ML1020 – Project
ImageNet Object Localization

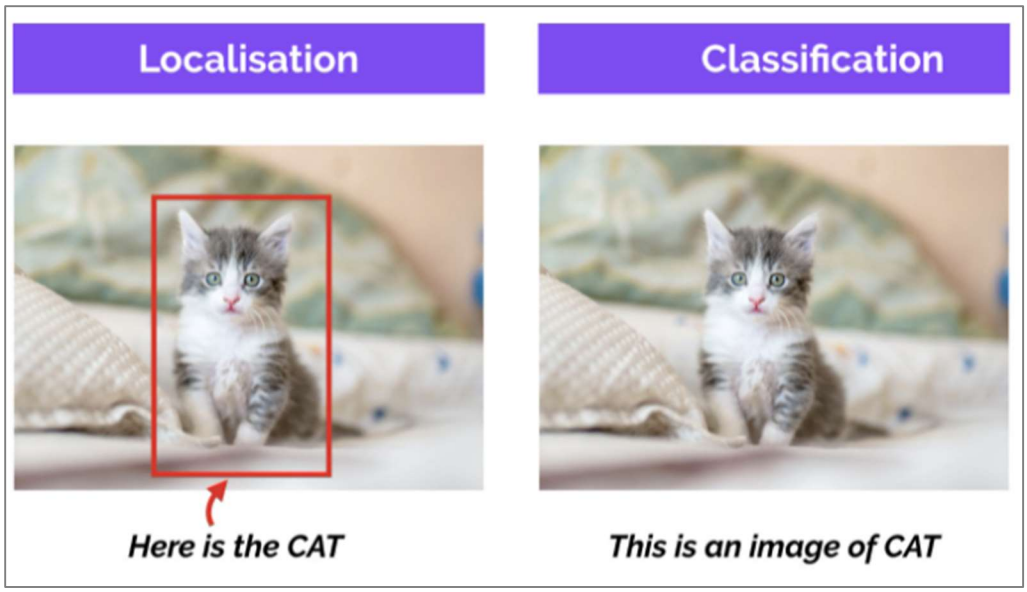
Submitted by: Michael Vasiliou

Contents

Executive Summary

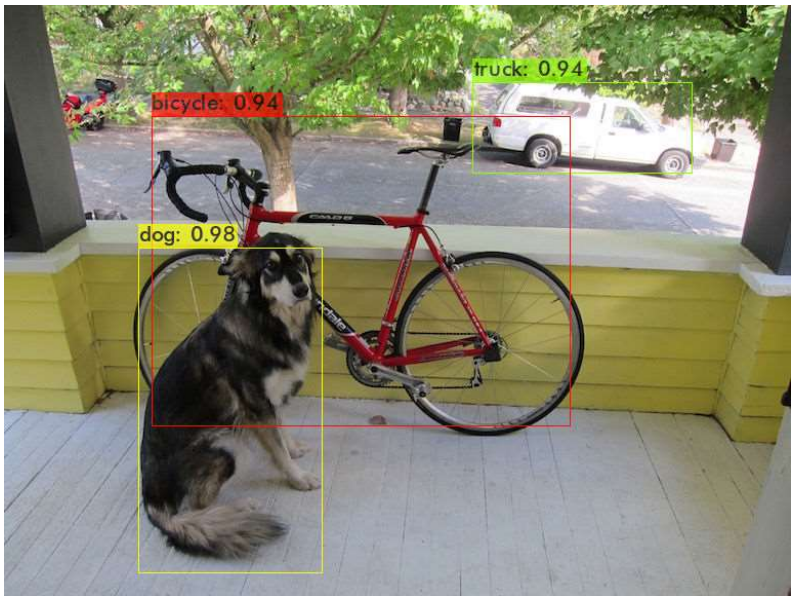
For my ML1020 project I will be working on the ImageNet Large Scale Visual Recognition Challenge. The ILSVCR is a challenge that evaluates algorithms for object detection and image classification at large scale.

The challenge consists of training a model to localize and classify 1000 different classes and the corresponding bounding boxes associated with the object in the image. While a standard classification challenge is one where you identify an item in an image, the localization challenge involves determining both what objects are in the picture as well as where they are in the picture.



Within the chosen dataset there the classes range from birds, spiders, snakes, dogs, candy, and seashores. Not only are there are large number of classes, but there are multiple classes that are extremely similar. For example, there are 17 types of classes for snakes such as: thunder snake, green snake, grass snake, vine snake, boa constrictor, Indian cobra.

The goal is to train a model which identifies the object within the image, it's associated bounding box, and prediction confidence level.



Data Overview

The original Kaggle challenge (https://www.kaggle.com/c/imagenet-object-localization-challenge/overview) was posted over 4 years ago and has received updates on the data files. The data files downloaded totaled 166GB and included over a million images and annotations for the training, validation, and test.

- Train: 1,281,167 image files, 544,546 annotations
- Val: 50,000 image files, 50,000 annotations
- Test: 100,000 (no annotation files)

Unpacking the data shows the following tree structure with root directory ILSVRC. Each subfolder under the first retains the same directory "CLS-LOC".

2. ILSVRC
    - Annotations
        - CLS-LOC
            - train (1,000 directories, 544,546 annotation files)
            - val (50,000 annotation files)
    - Data
        - CLS-LOC
            - test (100,000 images)
            - train (1,000 folders, 1,281,167 images)
            - val (50,000 images)
    - ImageSets
        - CLS-LOC
            - 4 text files with inventory of images

ILSVRC/Annotations/CLS-LOC Directory Overview

The Annotations directory contains XML files with the training and validation labels required for our model. The annotations are stored in XML format and include information about the images such as image dimensions, filename, class, and bounding box information. Each image could contain multiple objects on which to be trained.

```
<annotation>
    <folder>val</folder>
    <filename>ILSVRC2012_val_00050000</filename>
    <source>
        <database>ILSVRC_2012</database>
    </source>
    <size>
        <width>500</width>
        <height>375</height>
        <depth>3</depth>
    </size>
    <segmented>0</segmented>
    <object>
        <name>n02437616</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <bndbox>
            <xmin>237</xmin>
            <ymin>109</ymin>
            <xmax>446</xmax>
            <ymax>374</ymax>
        </bndbox>
    </object>
    <object>
        <name>n02437616</name>
        <pose>Unspecified</pose>
        <truncated>0</truncated>
        <difficult>0</difficult>
        <bndbox>
            <xmin>175</xmin>
            <ymin>89</ymin>
            <xmax>474</xmax>
            <ymax>342</ymax>
        </bndbox>
    </object>
</annotation>
```

There are differences in the setup between the “train” and the “val” folder. Within the “val” folder there are 50,000 XML annotation files, however, within the “train” folder there are 1000 subfolders, each representing a different class, then within each of those subfolders are multiple images, each of the same class, totally 544,546 XML files.

