Lab-3 Part2 Report

Lab Report

Group Members:

- Carolyn Cui 100399399
- Haolin Li 218600849
- Robert Palermo

Introduction

In our exploration of object detection models, we aim to determine the performance of existing popular models and, in addition, compare the process of single-node deep learning to distributed deep learning.

The topic that best suits our needs in this foray into object detection models is stop signs our motivations for choosing this topic is that stop sign detection, along with general street sign detection, is still a key challenge that any fully automatic or otherwise semi-automated car has to overcome to be viable for the market.

There are nearly 700,000 accidents that occur around stop signs each year in the U.S., of which more than 80% can be prevented. Numerous solutions do exist already, but performance is also continuously improving. Reasonably, because of scrutiny and safety concerns, these models must be rigorously tested and trained, then fine-tuned. The rollout of AI-assisted technology in cars proves to be beneficial not only for day-to-day drivers but also for disabled, elderly, and otherwise impaired drivers. We inch ever closer toward fully automated rider experiences as these models improve. As such, we decided that this topic is doable given the time limit, familiar to our group, and allows us to experience firsthand the challenges accompanying an object detection task.

Due to familiarity and ease of use, we have decided to use the YOLO models, known for their accuracy and real-time performance. This makes them optimal for tasks that require high accuracy and speed, such as our case: detecting stop signs.

Dataset Curation

The dataset was created last year from photos taken around UC Merced, the Bay Area, and San Diego. Additional images are screenshots courtesy of Google Street View. Totaling around 70 images, our dataset includes nighttime photos, stop signs at varying distances, and other street signs (e.g., do not enter). Roboflow was used to annotate the dataset.

We acknowledge that, for object detection, the dataset is on the smaller end. Though the optimal dataset size depends on the nature of the data and how accurate we want the model to be, 150 images are preferred for a more usable model. We opted for a lower number due to time and resource constraints.

Single-Node

For all single-node training, we decided to use the YOLO models. The original YOLO, and all versions through YOLOv3, were created by Joseph Redmon and based on the darknet neural network. Alexey Bochkovskiy continued his work in YOLOv4, which boasts higher accuracy but is not necessarily faster.

Branching off of there, we chose YOLOv5 due to its existing documentation for both single-node and distributed training, and the version also means it’s suitable for comparison to YOLOv4.

All training was conducted through Google Colab with a provided Tesla T4 GPU.

YOLOv4

Overview

As this initial application of YOLOv4 was one of our first times doing anything related to AI/ML, the approach and evaluation could be better, and some metrics may need to be included.

Training

Though a model was provided, we made adjustments to suit our needs better. Our model can run up to 2000 iterations, with some predetermined values as required by Darknet, such as a 0.001 learning rate and 64 batches. We settled on 32 subdivisions, meaning two images are evaluated simultaneously.

We split training into eight sessions, stopping every 100 iterations to save the weights. This ensured we could roll back to an earlier version in the case of potential overfitting.

Environment Setup: Darknet & Dependencies

We choose to use the YOLOv4 framework, which is implemented by Darknet

```
git clone https://github.com/AlexeyAB/darknet
# @title Change Config of darknet
# change configs

cd darknet
sed -i 's/OPENCV=0/OPENCV=1/' Makefile
sed -i 's/GPU=0/GPU=1/' Makefile
sed -i 's/CUDNN=0/CUDNN=1/' Makefile
sed -i 's/CUDNN_HALF=0/CUDNN_HALF=1/' Makefile
sed -i 's/LIBSO=0/LIBSO=1/' Makefile

make
```

Importing Dataset

```
# @title Copying Datasets and Configuration Files

# Copying datasets into /data

!cp -r /content/gdrive/MyDrive/TSG/stopsigns/dataset/train -d data/
!cp -r /content/gdrive/MyDrive/TSG/stopsigns/dataset/test -d data/

# Copying config file

!cp /content/gdrive/MyDrive/TSG/stopsigns/yolov4-obj.cfg ./cfg

# Copying object data

!cp /content/gdrive/MyDrive/TSG/stopsigns/obj.names ./data
!cp /content/gdrive/MyDrive/TSG/stopsigns/obj.data ./data
```

Download Pretrained weights

```
!wget https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v3_optimal/yolov4.conv.137
```

Training

```
!./darknet detector train data/obj.data cfg/yolov4-obj.cfg yolov4.conv.137 -dont_show -map 2>&1 > log.txt
```

Evaluation

The model was trained for 2000 iterations, which took approximately 12 hours. It was intentionally trained for a while longer than needed to determine the best performance possible and the point at which we would begin to see overfitting. After evaluating the performance at 1900 and 2000, we decided to stick with 1900 for our final model.

