Modified	Dec 8, 2020 by me
Opened	1:41 PM by me
Created	Dec 8, 2020 with Google Drive Web

- Authorize colab notebook to access the storage:

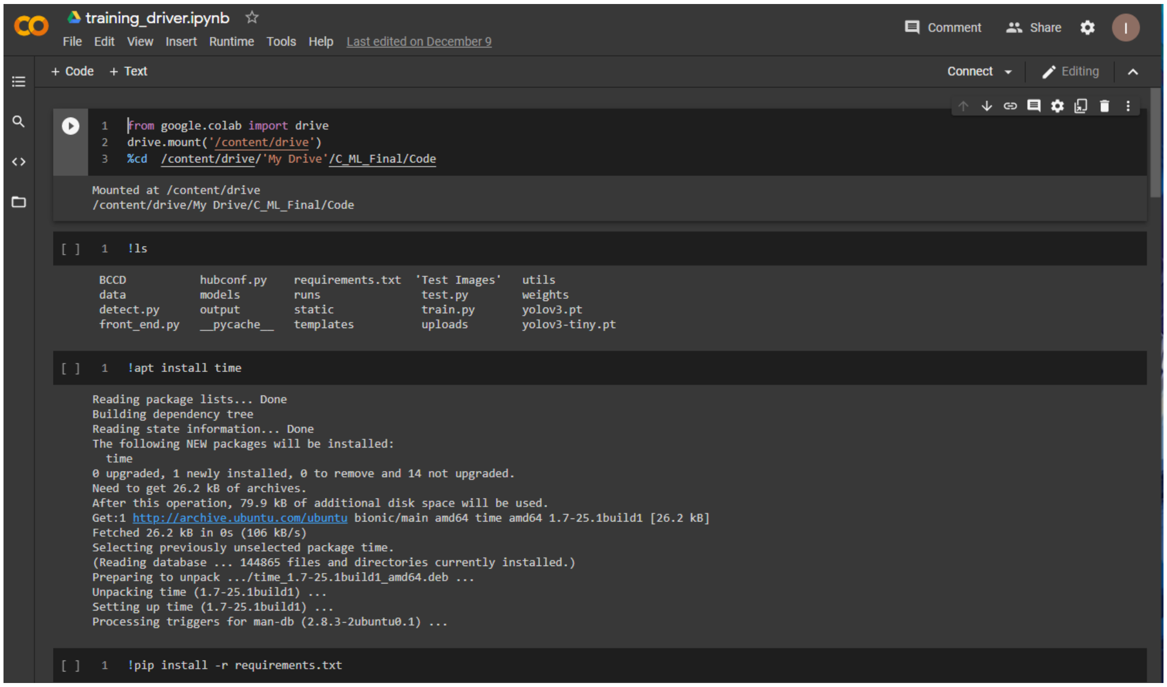


```
from google.colab import drive
drive.mount('/content/drive')
%cd  /content/drive/'My Drive'/C_ML_Project1/
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&scope=email%20https%3a%2f%2fwww

Enter your authorization code:
4/1AfDhmrhM_T29ayVFQylCgxo4d4_YRNVTerxi8zgHCA3Fw9Rsy0Hff3KQKwU
```

- Training_driver.ipynb has all the necessary code to install dependencies and access the storage


training_driver.ipynb
File Edit View Insert Runtime Tools Help Last edited on December 9
Comment
Share
+ Code + Text
Connect
Editing

```
from google.colab import drive
drive.mount('/content/drive')
%cd  /content/drive/'My Drive'/C_ML_Final/Code
```

```
Mounted at /content/drive
/content/drive/My Drive/C_ML_Final/Code
```

```
!ls
```