Note that not all image files have corresponding XML annotation/label files.

ILSVRC/Data/CLS-LOC Directory Overview

The “Data” directory contains three subfolders: “test”, “train”, and “val”. The “test” folder contains 100,000 images, and the “val” folder contains 50,000 images. The “train” folder structure mirrors the “train” directory structure previously noted under “Annotations” in that it contains 1000 subfolders, one for each class in our project, with 1,281,167 images contained within those 1000 subfolders.

ILSVRC/ImageSets/CLS-LOC Directory Overview

The “ImageSets” directory consists of 4 files:

1. “test.txt” – list of all the image names in the test directory without the file extension
2. “train_cls.txt” – list of all the image names in the “Data/CLS-LOC/train” directory without the file extension
3. “train_loc.txt” – list of all image names in the “Data/CLS-LOC/train” directory without the file extension, that have a corresponding XML annotation file
4. “val.txt” – list of all image names in the “Data/CLS-LOC/val” directory without the file extension

Algorithm/Model Chosen

The algorithm chosen to solve the object detection challenge is YOLO. YOLO stands for “You Only Look Once” and is a popular algorithm due to its speed and accuracy. When training a model for object detection there are two main goals:

1. What is the object?
2. Where is it in the image?

The YOLO algorithm requires only a single pass of the image and through regression provides the class probabilities, and bounding box of the detected images. Each image is divided into grids where every grid will detect objects that appear within them. YOLO uses a single bounding box regression to predict the height, width, center, and class of the objects. Because it is a single pass, it’s speed and usability are considered top tier.

The main package I will be using to train the model is called darknet. Darknet is the primary implementation of YOLO and can be used for training, prediction, and real time object detection in images and video streams.

It's extremely important to note that there are two main github repositories for utilizing YOLO, one by "pjreddie" and a fork by "AlexeyAB". Each of these two repositories have gone through several major updates over the past few years, and each of the packages has slightly different configurations, options, and documentation. The online documentation and previous code examples available online, while helpful, required quite some time to understand and modify to the new versions of YOLO.

Development Environment

Development and model training was not conducted on the cloud but rather a custom-built local server. A virtual machine was setup and configured under Linux QEMU/KVM with dedicated resources for this project.

Hardware

- CPU – Intel i9-12900K @ 5.2GHz (16 core/24 threads)
  - 8 hyperthreaded performance cores (16 threads)
  - 8 non-hyperthreaded efficiency cores (8 threads)
- Memory – 96GB DDR4 (64GB dedicated to training)
- Primary storage – WD SN850 1TB PCIe Gen4 NVME (read: 7GB/s, Write: 5.1GB/s)
- Secondary storage - Broadcom / LSI MegaRAID SAS 9361
  - Primary array: 36 GB (10 x 4GB Raid 5 on SAS controller)
  - Backup array: 36 GB (10 x 4GB Raid 5 on SAS controller)
- GPU 1 – Nvidia [GP106] GTX 1060 (6GB)
- GPU 2 – Nvidia [GM204] GTX 970 (4GB)
- Onboard graphics – Intel UHD Graphics 770
  - Note: Onboard graphics used for host display not model training

Software

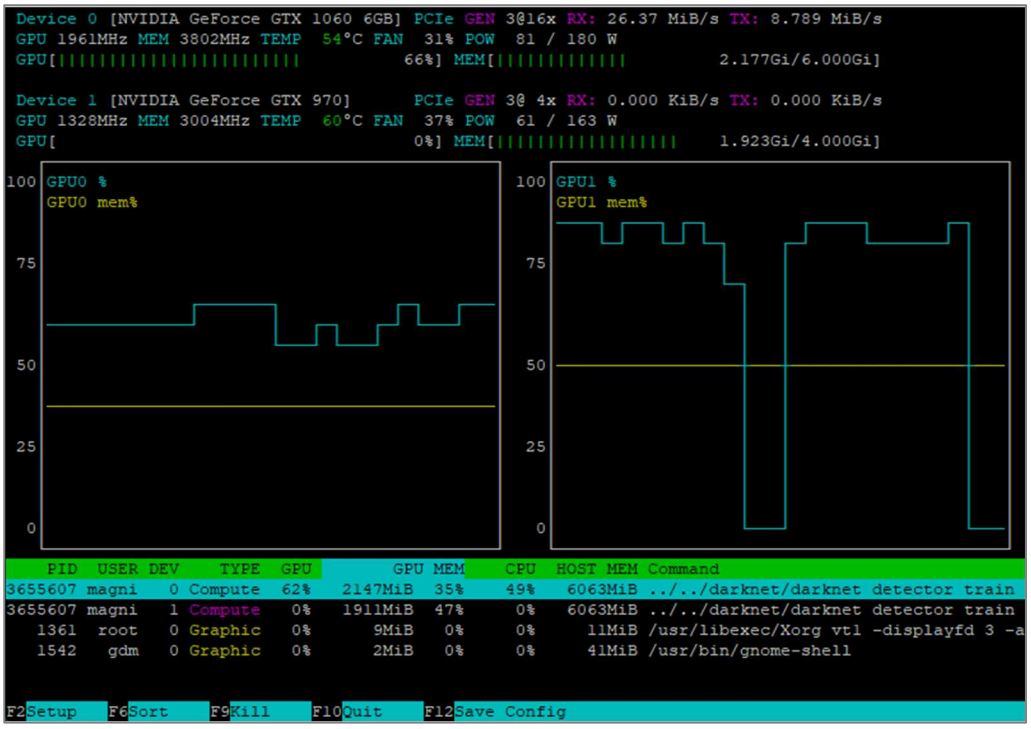
All experiments and development were conducted in a virtual machine with the GPUs being passed through as dedicated devices for the VM to use. Both the base OS managing the hardware and the VM used Fedora Linux 35 with QEMU/KVM as the VM software. Both Nvidia GPU's, 64GB Ram, 16 CPU cores, and dedicated NVMe storage were allocated to the VM to ensure sufficient performance and no resource bottlenecks.

Resource Monitoring (GPU, CPU, Disk Input/Output)

As various experiments and training unfolded it was important to monitor all system resources. There are numerous configuration options for YOLO training as well as numerous stages. Depending on the configuration as well as the stage some of them were constrained by CPU resources, some were constrained by GPU resources.

GPU Monitoring

During experiments and training GPU resources were monitored using "nvtop". GPU memory, utilization, temperature, data transfer and receive rates (TX/RX), and power consumption are monitored to ensure usage and no training concerns.