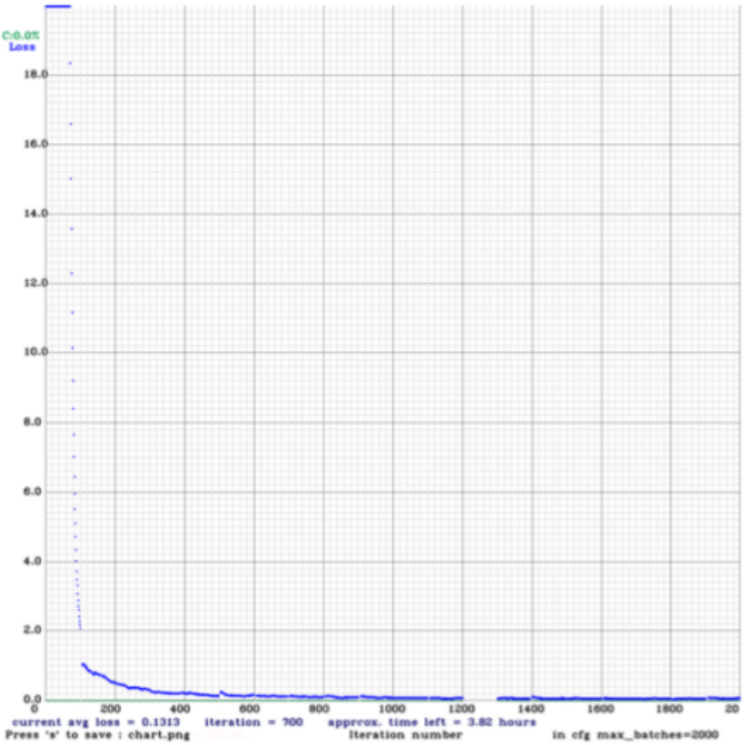


Figure 1. YOLOv4 Loss Chart

Our loss dropped rapidly within the first 100 iterations and gradually decreased. There was a minimal decrease starting around 800-1000 iterations. Overall, our accuracy or mean average precision (mAP) peaked at around 90% and an intersection over union (IoU) of around 79%.

At 1900 iterations, we had a final mAP of 87.88% and a final loss of 0.0633. As you can see from the chart, our loss spiked at 2000 iterations, which is why we decided to roll back the model to 1900.

With most street signs, stop signs were usually correctly detected. Early on, we faced some difficulties with many non-stop signs, but this was eventually relegated to only “do not enter” signs. We will discuss specifics in detail below.

Detection result with output weights:

Improvements and Difficulties

Once again, as this was our first time working with any sort of AI/ML model, we can make numerous improvements.

The first, and most costly improvement in terms of time, is adding more training data, especially null data. We did collect more nulls to add to the dataset if needed. But as mentioned above, we attempted to keep the dataset smaller, knowing the tradeoff would be in performance. Training an image detector is far more intensive than just training on data, and given the tight window in the original project as well, it was not in our best interest to pull from large, existing datasets.

As mentioned above, a significant later issue was the model believing “do not enter” signs were stop signs. To solve this, we added null data and resumed training, which improved its recognition. However, when we fed a demo video into the network, we discovered it also thought various yellow signs, such as pedestrian signs, were stop signs. Once we got close enough, the model correctly recognized that they were not stop signs. With more data and training, this was eventually mostly solved, though problems with “do not enter” signs remained at low enough detection thresholds. There was a significant difference between a 50% and 75% threshold. However, increasing this threshold also means real stop signs might not be detected in time. By extension, the detection threshold can also be fine-tuned to minimize this sort of error.

Furthermore, though YOLOv4 allows for greater model customization, due to our unfamiliarity with the configuration at the time, we could not fully configure the model to match our needs. Some parameters in the configuration were arbitrarily chosen, while other parameters were finalized after a lot of guessing and checking to ensure that the model was actually working. With a deeper understanding of machine learning concepts, we will be able to actually fine-tune similar models in the future.

YOLOv5

Implemented in PyTorch, YOLOv5 is designed to be faster than YOLOv4 while maintaining competitive accuracy. It focuses on real-time applications and is optimized for efficiency.

Training Steps

Environment Setup: YOLOv5 & Dependencies

We choose to use the YOLOv5 framework, which is implemented by PyTorch

```
git clone https://github.com/ultralytics/yolov5
cd yolov5
git reset --hard fbe67e465375231474a2ad80a4389efc77ecff99
```

```
# install dependencies as necessary
!pip install -qr requirements.txt  # install dependencies (ignore errors)
import torch
from IPython.display import Image, clear_output  # to display images
from utils.downloads import attempt_download  # to download models/datasets
```

Importing Dataset

For dataset, we utilized the tagged resources provided by Rainbow Flow, which offers high-quality and sufficient data.

```
...
project = rf.workspace("team-intense").project("stop-signs-cb6tq")
dataset = project.version(5).download("yolov5")
```