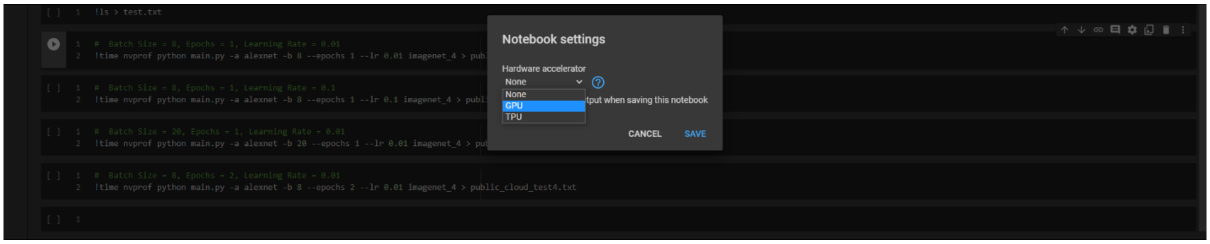
```
BCCD          hubconf.py    requirements.txt   'Test Images'   utils
data          models        runs                test.py        weights
detect.py     output        static              train.py       yolov3.pt
front_end.py  __pycache__   templates           uploads        yolov3-tiny.pt
```

```
!apt install time
```

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  time
0 upgraded, 1 newly installed, 0 to remove and 14 not upgraded.
Need to get 26.2 kB of archives.
After this operation, 79.9 kB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu bionic/main amd64 time amd64 1.7-25.1build1 [26.2 kB]
Fetched 26.2 kB in 0s (106 kB/s)
Selecting previously unselected package time.
(Reading database ... 144865 files and directories currently installed.)
Preparing to unpack .../time_1.7-25.1build1_amd64.deb ...
Unpacking time (1.7-25.1build1) ...
Setting up time (1.7-25.1build1) ...
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...
```

```
!pip install -r requirements.txt
```

- Before executing the notebook, go to Runtime > Change runtime type> Hardware accelerator> GPU



Hardware accelerator: GPU

```
!ls > test.txt
```

```
#  Batch Size = 8, Epochs = 1, Learning Rate = 0.01
!time nvprof python main.py -a alexnet -b 8 --epochs 1 --lr 0.01 imagenet_4 > pub
```

```
#  Batch Size = 8, Epochs = 1, Learning Rate = 0.1
!time nvprof python main.py -a alexnet -b 8 --epochs 1 --lr 0.1 imagenet_4 > publ
```

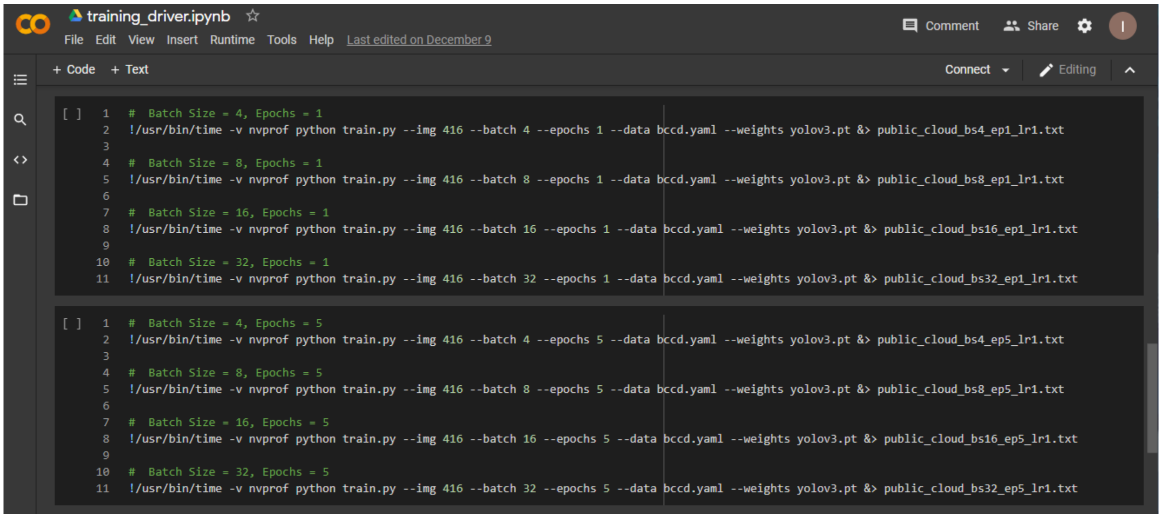
```
#  Batch Size = 20, Epochs = 1, Learning Rate = 0.01
!time nvprof python main.py -a alexnet -b 20 --epochs 1 --lr 0.01 imagenet_4 > pu
```