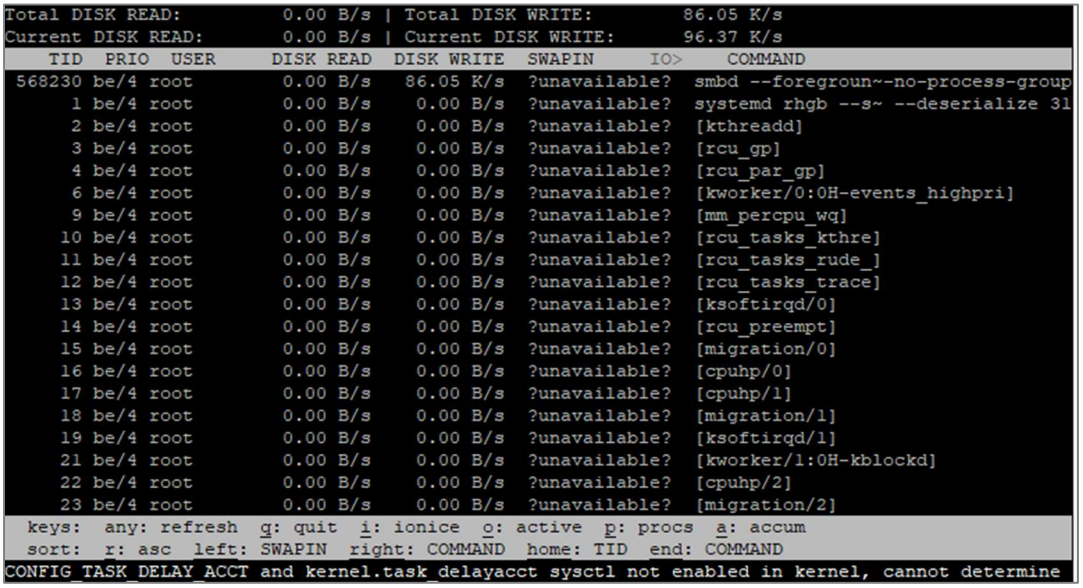


CPU Monitoring

CPU monitoring was performed using “s-tui” which shows detailed CPU information at the core level including, utilization, core clock frequency, and core/package temperatures.

I/O Monitoring

Monitoring of I/O was completed using “iotop”. The I/O subsystem was not expected to be a bottleneck based on NVMe transfer rates of 7GB/s write and 5GB/s read.



Hardware and Operating Environment

For simplicity this summary will not include physical hardware installation or operating system setup and configuration.

There are a few key factors when setting up a virtual machine with GPU passthrough on Linux that are important:

1. Virtual Machines under QEMU/KVM will be created using legacy BIOS instead of UEFI. If you are looking to use GPU passthrough (where the GPU is fully under the control of the virtual machine), or any other hardware passthrough, UEFI should be enabled.
2. GPU’s that are intended to be passed through to the virtual machine need to have a stub driver (VFIO) loaded at boot time so that the main operating system does not utilize them.

Within the virtual machine several packages and configurations must be made for GPU enabled training as well as utilization of full YOLO and darknet capability.

- Nvidia drivers
- CUDNN drivers (Nvidia support drivers)
- OpenCV

Once the supporting packages have been installed and configured, the installation and setup of darknet itself is rather trivial.

1. Download darknet from github. For this project I used the “AlexeyAB” version of darknet.
2. Modify “Makefile” in darknet directory, key parameters:
    - GPU=1 (set to 1 if you are configured for Nvidia GPU acceleration)
    - CUDNN=1 (set to 1 if you have installed CUDNN)
    - CUDNN_HALF=0 (Set to 1 if you have a high end Nvidia (non-consumer GPU)
    - OPENCV=1 (set to 1 if you have compiled and installed OpenCV)
        - OpenCV allows for auto visuals in detection, as well as real time object detection in video
    - OPENMP=1 (set to 1 if you have a multi-processor system)
3. Run “Make” in the darknet directory

*References for VFIO, stubbing, and boot parameter configuration, Nvidia drivers, and CUDNN installation are included in the references section

Training Configuration

At this point I have setup, installed, and tested all hardware and packages that are needed to complete the model training. The next steps were to get and prepare the data as well as configure the model for appropriate training. The process was exceptionally iterative where data preparation was completed, model parameters were configured, and training begun. Several rounds of 10-40hr training sessions were completed, each requiring additional data manipulation as well as model and training configuration, and each providing their own findings and challenges to overcome.

Data Preparation

Preparing the data for consumption by the YOLO models had several challenging components. It seems that while the label format that YOLO uses for model training and validation has not changed, the location

1. Download and unzip the competition data from Kaggle
    - ❖ Note: The file contents are described in the data section above

2. Convert labels from XML for consumption by YOLO model training

    Some helpful scripts were found online to assist in converting the data from the original XML format provided in the Kaggle download to the “txt” file required by YOLO. Older scripts that were found online to convert from the XML format to YOLO were helpful but did not match the requirements of the current implementation.

    Appropriate labels were required to be generated for both training images as well as validation images. The test images had no labels provided as those were intended for submission in the Kaggle competition.

    Older versions of YOLO required the label files to be in a subdirectory of where the training image is located called labels.

        Training Image: /path/to/training_image/img_01.JPEG

Label: /path/to/training_image/labels/img_02.txt

Three issues arose while preparing the labels. The first issue was that the newer versions of YOLO required the labels to reside in the same directory as the image and not a subdirectory. The labels are to retain the same name as the image file but with a ".txt" extension instead of the image extension (e.g., "JPEG"). This was discovered after some training was completed and logs showed missing labels.

The second issue that arose was the folder structure for the "val" and "train" images were setup differently. While the "val" folder contained all the images directly, the "train" folder had subfolders, one folder for each of the 1000 classes, with the images contained beneath.

The third issue that arose was that not every image in the "train" set had an associated XML annotation/label. My research indicated that during training of the YOLO model it was acceptable to include non-labeled images of different objects as it can help the model generalize the results. What was not clear in my research was whether the unlabeled images should be of objects that were not being trained for detection.

To address the above 3 issues new data scripts were written to parse the XML, generate the appropriate "txt" label file in the correction location, and remove any images that do not have any associated labels.

Generating the label files involves loading the associated XML file, parsing out the object class and bounding box information of each class that is in the image, converting it to YOLO label format, and writing it alongside the associated image file.