Adjusting Model Configuration

```
writetemplate /content/yolov5/models/custom_yolov5s.yaml
```

```
%%writetemplate /content/yolov5/models/custom_yolov5s.yaml

# parameters
nc: {num_classes}  # number of classes
depth_multiple: 0.33  # model depth multiple
width_multiple: 0.50  # layer channel multiple

# anchors
anchors:
  - [10,13, 16,30, 33,23]  # P3/8
  - [30,61, 62,45, 59,119]  # P4/16
  - [116,90, 156,198, 373,326]  # P5/32
```

```
# YOLOv5 backbone
backbone:
  # [from, number, module, args]
  [[-1, 1, Focus, [64, 3]],  # 0-P1/2
   [-1, 1, Conv, [128, 3, 2]],  # 1-P2/4
   [-1, 3, BottleneckCSP, [128]],
   [-1, 1, Conv, [256, 3, 2]],  # 3-P3/8
   [-1, 9, BottleneckCSP, [256]],
   [-1, 1, Conv, [512, 3, 2]],  # 5-P4/16
   [-1, 9, BottleneckCSP, [512]],
   [-1, 1, Conv, [1024, 3, 2]],  # 7-P5/32
   [-1, 1, SPP, [1024, [5, 9, 13]]],
   [-1, 3, BottleneckCSP, [1024, False]],  # 9
  ]

# YOLOv5 head
head:
  [[-1, 1, Conv, [512, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [[-1, 6], 1, Concat, [1]],  # cat backbone P4
   [-1, 3, BottleneckCSP, [512, False]],  # 13

   [-1, 1, Conv, [256, 1, 1]],
   [-1, 1, nn.Upsample, [None, 2, 'nearest']],
   [[-1, 4], 1, Concat, [1]],  # cat backbone P3
   [-1, 3, BottleneckCSP, [256, False]],  # 17 (P3/8-small)

   [-1, 1, Conv, [256, 3, 2]],
   [[-1, 14], 1, Concat, [1]],  # cat head P4
   [-1, 3, BottleneckCSP, [512, False]],  # 20 (P4/16-medium)

   [-1, 1, Conv, [512, 3, 2]],
   [[-1, 10], 1, Concat, [1]],  # cat head P5
   [-1, 3, BottleneckCSP, [1024, False]],  # 23 (P5/32-large)

   [[17, 20, 23], 1, Detect, [nc, anchors]],  # Detect(P3, P4, P5)
  ]
```

Download Pretrained weights

Use the official pretrained weights for YOLOv5s

```
%cd /content/yolov5/
!wget https://github.com/ultralytics/yolov5/releases/download/v7.0/yolov5s.pt
```

Training

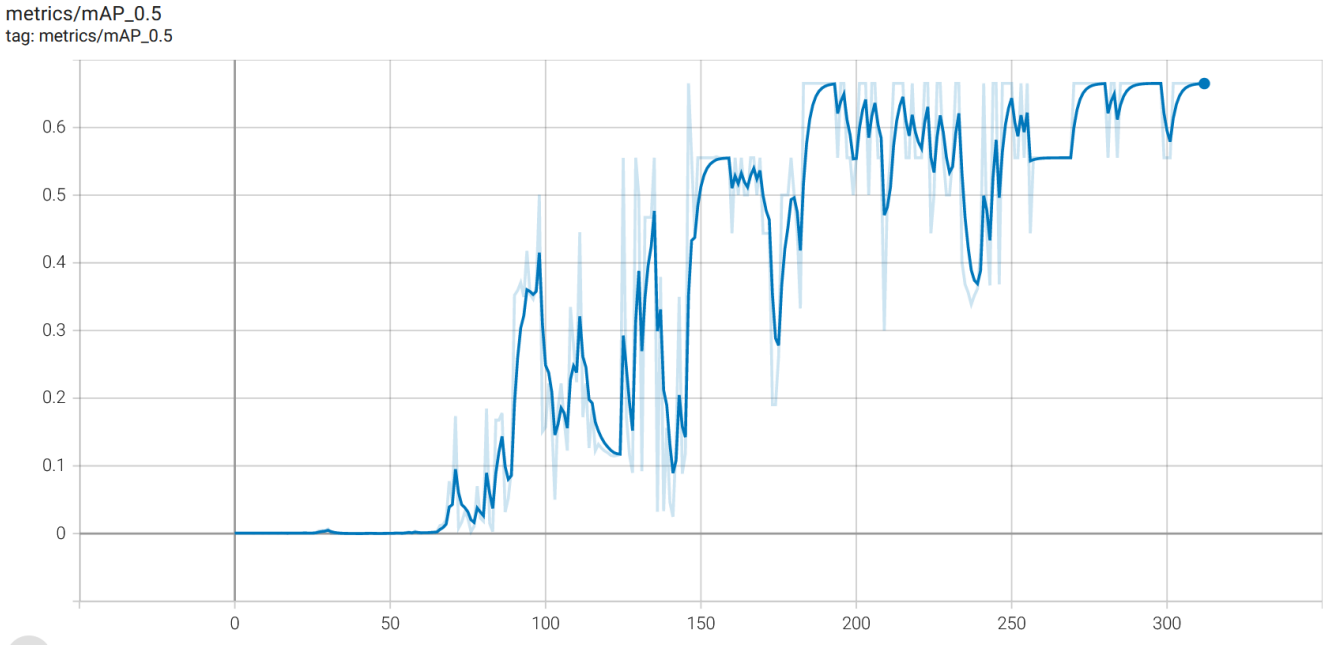
By calling the `train.py` script, we can have a simple training with 100 epochs

```
python train.py --img 416 --batch 16 --epochs 100 --data {dataset.location}/data.yaml --cfg
./models/custom_yolov5s.yaml --weights '' --name yolov5s_results  --cache
```