```
#  Batch Size = 8, Epochs = 2, Learning Rate = 0.01
!time nvprof python main.py -a alexnet -b 8 --epochs 2 --lr 0.01 imagenet_4 > public_cloud_test4.txt
```

- Run the cells of the notebook.

4. Execution on Cloud Environment

Run the cells of the notebook. Screenshots below show the code and profiler running:


training_driver.ipynb
File Edit View Insert Runtime Tools Help Last edited on December 9
Comment Share
+ Code + Text
Connect
Editing

```
#  Batch Size = 4, Epochs = 1
!/usr/bin/time -v nvprof python train.py --img 416 --batch 4 --epochs 1 --data bccd.yaml --weights yolov3.pt &> public_cloud_bs4_ep1_lr1.txt

#  Batch Size = 8, Epochs = 1
!/usr/bin/time -v nvprof python train.py --img 416 --batch 8 --epochs 1 --data bccd.yaml --weights yolov3.pt &> public_cloud_bs8_ep1_lr1.txt

#  Batch Size = 16, Epochs = 1
!/usr/bin/time -v nvprof python train.py --img 416 --batch 16 --epochs 1 --data bccd.yaml --weights yolov3.pt &> public_cloud_bs16_ep1_lr1.txt

#  Batch Size = 32, Epochs = 1
!/usr/bin/time -v nvprof python train.py --img 416 --batch 32 --epochs 1 --data bccd.yaml --weights yolov3.pt &> public_cloud_bs32_ep1_lr1.txt
```

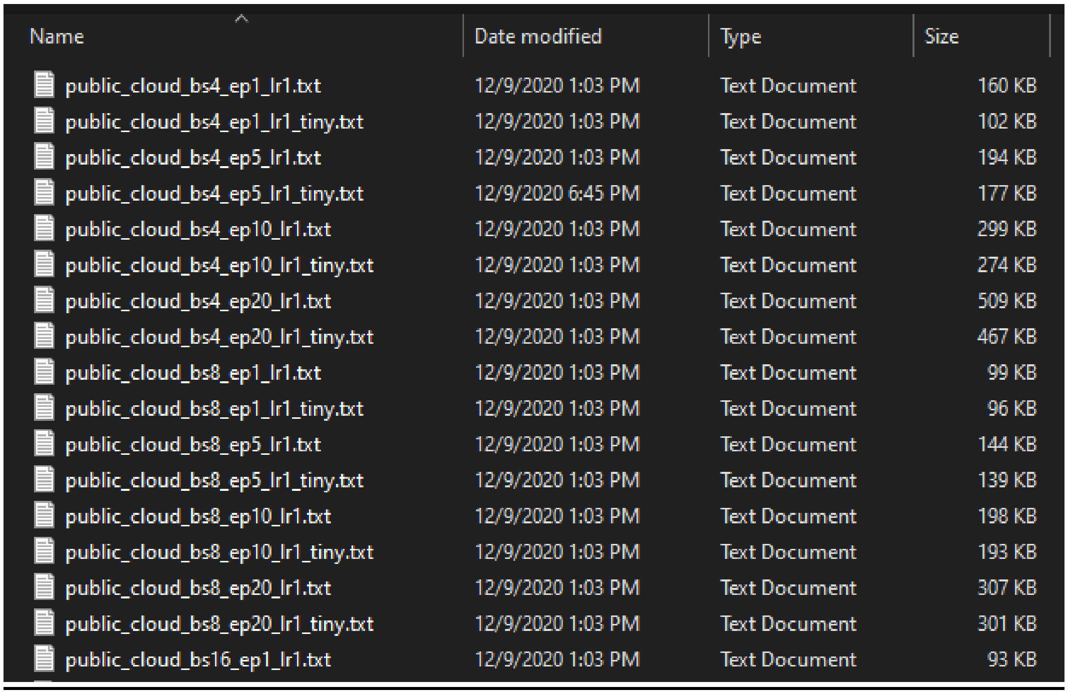
```
#  Batch Size = 4, Epochs = 5
!/usr/bin/time -v nvprof python train.py --img 416 --batch 4 --epochs 5 --data bccd.yaml --weights yolov3.pt &> public_cloud_bs4_ep5_lr1.txt

#  Batch Size = 8, Epochs = 5
!/usr/bin/time -v nvprof python train.py --img 416 --batch 8 --epochs 5 --data bccd.yaml --weights yolov3.pt &> public_cloud_bs8_ep5_lr1.txt