Training Preparation and Configuration

The next step was to configure the associated parameters for training and validating the model. While there are a significant number of parameters available, there were several key parameters that were required to be altered for a custom dataset. Several variations were recommended online, some with only a few parameter changes, however through several training experiments the following items in the YOLO.cfg file was key to running a training session:

Parameter(s)	Notes
batch=64 subdivision=64	Batch represents the number of sample images to be processed in one batch, while subdivisions represent how many "mini_batches" are to be processed by the GPU at once. The GPU processes "mini_batch" samples at once and the weights will be updated for "batch" samples The formula is mini_batch = batch/subdivisions. It would be ideal to have a lower subdivision number however it consumes more GPU memory. The subdivision number should be increased to where there are no GPU memory issues.

width=512 Height=512	The width and height of the images to be processed. If the images are larger than this, they will be resized. Larger images provide better accuracy but also take up additional GPU memory. The parameter must be a multiple of 32.
learning_rate=.0013	The initial learning rate for the model during training
burn_in=1000	The number of batches over which the learning rate will slowly increase from 0 to the "learning_rate"
max_batches	Maximum number of batches before training stops. The recommended formula for this is max_batches=<num_classes*2000>. For our ImageNet training of 1000 classes, it would be 1,000 * 2,000 = 2,000,000 batches for training.
steps=5000,8000 scales=.1,.1	Steps indicate when the learning rate should be altered by the scales parameter. In this case at 5,000 iterations the learning rate will be multiplied by .1, and then at 8,000 iterations the learning rate will again be multiplied by .1. The recommendation is to set steps at 80% and 90% of max_batches
classes=<num_classes> filters=255	The YOLO model has 3 separate YOLO layers defined within its configuration. Each of these layers requires the classes be set to the number of classes for which you are training. Before each YOLO layer is a convolutional layer which requires it's filters be set to <(num_classes +5)*3>. For 1,000 classes the number of filters would be 3015.

The YOLO model also requires a "data" file with configuration information. Within this file are 5 parameters: classes, train, valid, names, backup.

- classes – number of classes on which to be trained. This does not auto-translate over to the YOLO layer configurations within the model and must be specified separately.
- train – the location of a file containing a list of all images the model is to be trained on. The images listed within this file should use the full path and not relative paths. Relative paths work in some instances but cause challenges in others.
- valid – the location of a file containing a list of all images the model will be using for validation and mAP calculations. As with the train file it is best to use full paths to identify the image locations
- backup – a directory where the training weights will be stored

The YOLO model I will be using for this project will be YOLOv4 full model which has 162 layers consisting of:

- 110 convolutional layers
- 3 max layers
- 21 routing layers
- 23 shortcut layers
- 2 upsample layers
- 3 YOLO layers

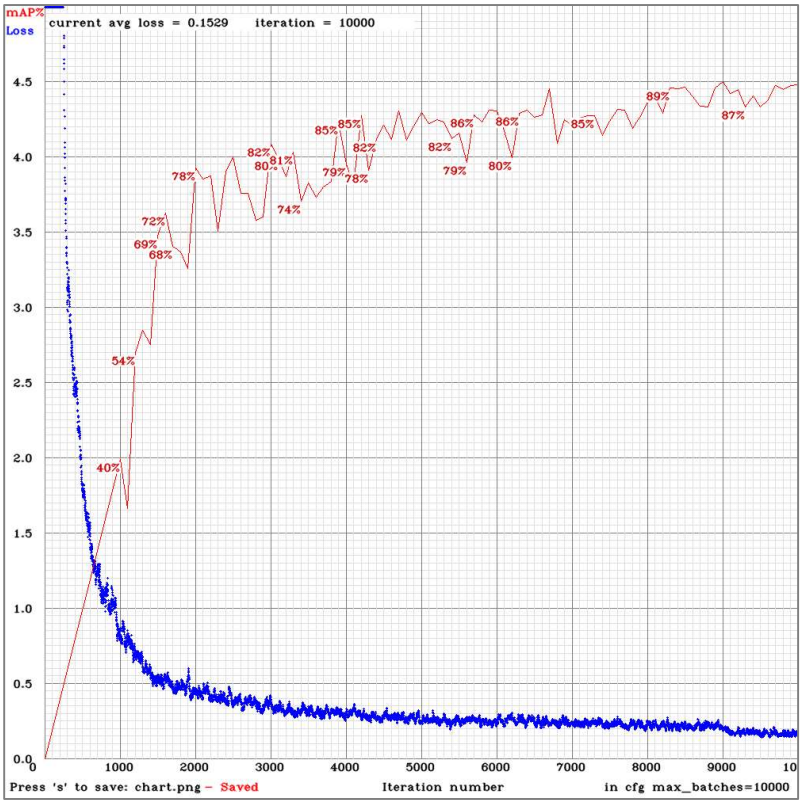
The base weights used have been trained on 500,500 iterations over 6.4 million images.

Training Rounds - Challenges and Findings

There were several rounds of training completed and each of them showed challenges, and changes that were required to move forward on the next round. The initial goal was to establish a baseline model which could predict, with a greater than 0% accuracy, the classes we are training to identify. During training a loss chart is produced with current avg loss as well as mAP statistics on the training data.

***The training parameters noted above such as burn_in, steps, and max_batches were learned at various stages throughout the experimentation. Some of my early experiments did not use burn_in, max_batches, or steps in the recommended way.

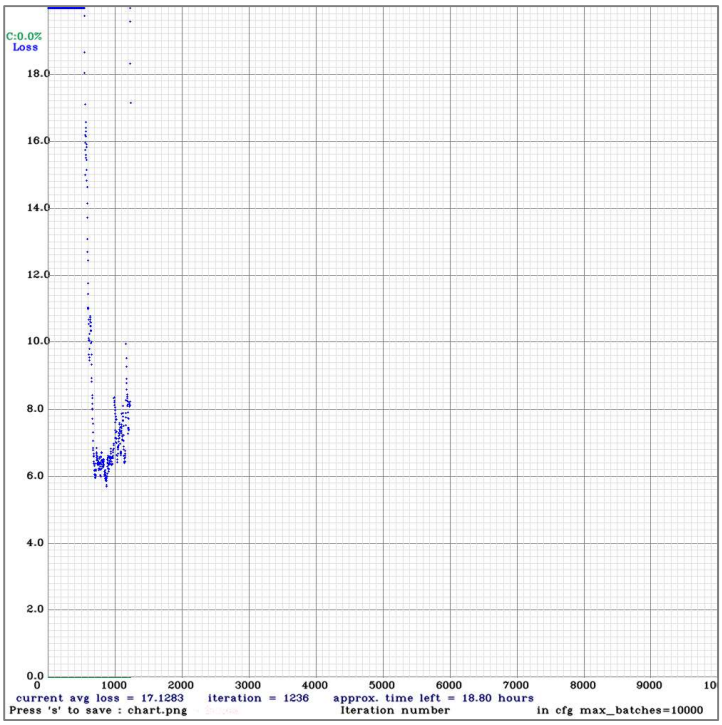
Below is an example training chart that was sourced from the internet and is not related to this project. The training loss (blue line) is desired to be as low as possible. It indicates how bad the model's prediction was on a single example. If the model's prediction is perfect, the loss is zero. As a rule, it seems that it is unlikely to get a loss below 0.06. The mAP is a measure of how accurately the object localization and detection is. The higher the mAP the better.