Evaluation

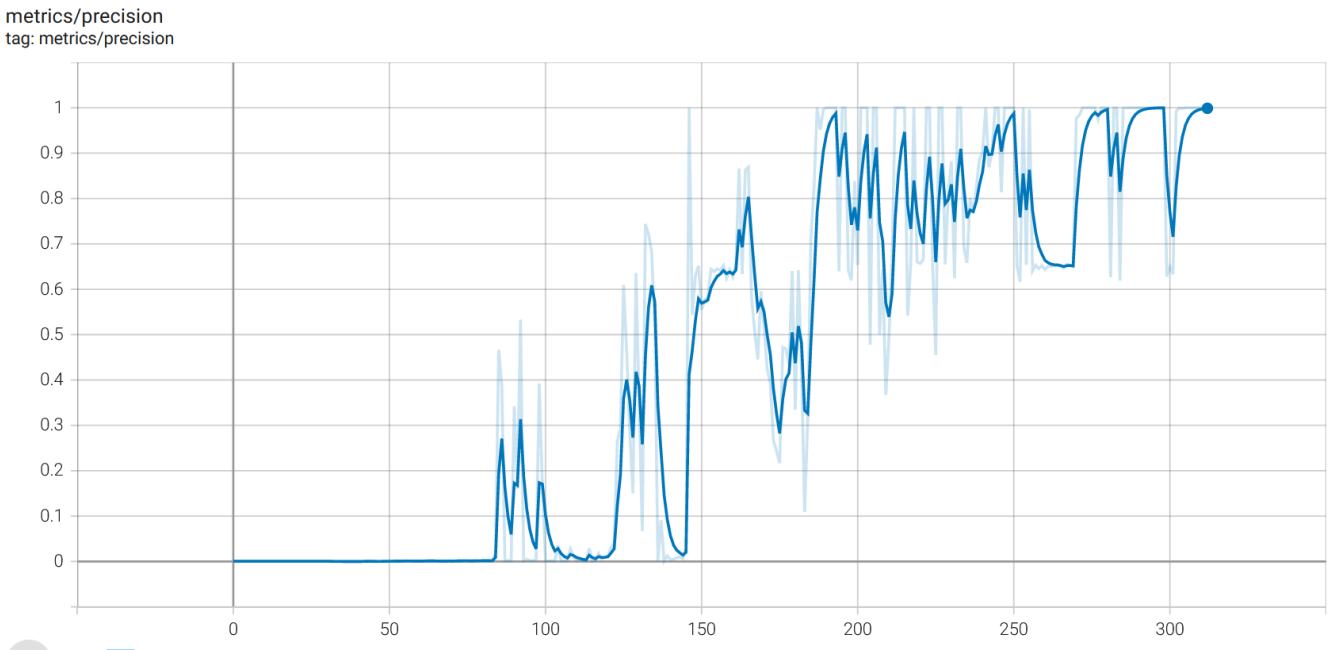
Training losses and performance metrics are saved to Tensorboard and also to a logfile defined above with the **--name** flag when we train. In our case, we named this `yolov5s_results`. The results file is plotted as a png after training completes.