#  Batch Size = 16, Epochs = 5
!/usr/bin/time -v nvprof python train.py --img 416 --batch 16 --epochs 5 --data bccd.yaml --weights yolov3.pt &> public_cloud_bs16_ep5_lr1.txt

#  Batch Size = 32, Epochs = 5
!/usr/bin/time -v nvprof python train.py --img 416 --batch 32 --epochs 5 --data bccd.yaml --weights yolov3.pt &> public_cloud_bs32_ep5_lr1.txt
```

Note – Logs for each of the experiment is stored in the root dir under logs/logs-cloud



Name	Date modified	Type	Size
public_cloud_bs4_ep1_lr1.txt	12/9/2020 1:03 PM	Text Document	160 KB
public_cloud_bs4_ep1_lr1_tiny.txt	12/9/2020 1:03 PM	Text Document	102 KB
public_cloud_bs4_ep5_lr1.txt	12/9/2020 1:03 PM	Text Document	194 KB
public_cloud_bs4_ep5_lr1_tiny.txt	12/9/2020 6:45 PM	Text Document	177 KB
public_cloud_bs4_ep10_lr1.txt	12/9/2020 1:03 PM	Text Document	299 KB
public_cloud_bs4_ep10_lr1_tiny.txt	12/9/2020 1:03 PM	Text Document	274 KB
public_cloud_bs4_ep20_lr1.txt	12/9/2020 1:03 PM	Text Document	509 KB
public_cloud_bs4_ep20_lr1_tiny.txt	12/9/2020 1:03 PM	Text Document	467 KB
public_cloud_bs8_ep1_lr1.txt	12/9/2020 1:03 PM	Text Document	99 KB
public_cloud_bs8_ep1_lr1_tiny.txt	12/9/2020 1:03 PM	Text Document	96 KB
public_cloud_bs8_ep5_lr1.txt	12/9/2020 1:03 PM	Text Document	144 KB
public_cloud_bs8_ep5_lr1_tiny.txt	12/9/2020 1:03 PM	Text Document	139 KB
public_cloud_bs8_ep10_lr1.txt	12/9/2020 1:03 PM	Text Document	198 KB
public_cloud_bs8_ep10_lr1_tiny.txt	12/9/2020 1:03 PM	Text Document	193 KB
public_cloud_bs8_ep20_lr1.txt	12/9/2020 1:03 PM	Text Document	307 KB
public_cloud_bs8_ep20_lr1_tiny.txt	12/9/2020 1:03 PM	Text Document	301 KB
public_cloud_bs16_ep1_lr1.txt	12/9/2020 1:03 PM	Text Document	93 KB

Demo

For inference, we dockerized the web application written using python and flask. This docker image was hosted on IBM cloud using Kubernetes for orchestration.

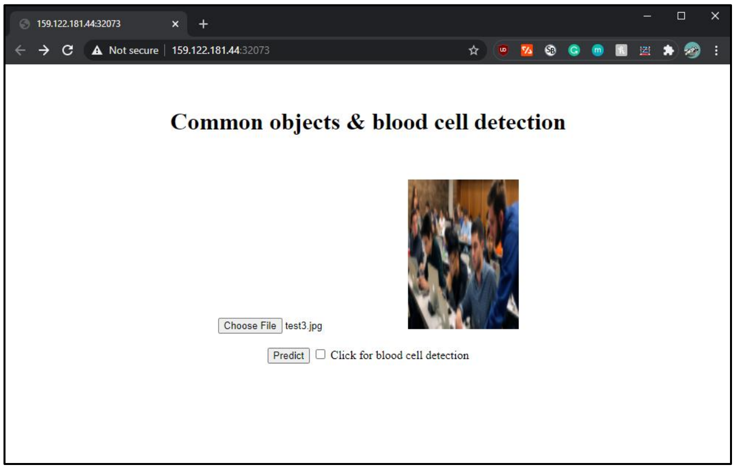
The inference web application can be accessed by going to the following link: http://159.122.181.44:32073/

Test Case 1 - Common Object Detection

For the first case, we choose a random image by clicking on Choose File button.

Next, we use the Predict button to start the common object detection.

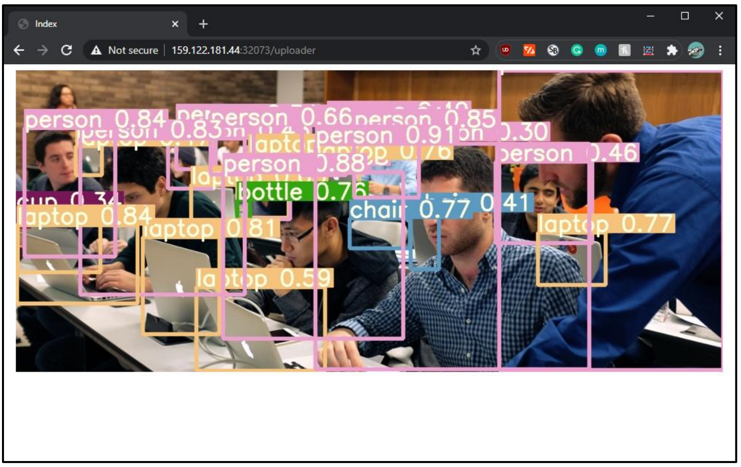
The resulting webpage shows the inference for the submitted image.



Common objects & blood cell detection

Choose File test3.jpg

Predict □ Click for blood cell detection



Left figure: Common object Image selection

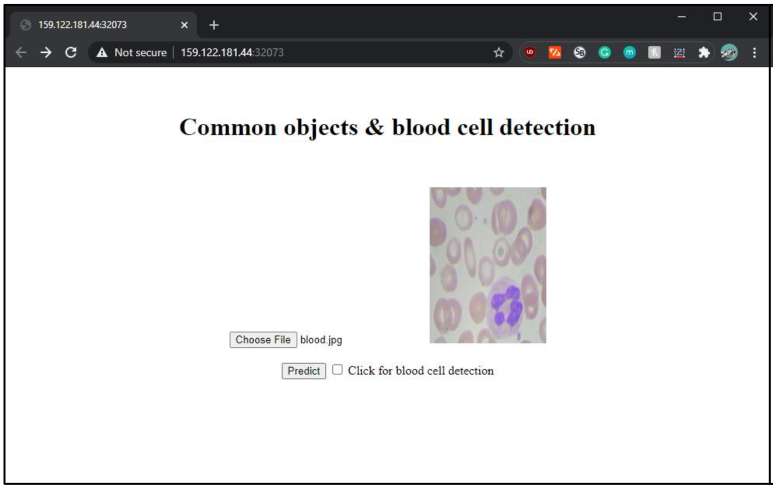
Right figure: Inference

Test Case 2 - Blood Cell Detection

For the first case, we choose a blood cell image by clicking on **Choose File** button.

Before clicking the **Predict** button, make sure that the **Checkbox** is ticked so that the web application knows to load the blood cell detection model.

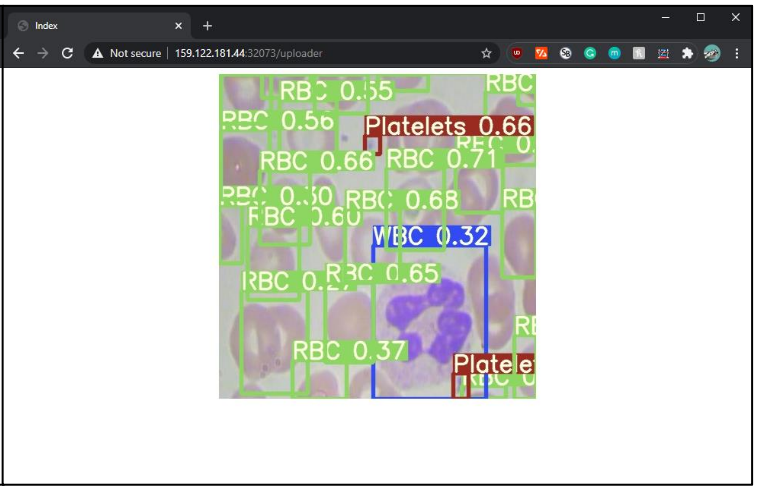