Experiment - Round 1

After a significant amount of online research and configuration tweaking, the first round of training was configured:

Parameter	Value
Batch	64
Subdivisions	64
Width	256
Height	256
Learning_rate	0.0013
Burn_in	1,000
Max_batches	10,000
Steps	400000, 450000
Scales	.1,.1



The experiment started successfully enough with an average loss of 1491.41. As desired the average loss decreased over training down to 6, but then fluctuated over several hundred iterations, rose to an average loss of 22.4, and then after 4.5 hours of training crashed. There were no indications what caused the crash or the fluctuations.

Note: At this stage I was not yet aware of the mAP chart option, so it was not included.

Experiment – Round 2

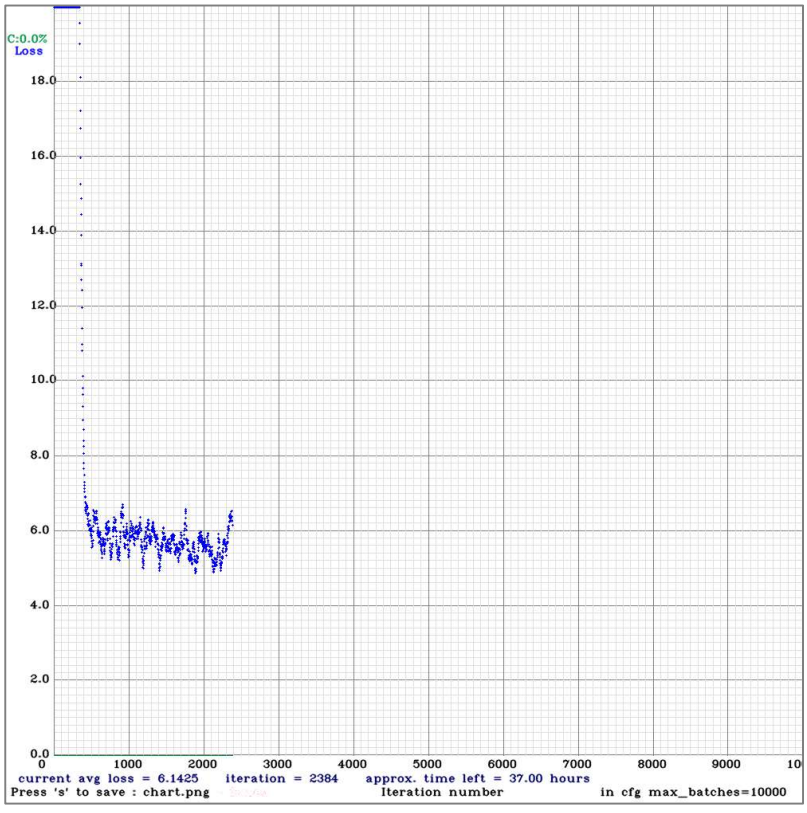
The improvement goals during this round of experimentation were:

1. Complete the training cycle
2. Reduce avg. loss
3. Remove wild fluctuations in avg. loss

During this round I learned about the burn_in rate, and the steps to adjust the learning_rate as training proceeded. The burn_in was lowered to 500 to make the learning rate reach it's maximum before 1000 iterations, the width/height was increased to provide a higher resolution for better accuracy, batch and subdivision were increased to improve training time, and the training rate was stepped at 600 and 2000 iterations. The width and height parameters were increased to 418 to improve resolution and hopefully training accuracy.

Parameter	Value
Batch	128
Subdivisions	128
Width	416
Height	416
Learning_rate	0.0013
Burn_in	500
Max_batches	10,000
Steps	600, 2000
Scales	.1,.1

Highlighted items changed since previous run



Again, the experiment started successfully with an initial avg loss of 1907 which dropped down to 6 by iteration 420. There was less subsequent fluctuation in the loss chart than the previous experiment. After approximately 8 hours and 2400 iterations the model training crashed. There was no indication as to the reason.

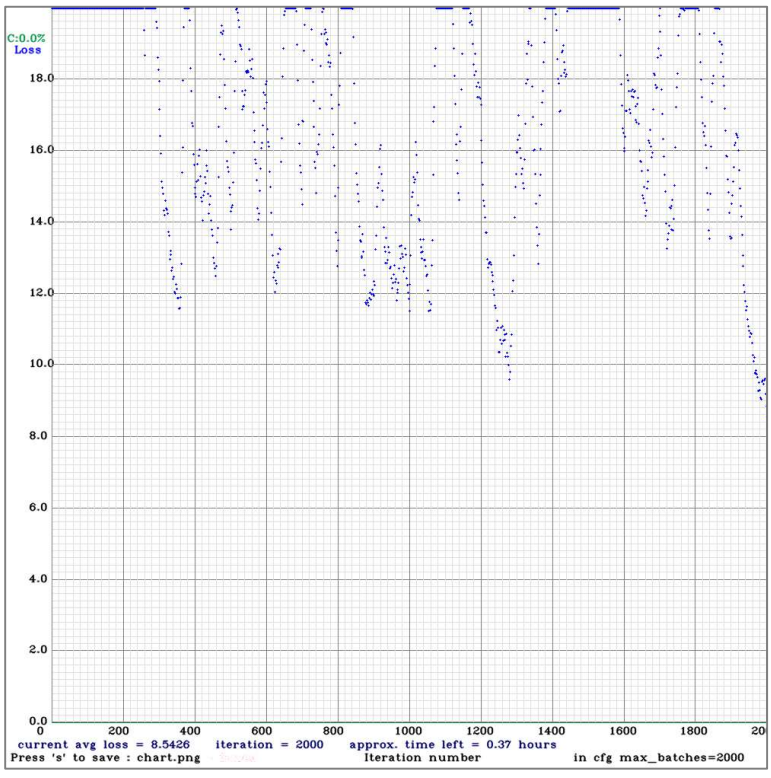
Note: At this stage I was not yet aware of the mAP chart option, so it was not included.

Experiment – Round 3

As of round 3 there had still not been a successful completion of training. This round was dedicated to ignoring accuracy and focusing on stabilizing the avg.loss chart as well as completing training. To that end, batch and subdivisions were reduced to 64, learning rate was reduced, burn in was reduced, max_batches was reduced to 2000, and the steps for learning rate reduction was changed to 300,400, and 600.