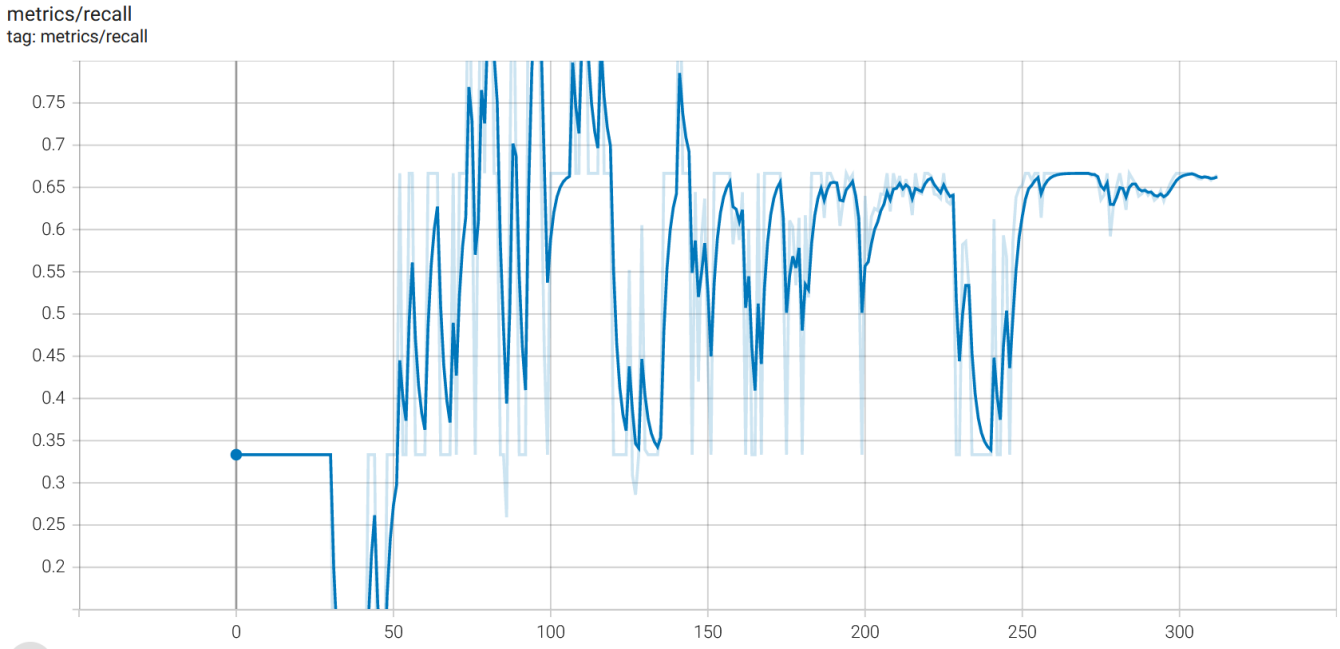
Mean Average Precision (mAP)



The framework monitored the Mean Average Precision (mAP) at different intersection-over-union (IoU) thresholds. The mAP at an IoU of 0.5 **showed a notable improvement over the training epochs**, indicating that our model's ability to match the predicted bounding boxes with the ground truth is improving.