The resulting webpage shows the inference for the submitted image.



Common objects & blood cell detection

Choose File blood.jpg

Predict ☐ Click for blood cell detection



<u>Left figure</u>: Blood cell Image selection

<u>Right figure</u>: Inference

Evaluation

For evaluation of the model and the execution environment, we used the following hyperparameter configurations for train the model:

- Number of Epochs : 1, 5, 10 & 20
- Batch size : 4, 8, 16 & 32
- Models Chosen : YOLO & YOLO Tiny
- Execution environment : Cloud (Tesla T4, 15079MB) & BareMetal (Tesla P40, 22919MB)

Hence, the total number of experiments are **64**.

This setup was done to comprehensively cover all the different scenarios and to give up a deeper look into the model behavior and to analyze the impact of execution environment chosen for the training.

Now in order to evaluate and draw inferences for the model performance, the following metrics were chosen:

- Precision:
  tp / (tp + fp) or how many selected items are relevant
- Recall:
  tp / (tp + fn) or how many relevant items are selected
- Objectness score:
  To identify if an object is present in the image and the class of the object
- Classification score:
  To measure how well the detector identifies the locations and classes of objects during navigation

For the profiling of execution environment, the following metrics were recorded:

- User time (seconds)
- Percent of CPU a job got
- Maximum resident set size (kbytes)
- NVPROF profiling (Avg. time for biggest GPU activity)

Next four pages contain the metrics for 64 experiments.

Cloud : Tesla T4, 15079MB

LR : 0.2 – 0.01

Model : YOLO

Batch size	Epoch	User time (seconds)	Percent of CPU this job got	Maximum resident set size (kbytes)	NVPROF Avg time for Biggest GPU activity*	Precision	Recall	Objectness Score	Classification Score
4	1	59.68	20%	8078296	216.13us	0	0	0.1282	0.03834
4	5	219.07	124%	8127668	209.45us	0.1041	0.3146	0.1443	0.01785
4	10	412.60	124%	8145492	228.89us	0.4084	0.9464	0.1257	0.01051
4	20	800.69	125%	8249352	209.24us	0.5478	0.9803	0.115	0.003764
8	1	43.49	106%	8058616	404.16us	0	0	0.1263	0.03874
8	5	157.79	107%	8037768	403.41us	0.1001	0.1917	0.1548	0.02282
8	10	288.97	98%	8132920	406.81us	0.4042	0.51	0.137	0.01418
8	20	557.73	107%	8145180	408.72us	0.5477	0.9834	0.1197	0.005045
16	1	39.03	104%	8026880	831.73us	0	0	0.1227	0.03905
16	5	127.21	100%	8103536	793.00us	0.1392	0.002035	0.1618	0.02621
16	10	232.83	91%	8168672	790.48us	0.09563	0.3475	0.1449	0.0165
16	20	430.76	89%	7937896	787.32us	0.4309	0.9302	0.126	0.01053
32	1	36.94	96%	8188700	1.5461ms	0	0	0.121	0.03939
32	5	100.20	91%	8167160	1.5375ms	0	0	0.1598	0.02832
32	10	188.09	82%	8247332	1.5436ms	0.08012	0.2426	0.1582	0.01984
32	20	358.07	77%	8024508	1.5376ms	0.4358	0.5017	0.1361	0.01332