Parameter	Value
Batch	64
Subdivisions	64
Width	416
Height	416
Learning_rate	0.00007
Burn_in	150
Max_batches	2,000
Steps	300, 400, 600
Scales	.1,.1,.1

Highlighted items changed since previous run



A milestone was reached this round in that the training actually completed although the avg.loss chart for the training was the most chaotic of all experiments so far and gave little confidence that it had trained anything successfully.
The training started at an avg.loss of 1971, decreased to 12 at iteration 336, and then wildly fluctuated between an avg.loss of 22 and 9 before the training completed after 8 hours.
Using the final model weights, a test was run on the validation images. In every image, the model predicted that every item of the 1000 classes was present in the image with 100% confidence. Considering the extremely short training cycle, and wildly fluctuating avg. loss I would have expected that no objects would have been detected or predicted. Predicting every class object, in every image seemed to point out that something bigger was awry.

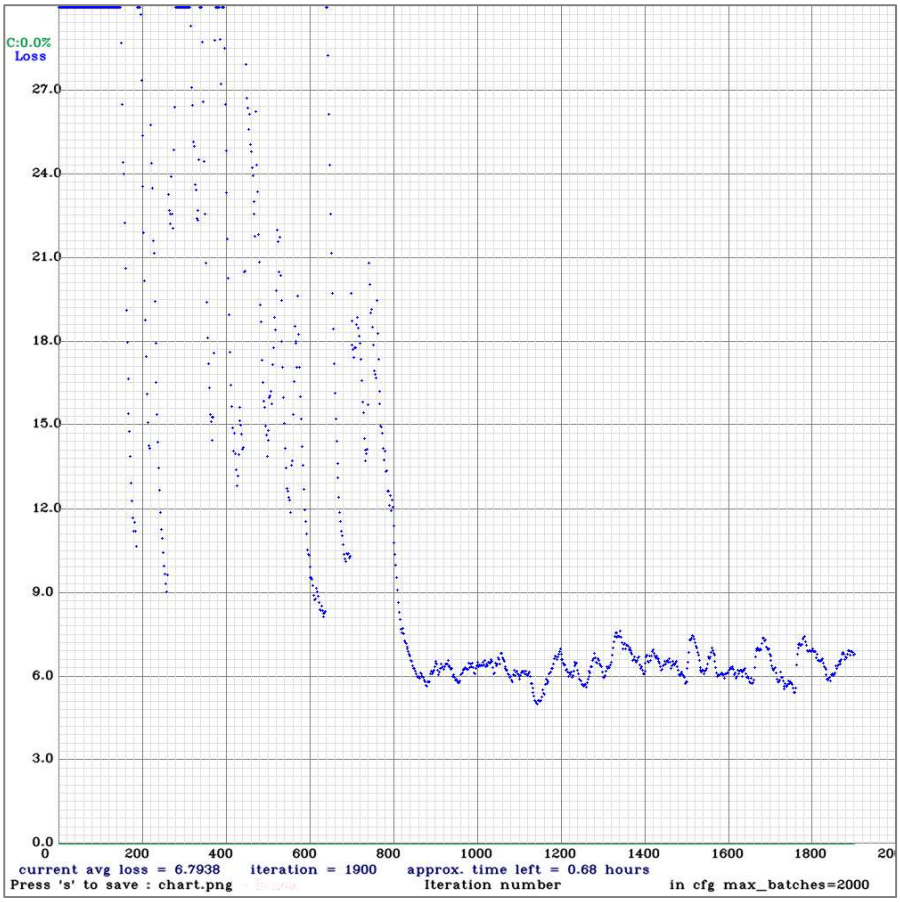
Note: At this stage I was not yet aware of the mAP chart option, so it was not included.

Experiment – Round 4

With the first training run completing successfully however with no ability to predict (everything is predicted at 100% for all classes/images), my assumption was that there was an issue with the extremely low learning rate used in the previous experiment. For this round, the learning rate was increased through every stage of the 2,000 iterations.
The expectation of training at this stage is to have a successful completion and be able to predict something with a greater than 0% accuracy.

Parameter	Value
Batch	64
Subdivisions	64
Width	416
Height	416
Learning_rate	0.0013
Burn_in	150
Max_batches	2,000
Steps	400, 600, 800, 1000
Scales	.5,.5,.1,.1

Highlighted items changed since previous run



This experiment started with an avg. loss of 2125, had some strong fluctuations of avg. loss between 9 and 26 from iteration 175 to 820 where it settled closer to an avg. loss of 6. After 8 hrs, the training did not complete successfully and crashed.

The only differentiation between a successful completion of the model in experiment 3 and this unsuccessful round was the learning rate changes. While I expected the learning rate to change the accuracy of the model prediction it was extremely curious that changing the learning rates caused the model to fail training.

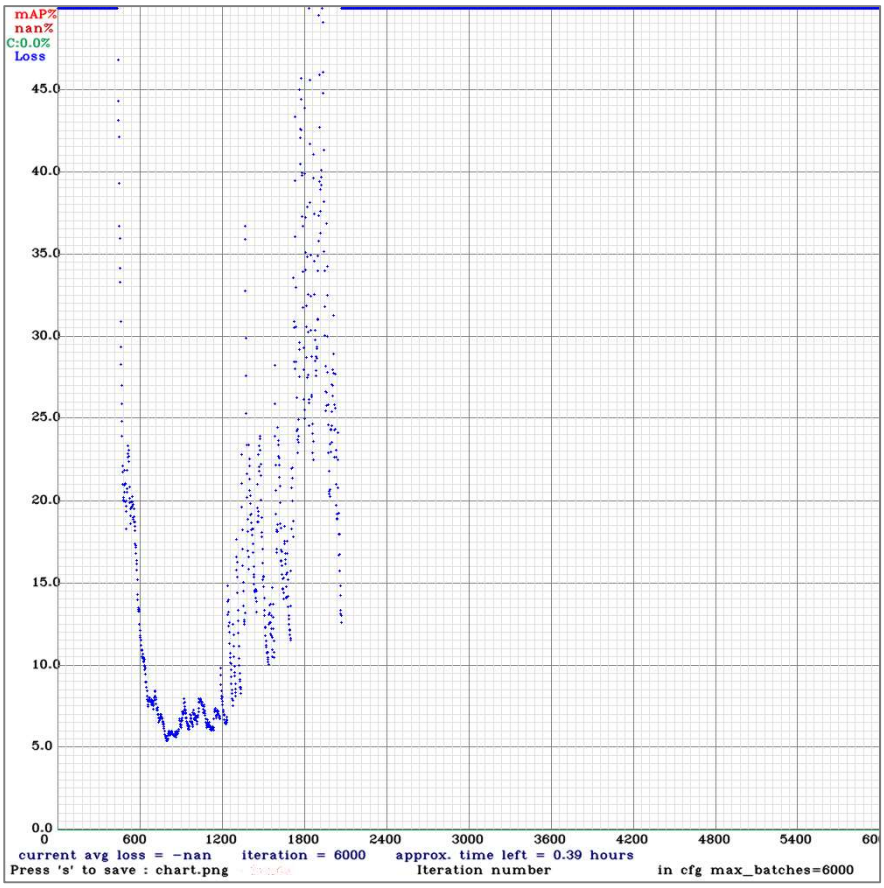
Experiment – Round 5

After additional research I found some of the parameter recommendations (steps, scales). The recommended parameters were set for those in this experiment. This round of experiments also introduced the mAP chart as an addition to the loss chart.