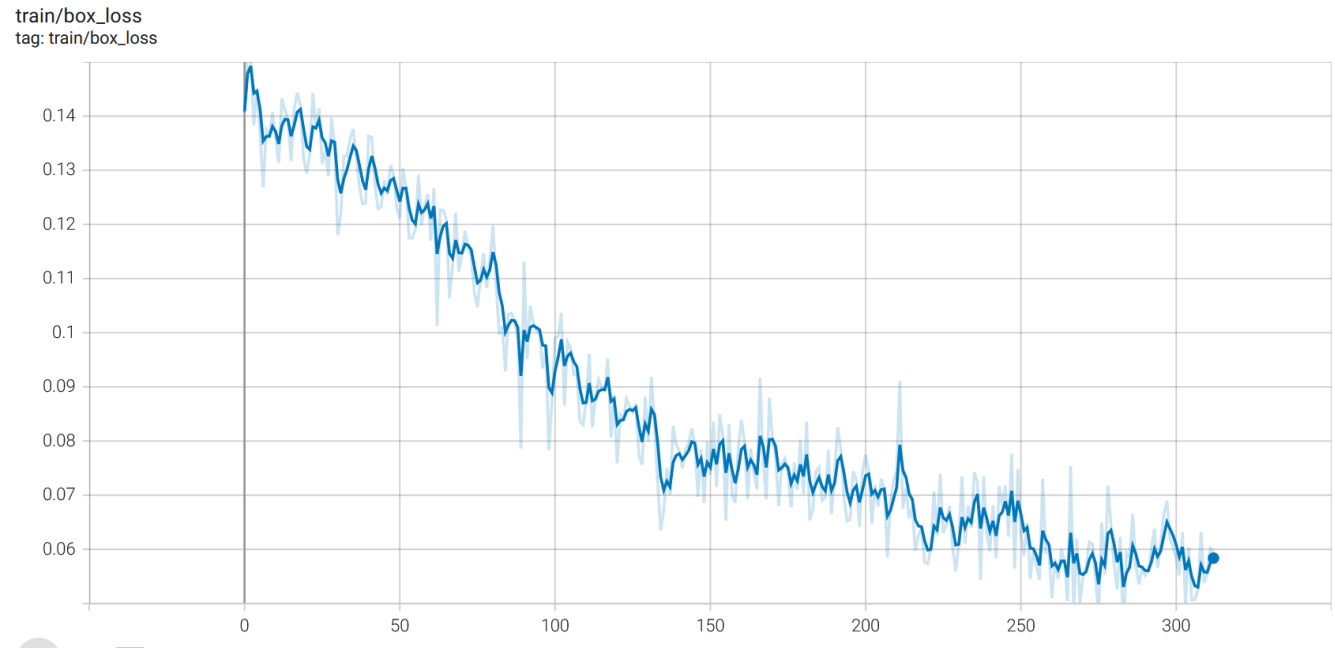
Precision and Recall

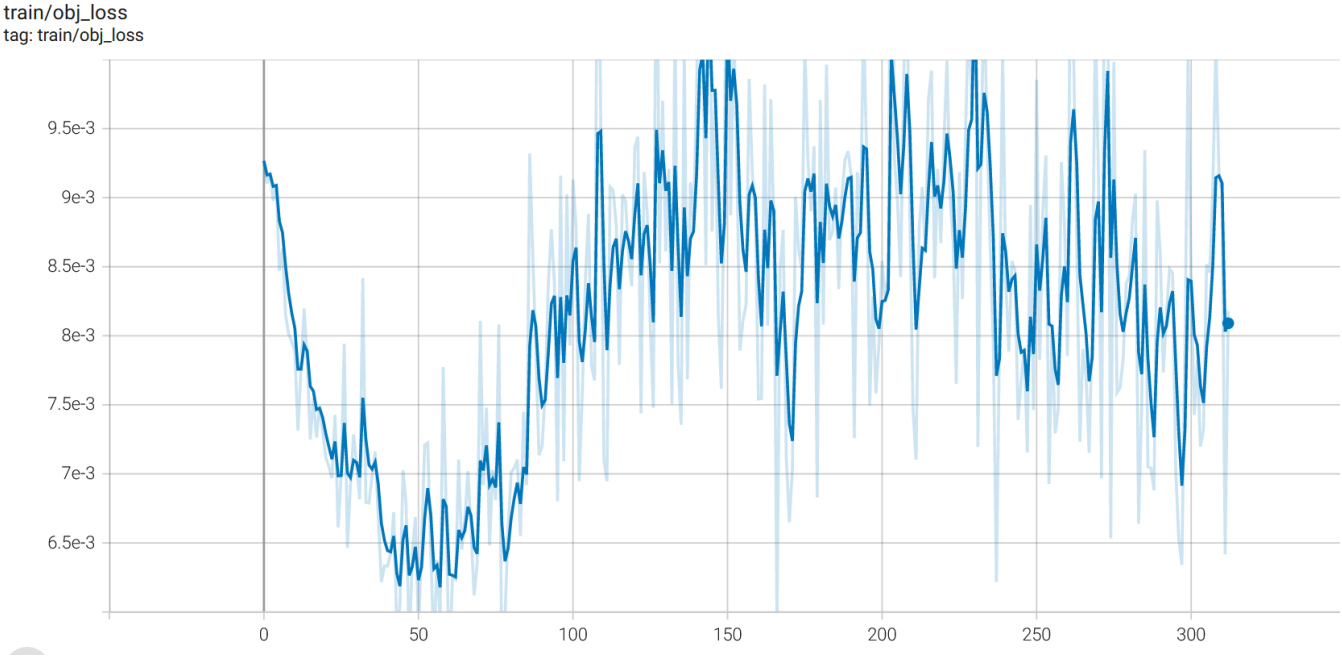




Precision metrics over the course of training depicted an upward trend, suggesting that the proportion of positive identifications that were correct is increasing. Similarly, the recall metrics showed that the model's ability to find all the relevant instances of objects within the dataset improved as the training progressed. The balance between precision and recall is crucial, especially in scenarios where either false positives or false negatives have significant consequences.