* GPU Process Name: turing_s1688cudnn_fp16_128x128_ldg8_wgrad_idx_exp_interior_nhwc_nt_v1

Cloud	: Tesla T4, 15079MB
LR	: 0.2 – 0.01
Model	: YOLO Tiny

Batch size	Epoch	User time (seconds)	Percent of CPU this job got	Maximum resident set size (kbytes)	NVPROF Avg time for Biggest GPU activity*	Precision	Recall	Objectness Score	Classification Score
4	1	35.41	86%	7791772	493.46us	0.3333	0.000407	0.3489	0.03383
4	5	113.99	142%	7872560	519.20us	0.1224	0.2856	0.3949	0.01768
4	10	211.06	148%	7983268	498.96us	0.4919	0.7248	0.3522	0.01251
4	20	403.25	149%	8031684	576.02us	0.4631	0.9638	0.3704	0.004965
8	1	31.69	118%	7791604	834.39us	0	0	0.3443	0.03426
8	5	89.82	135%	7906188	787.11us	0.07403	0.2478	0.3863	0.02012
8	10	159.81	140%	7991160	857.89us	0.3285	0.4478	0.3554	0.01533
8	20	305.24	144%	8043184	921.07us	0.3907	0.9397	0.3734	0.007637
16	1	31.04	121%	7831888	1.2530ms	0	0	0.3388	0.03472
16	5	72.78	120%	7971648	1.0918ms	0.06874	0.2362	0.3887	0.02186
16	10	133.78	132%	8068780	612.43us	0.09037	0.3128	0.3535	0.01665
16	20	254.68	136%	8055140	1.1414ms	0.3875	0.7995	0.3815	0.01247
32	1	30.89	121%	7997356	971.44us	0	0	0.3311	0.03511
32	5	67.87	123%	8032080	818.98us	0.1123	0.06577	0.4088	0.02338
32	10	125.99	140%	8166852	792.54us	0.06413	0.2804	0.3506	0.01776
32	20	229.88	138%	8271416	1.7303ms	0.3208	0.4627	0.3851	0.01511

*GPU Process Name: turing_s1688cudnn_fp16_128x128_ldg8_wgrad_idx_exp_interior_nhwc_nt_v1

BareMetal : Tesla P40, 22919MB

LR : 0.2 – 0.01

Model : YOLO

Batch size	Epoch	User time (seconds)	Percent of CPU this job got	Maximum resident set size (kbytes)	NVPROF Avg time for Biggest GPU activity*	Precision	Recall	Objectness Score	Classification Score
4	1	64.36	87%	4446340	536.73us	0	0	0.1311	0.03814
4	5	237.12	98%	4536808	535.41us	0.09708	0.3486	0.1499	0.01811
4	10	441.65	99%	8638968	534.50us	0.3902	0.9296	0.1257	0.01085
4	20	860.14	99%	16842640	535.31us	0.5031	0.9835	0.1114	0.004033
8	1	46.49	96%	4530028	1.0198ms	0	0	0.1361	0.03849
8	5	164.83	98%	4579324	1.0203ms	0.1114	0.1734	0.1517	0.02264
8	10	300.99	99%	5276692	1.0207ms	0.3616	0.495	0.1387	0.01367
8	20	563.84	99%	9962264	1.0192ms	0.573	0.9807	0.112	0.005232
16	1	37.34	94%	4676540	2.0153ms	0	0	0.1324	0.03895
16	5	119.94	97%	4792180	2.0123ms	0.3333	0.002409	0.1593	0.02588
16	10	226.52	98%	4894208	2.0142ms	0.08948	0.3385	0.1476	0.01615
16	20	416.45	99%	6382660	2.0382ms	0.426	0.9488	0.1187	0.01102
32	1	35.80	93%	4905660	3.9772ms	0	0	0.1301	0.03931
32	5	99.48	97%	4998552	3.9899ms	0	0	0.1567	0.02835
32	10	187.84	98%	5194808	3.9758ms	0.07667	0.2491	0.1617	0.01912
32	20	349.26	98%	5261972	3.9723ms	0.4492	0.5051	0.1269	0.01403

* GPU Process Name: maxwell_fp16_scudnn_winograd_fp16_fp32_128x128_ldg1_ldg4_tile228n_nt

BareMetal : Tesla P40, 22919MB

LR : 0.2 – 0.01

Model : YOLO Tiny