Experiment 3 completed training successfully, while experiment 4 did not complete and displayed some significant fluctuations in avg. loss in the early half of training. To smooth out the early variances, and hopefully successfully complete the training, the burn_in was increased to 900 and learning rate scales down by .1 at both 80% and 90% of the new max_batches of 6000.

Parameter	Value
Batch	64
Subdivisions	64
Width	416
Height	416
Learning_rate	0.0013
Burn_in	900
Max_batches	6,000
Steps	4800, 5400
Scales	.1,.1

Highlighted items changed since previous run



The training completed and did not crash. With only changes in the learning rates, we still see extreme fluctuations in the avg. loss over a short period of time. The training loss began at a similar level of 1934, lowered to 5 at 700 iterations, and fluctuated between 6 and 60 over a few hundred iterations. At iteration 2074, an error was encountered, and training values in the logs stop. Instead of avg. loss "-nan" values were displayed until training completion. While training completed it was certainly not completed successfully.

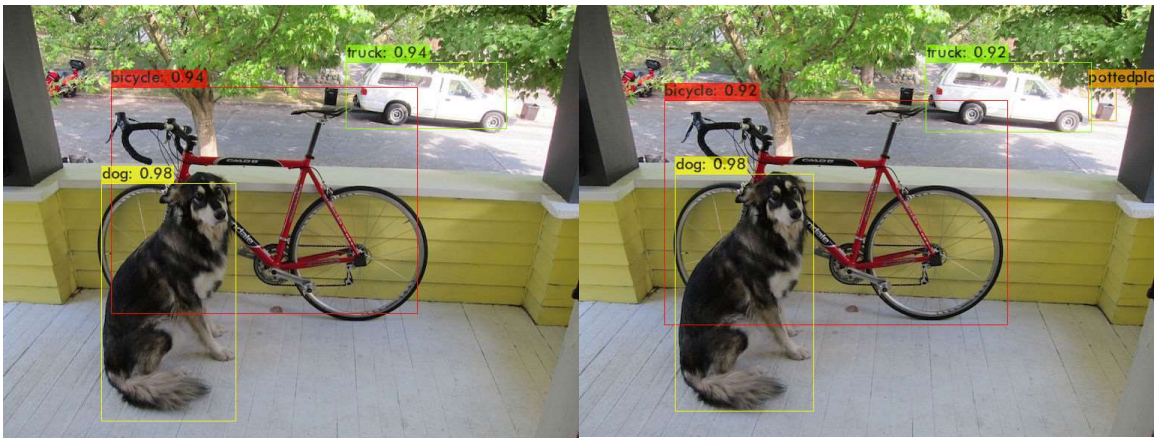
Although I would not call it successful, the training completed in that it did not crash. With the addition of the mAP chart, at the end of training a class summary is provided. In this case there were 0% detections across all classes in our validation images. With both low, and poor training results, 0% seems appropriate.

Considering that the only changes in this training are learning rates, it is again odd that the training does not consistently complete.

Experiment Interlude – Revelations (Round 6)

After running the first five experiments with varying results, none of which provided any predictive capability, I went back to ground zero and reinvestigated all configurations and options available.
I started by investigating the YOLO environment to determine if there are any configuration issues. I updated and rebuilt all required packages,

When performing a test with the same weight file but different YOLO configurations we received different results. This is of course understandable as the model configuration has a significant impact on results. Not only were detection percentages different, but one of the models detected a potted plant at 33% confidence that the other model did not detect.



After comparing the github repositories of pjreddie and AlexeyAB and researching numerous articles online I determined that one of the original configuration options noted in an online tutorial I utilized, had incorrectly paired a YOLO model with the wrong weights file. While the weight file should technically work, it can cause significant issues in both prediction and training.

With time for project completion quickly running to a close I ran numerous short experiments to determine a new and viable configuration for training the Kaggle dataset.

Experiment – Round 7

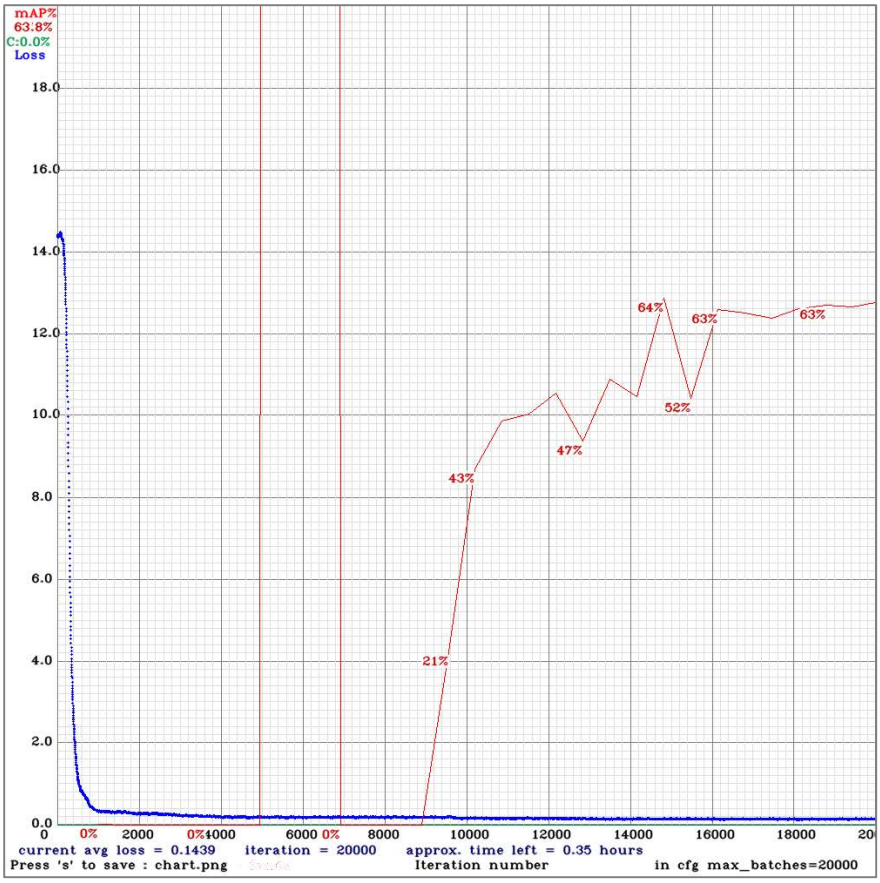
This is the final round of experiments and includes several changes including weights and configuration files. There are a significant number of YOLO configuration and pre-trained weight files, and while there may be an optimal one to use for this dataset, I decided to use the full YOLO v4 configuration to get some predictive capability from the trained model.