Loss Metrics

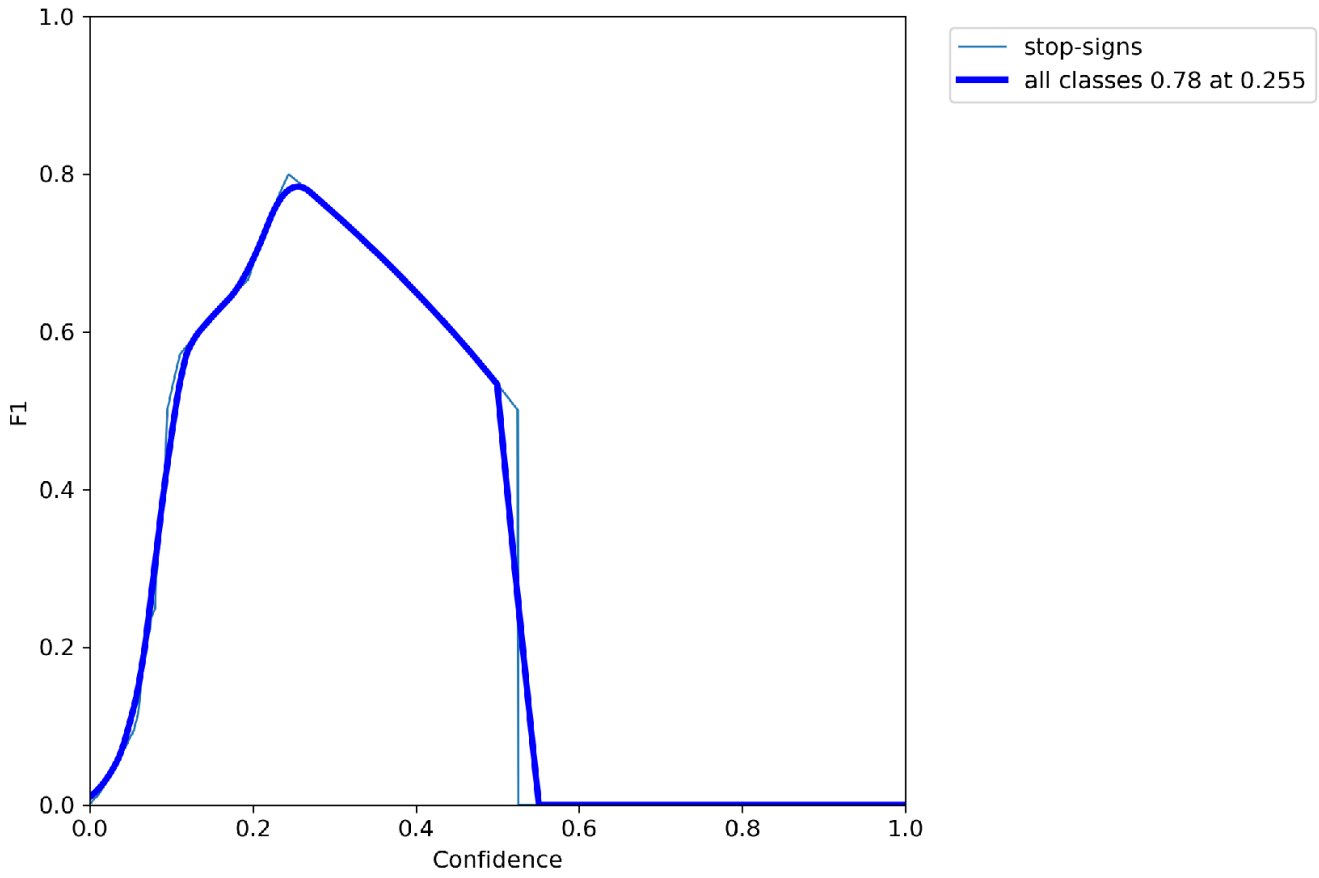




In terms of loss metrics, the bounding box loss (`train/box_loss`) decreased steadily, which is indicative of the model's increasing precision in predicting the location and scale of the bounding boxes around the objects. The objectness loss (`train/obj_loss`), which reflects the model's confidence in the presence of an object within the box, also showed a decreasing trend. This is a positive sign that the model is learning to discriminate between objects and background effectively.

However, the classification loss (`train/cls_loss`) remained unexpectedly flat and low throughout training. While low classification loss is typically a positive indicator, the lack of fluctuation might point to an issue with data labeling or an anomaly in the training process that warrants further investigation.

F1 Score



The F1 score curve revealed the model's best balance between precision and recall occurred at a specific confidence threshold. The highest point on the curve indicated the optimal threshold where the model achieves its best performance in terms of the harmonic

mean of precision and recall.

Overall, the evaluation metrics indicate a model that is learning and improving in its detection capabilities.

Generally, the scale of our dataset is not large enough. As indicates by the loss metrics, the risk of over fitting exits. To further improve this problem, potential methods include:

- Increase the amount of data: Try collecting more data or use data augmentation techniques to expand the training set.
- Adjust model complexity: For small-scale data sets, you can try using a simpler model architecture.
- Early stopping method: Stop training when performance on the validation set no longer improves.
- Optimize hyperparameters: adjust learning rate, training batch size, etc.

Distributed Training

Due to time constraints, we weren't able to complete distributed training and compare to a single node. However, we do have thoughts regarding how to achieve

For single node with multiple GPUs

YOLOv5 already supports multi-GPU training, which means that some of the settings for distributed training **have been implemented in the framework**. Here are some basic steps:

1. **Environment configuration**: Make sure training nodes have the same version of PyTorch, YOLOv5 and other related dependencies installed.
2. **Modify training script**:
   - Use `torch.distributed.launch` or `torch.distributed.run` to launch the training script. These tools help you launch training in multiple processes, one for each GPU.
   - In training scripts, use `torch.nn.parallel.DistributedDataParallel` instead of `torch.nn.DataParallel`. This is a class in PyTorch for model parallelism across multiple processes.
   As a modify sample, the training instruction should belike:

```
python -m torch.distributed.run --nproc_per_node 2 train.py --batch 64 --data coco.yaml --weights yolov5s.pt --device 0,1
```

4. **Start distributed training**: Use the appropriate command to start distributed training.This may include setting environment variables such as the node's address and port.

For multiple nodes

To deploy the training on different nodes, we need to configure and execute distributed training for YOLOv5 across multiple nodes using SLURM.