Batch size	Epoch	User time (seconds)	Percent of CPU this job got	Maximum resident set size (kbytes)	NVPROF Avg time for Biggest GPU activity*	Precision	Recall	Objectness Score	Classification Score
4	1	31.41	97%	4138872	307.62us	0	0	0.3455	0.03378
4	5	117.00	99%	4259288	297.86us	0.1039	0.3165	0.379	0.01791
4	10	216.31	99%	4466392	310.17us	0.415	0.752	0.388	0.01168
4	20	393.42	99%	7143760	313.74us	0.4289	0.969	0.3633	0.005112
8	1	27.35	98%	4255216	520.55us	0	0	0.338	0.0341
8	5	84.10	99%	4376384	520.37us	0.07189	0.2482	0.3741	0.0204
8	10	155.67	99%	4551084	544.44us	0.1691	0.354	0.39	0.01529
8	20	292.14	99%	4871752	558.41us	0.4673	0.9108	0.367	0.007233
16	1	26.23	98%	4487624	653.85us	0	0	0.3315	0.03449
16	5	69.10	99%	4599892	595.89us	0.07061	0.2212	0.3747	0.02196
16	10	125.14	99%	4770032	585.13us	0.08782	0.2987	0.3832	0.01689
16	20	Job Failed	Job Failed	Job Failed	Job Failed	Job Failed	Job Failed	Job Failed	Job Failed
32	1	28.16	98%	4626496	1.4311ms	0	0	0.3231	0.0349
32	5	64.06	99%	4804504	1.2067ms	0.1158	0.05495	0.3911	0.02353
32	10	112.38	99%	4931232	1.1705ms	0.05975	0.3012	0.3855	0.0181
32	20	207.22	99%	5136484	1.1501ms	0.3804	0.4844	0.3772	0.01412

*GPU Process Name: maxwell_fp16_scudnn_winograd_fp16_fp32_128x128_ldg1_ldg4_tile228n_nt

Inferences

For easier understanding, we have divided it into two parts:

Inferences regarding the Model

- Yolo Tiny trains faster irrespective of training platform (cloud or bare metal)
    - Inference: Lesser layers in Yolo tiny
- Both models reach higher objectness score for batch size 32 irrespective of the number of epochs
    - Inference: Move inclusive dataset (containing all the classes)
- Higher precision or recall
    - Inference:
        - For Blood Cell detection - recall is a better as our priority is how many relevant items are selected
        - For common object detection - precision is better as our priority is how many selected items are relevant

Inferences regarding the Execution Environment

- More consistent metrics for use of resources in bare metal for specific batch size
    - Inference: Due to no sharing of resources on bare metal
- Diff in time and way of handling same matrix multiplication while training – GPU in Cloud is faster:
    - Inference:
        - Different architecture of GPU (Maxwell – bare metal vs Turing - cloud)
        - Cloud give easy and faster access to latest (Sep 2018 vs Sep 2016) without maintenance costs
- Higher CPU usage in cloud
    - Inference: More movement of data in cloud as the data was streamed to and from other storage

Discussion: Challenges and Lessons Learned

Usability – user experience

Sr.no	Baremetal	Public Cloud
1.	Requires access to specialized tools like WinSCP and SSH clients to transfer code / data to the server	Most of the cloud providers has intuitive and easy ways to upload code / data