The original project goal was to train a model to identify 1000 different items in an image with associated bounding boxes. The recommended number of batches to train 1000 classes for detection is (num_classes * 2000), which would be 2,000,000 iterations. To shorten the time and arrive at a viable predictive model, I chose a subset of 10 classes and their associated validation images and trained a model based on that. The goal is that this should provide a baseline understanding of when predictive capability in the model will be generated based on number of iterations of training.

The number of training images available per class different significantly, from less than 300 to 1200. The 10 chosen classes were those that had the most training images available.
Note: As the goal was to train the model to detect 10 classes, the top 10 classes were likely not excellent choices as a base. The top 10 classes included multiple dogs, as well as abstract items like “seashore”, and “confectionary, candy bar”. While this is viable, a better test would have been to choose items that were obviously and distinctly different such as: dogs, birds, telephone, car.

To achieve this goal, an entire new set of scripts, data extraction, and configuration had to be performed to generate the new image repository with associated labels.

Parameter	Value
Batch	64
Subdivisions	64
Width	416
Height	416
Learning_rate	0.0013
Burn_in	1000
Max_batches	20,000
Steps	16000, 18000
Scales	.1,.1


mAP%
63.8%
C:0.0%
Loss
18.0
16.0
14.0
12.0
10.0
8.0
6.0
4.0
2.0
0.0
0
2000
4000
6000
8000
10000
12000
14000
16000
18000
0%
0%
0%
21%
43%
47%
52%
64%
63%
63%
current avg loss = 0.1439 iteration = 20000 approx. time left = 0.35 hours
Press 's' to save : chart.png
Iteration number
in cfg max_batches=20000


n02106166 Border collie: 0.75

Detection	Class
91.33%	boxer
88.43%	Border collie
83.37%	Doberman, Doberman pinscher
74.77%	standard poodle
65.68%	maze, labyrinth
61.88%	stone wall
61.47%	trailer truck, tractor trailer
52.15%	pay-phone, pay-station
47.35%	seashore, coast, seacoast
11.86%	confectionery, confectionary, candy store

Precision	Recall	F1-Score	Avg.IoU	mAP
0.88	0.47	0.61	0.70	0.64

After 42 hrs not only did the model complete it's training but was able to predict and localize objects as per our original project goals. The average loss started much lower in this round of training with 14.2, decreased to an average loss of 1 by 500 iterations, and continued to decline reaching an average loss of 0.14 over 20,000 iterations.

The base model and weights were trained on 6.4 million images over 500,000 iterations. This experiment added an additional 20,000 iterations (additional 4%) training 2.5 million additional images. The results seem quite promising and indicate that with significant additional training the 1000 images would be detected. I configured a similar training session and started it to determine how long it would take on the full dataset. The initial training indicated that it would take 8,537 hours or 356 days of training.

Conclusions and Next Steps

Although many hurdles have been overcome, the goal of generating a model to classify 1000 items was not fully completed. The base model at detecting 10 different classes was successful but there is opportunity to be improved through additional training as well as training on image variations (rotations, cropping, etc)

There is a great deal of potential using the YOLO algorithm for image detection. It has an integration with OpenCV which allows it to link to video for real-time object detection from webcam.

While one may conceptually understand that training a large-scale Machine Learning model is time consuming, it is certainly a revelatory experience to have walked through it using 1000 classes and more than 100GB of images. Using 10 of the 1000 (1%) of the image classes, and over 40 hours of training provided a workable model. While still not a large model by any means, training the entire 1000 classes would take approximately 8530 hours (356 days) on the current hardware configuration.

References

Hardware/VM/CUDA

i. Nvidia Developer Zone. CUDA Toolkit Documentation: https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html#pre-installation-actions
ii. Nvidia Accelerated Computing. NVIDIA CUDNN documentation: https://docs.nvidia.com/deeplearning/cudnn/install-guide/index.html
iii. Wendell. Fedora 33: Ultimate VFIO Guide: https://forum.level1techs.com/t/fedora-33-ultimiate-vfio-guide-for-2020-2021-wip/163814
iv. Ekistece. Github.com. Fedora 33 VFIO Guide. https://github.com/ekistece/Fedora-33-VFIO-guide

YOLO/Darknet

i. Kaggle.com. Image Net Object Localization Challenge: https://www.kaggle.com/c/imagenet-object-localization-challenge/overview
ii. AlexeyAB. Github.com. Darknet YOLO package: https://github.com/AlexeyAB/darknet
iii. Mingweihe. Github.com. Kaggle trial on ImageNet. https://github.com/mingweihe/ImageNet
iv. Train Darknet on Custom Dataset. RobotocsKnowledgeBase.com: https://roboticsknowledgebase.com/wiki/machine-learning/train-darknet-on-custom-dataset/
v. S.Yohanandan. TowardsDataScience.com. mAP(Mean Average Precision) might confuse you!: https://towardsdatascience.com/map-mean-average-precision-might-confuse-you-5956f1bfa9e2
vi. G.Karimi. Section.io. Introduction to YOLO Algorithm for Object Detection. https://www.section.io/engineering-education/introduction-to-yolo-algorithm-for-object-detection/#:~:text=YOLO%20is%20an%20algorithm%20that,%2C%20parking%20meters%2C%20and%20animals.

vii. Livebook.manning.com. 7 Object Detection with R-CNN, SSD, and YOLO: https://livebook.manning.com/book/grokking-deep-learning-for-computer-vision/chapter-7/v-6/36

V. Subramanyam. Medium.com. Basics of Bounding Boxes: https://medium.com/analytics-vidhya/basics-of-bounding-boxes-94e583b5e16c

A.Rosebrock. pyimagesearch.com. Multi-class object detection and bounding box regression with Keras, TensorFlow, and Deep Learning: https://www.pyimagesearch.com/2020/10/12/multi-class-object-detection-and-bounding-box-regression-with-keras-tensorflow-and-deep-learning/

Github.com. Support thread. https://github.com/pjreddie/darknet/issues/2263

Github.com. Support thread. https://github.com/pjreddie/darknet/issues/723

Github.com. Support thread. https://github.com/pjreddie/darknet/issues/903