1. **Environment Setup:**
   Ensure uniformity across all computing nodes in terms of software environments, including identical versions of Python, PyTorch, CUDA, and NCCL.
2. **Code Preparation for Distributed Training:**

- Modify the YOLOv5 training script to utilize `torch.nn.parallel.DistributedDataParallel` instead of `torch.nn.DataParallel`.
- Implement `torch.utils.data.distributed.DistributedSampler` for dataset management, ensuring each training process accesses a unique data subset.
- Define environment variables like `MASTER_ADDR`, `MASTER_PORT`, `WORLD_SIZE`, and `RANK`, necessary for PyTorch's distributed training setup.

3. **SLURM Job Script Creation:**

- Develop a SLURM job script specifying the required resources (e.g., number of GPUs, memory per node, and runtime).
- The script should employ the `srun` command to initiate the distributed training process. An example snippet is as follows:

```
#!/bin/bash
#SBATCH --job-name=distributed_training
```

```
#SBATCH --partition=gpu

# 2 nodes
#SBATCH --nodes=2

# two GPUs
#SBATCH --gres=gpu:2

# I have set the following to some random values
#SBATCH --ntasks=8                      # Number of MPI ranks
#SBATCH --cpus-per-task=8               # Number of cores per MPI rank
#SBATCH --ntasks-per-node=4             # How many tasks on each node
#SBATCH --ntasks-per-socket=2           # How many tasks on each CPU or socket
#SBATCH --mem-per-cpu=100mb             # Memory per core

# set to 16 but should be changed to match our specifications
#OR SWAP mem per cpu for #SBATCH --mem=16G

# maximum run time (set to 1 hour just because)
#SBATCH --time=01:00:00

# loading modules
module load cuda/11.0
module load cudnn/8.0.2

# same as the bash script from here
# replace with ipynb OR download the python script from colab and replace
training="train.py"
# or uncomment to test if this works
# jupyter nbconvert --to script Train_YOLOv5_ipynb.ipynb
# checking existence
if [ -f "$training" ];
then
    echo "running"
    # in the example I based this off of, they used mpi (message passing interface) + srun is used for slurm
    srun --mpi=pmix_v3 python3 "$training"
    echo "done"
else
    echo "does not exist"
fi
```

In nutshell, SLURM enables the leveraging of YOLOv5's capabilities on a larger scale, enhancing the efficiency and scope of object detection tasks.

Related Concept & Discussion

SLURM

In order to allocate the necessary resources for distributed training on a cluster, we will need to use **SLURM**, or Simple Linux Utility for Resource Management. SLURM is a job scheduler and resource manager widely used in high-performance computing (HPC) and its purpose is to efficiently allocate said resources (e.g. nodes, CPUs, GPUs, memory, time).

The SLURM framework provides us with this framework for parallelism. Jobs are submitted through a SLURM script (.sbatch file), wherein the user requests the necessary resources, specifies modules and dependencies, etc. SLURM then manages the allocation of resources and schedules the jobs for execution.

To further understand SLURM, we must also familiarize ourselves with some of the terminology associated with the framework. A job, as mentioned above, can be understood as some unit of work submitted to SLURM and includes tasks. The environment in which we will be working in is also made up of nodes, which are some computational resource, e.g. a server, that is part of the cluster. A cluster, then, is is just a group of interconnected nodes that work together to perform said tasks. Lastly, a task is a sub-process within a job, and tasks can be executed simultaneously across multiple nodes.

In order to actually run these tasks, the command lies in the SLURM script: srun. Running this command in a manner such as in the code snippet below is what we need to actually submit the job, after which resource allocation, etc. also begins. For parallelization purposes, we can utilize systems like message passing interface (MPI) in tandem with SLURM.

NCCL

NCCL, which stands for NVIDIA Collective Communications Library, is a critical component in the realm of distributed training, particularly in deep learning applications. Its role is to optimize and facilitate the communication processes between GPUs, especially when they are distributed across multiple nodes.

In distributed training, there are multiple methods that can be applied, e.g. data, model, and hybrid parallelism. However, such the implementation of such parallelism requires frequent and efficient communication between GPUs to synchronize data. This is where NCCL comes into play. It provides highly optimized implementations of collective communication operations such as all-reduce, all-gather, broadcast, and reduce. These operations are essential for aggregating data and updating model parameters consistently across all GPUs.

NCCL specifically is great for high-bandwidth and low-latency communication, which is vital in reducing training time. It is designed to work effectively across various network architectures and NVIDIA GPU architectures. Hence, it can be integrated seamlessly into various distributed deep learning frameworks like TensorFlow, PyTorch, and MXNet. For example, YOLOv5 has built-in support.

Scalability is another key feature; NCCL can efficiently manage communication in small-scale training environments with a few GPUs, such as the training we would have done for this lab, and also scale up to handle more complex multi-node tasks.

In summary, NCCL is great for distributed training, especially in the avenue of optimizing GPU-to-GPU communication, enabling faster and more efficient training of deep learning models across multiple nodes. Its high performance, flexibility, and scalability make it an excellent tool.