2.	User is responsible for solving dependencies of tools and libraries for the code to work	Most of the clouds have versatile collection and versions of libraries that get assigned automatically according to the code
3.	Availability of GPU due to multiple active users is a problem	Additional GPU can be easily provisioned with a single click
4.	Fixed resource allocation like memory and disk, which causes waste of resources due to idling	Automatic scaling of resources according to need

Other challenges and lessons learned

- Deploying object detection models generally takes a LOT of memory and computation power, especially on machines we use daily.
- We used Yolo tiny for inference due to faster and compact size
- Running 64 experiments and analyzing the results was a tedious and time-consuming task.
- Working our way through the Yolo framework required significant research as we needed to train it on custom dataset.
- To use the model on a custom dataset we had to figure a way to generate labels for the BCCD dataset as different object detection models use different label format.
- All object detection frameworks continue to struggle with small objects, especially those bunched together with partial occlusions as it can be seen on blood cell detection, where it misses out on some platelets.

Conclusion and future work

Conclusion

- Cloud provides access to the latest and faster GPU but BareMetal provides better reservation of resources.
- Tiny YOLO trains faster, gives nearly same performance with smaller model size.
- We were successful in creating a web application which can be used to detect both cases with just a flick of checkbox.
- The results discussed showed good insights for:
    - Model training – which metrics matters the most and how to select the hyperparameters for best accuracy.
    - Execution environment – how training the same application in different environments shows a stark difference:
        - For model evaluation and experimentation – BareMetal is preferred.
        - For deployment with low maintenance cost – Cloud is good.

Future work

- More in-depth evaluation of logs and model collected for each experiment (10 GB of data & 64,000 lines of logs) is required to gain more insights.
- An option to train the model with user provided hyperparameters can be added to the front end. For example, form fields where the user can key in hyperparameters like batch size, learning rate, epochs etc.

  Doing this would allow user to train and then immediately test out the performance of the model
- Instead of multi model approach, we can create single model for both use cases (blood cell and common object detection), hence the size of the model can be halved. But it is to be seen how this would impact the accuracy of the model.
- Lastly, image segmentation can be used for better localization of objects