DETR: END-TO-END OBJECT DETECTION WITH TRANSFORMERS

Whitepaper

W251 Final ● 07 August 2020

By Sirak Ghebremusse

Project Repo: http://www.github.com/sirakzg/detr

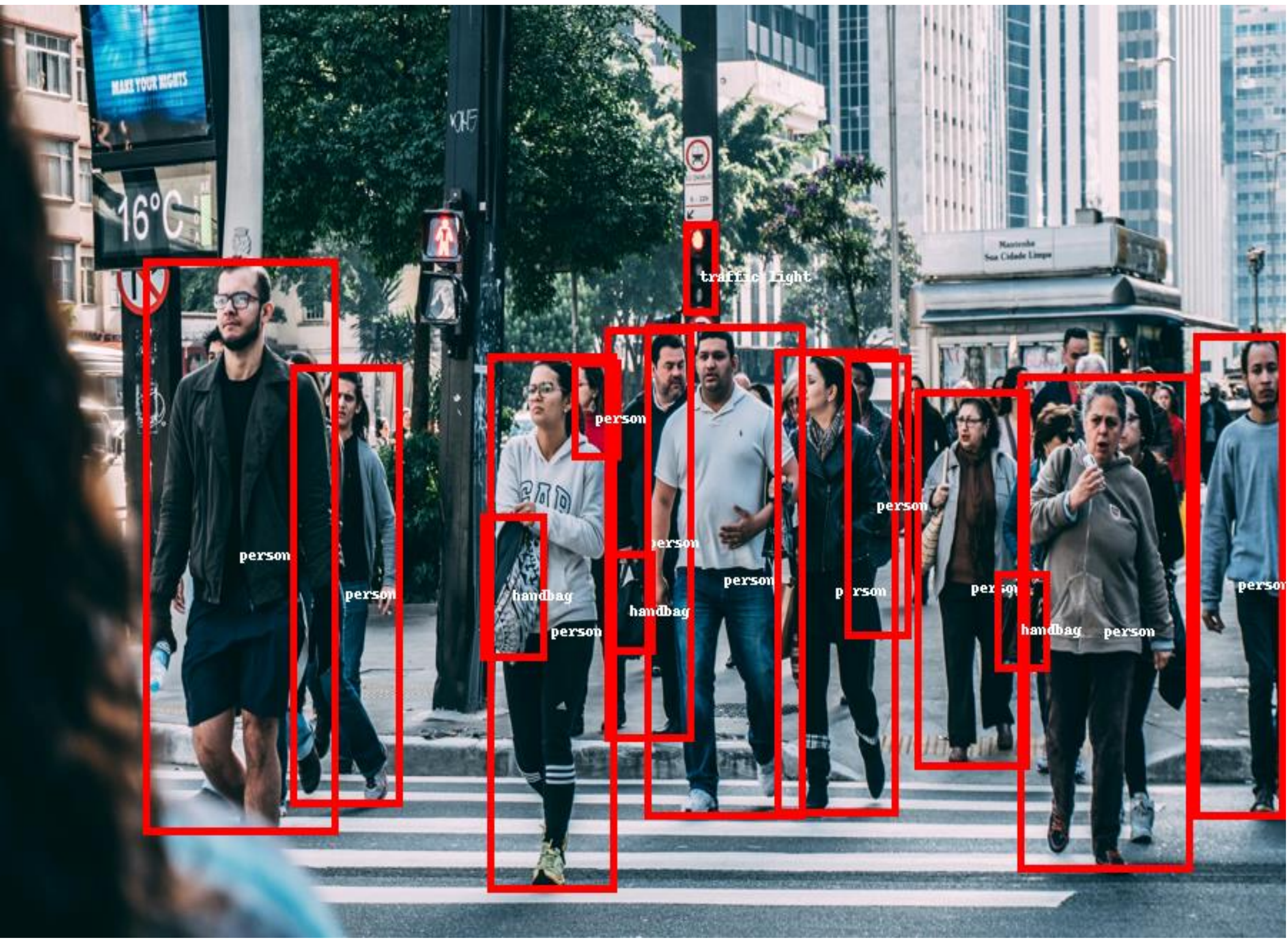


Table of Contents

Introduction

In May of 2020 Facebook AI published a paper entitled 'End-to-End Object Detection with Transformers', one of the first research papers to demonstrate the use of transformer networks to the task of computer vision. Transformers are still a relatively new architecture for neural networks, introduced in 2017 and mainly focused on Natural Language processing.

Even with their radically new approach to computer vision tasks like object detection and instance segmentation, Facebook was still able to show that this model was able to compete with state of the art approaches like Yolo and Faster R-CNN, and requires significantly less post processing code.

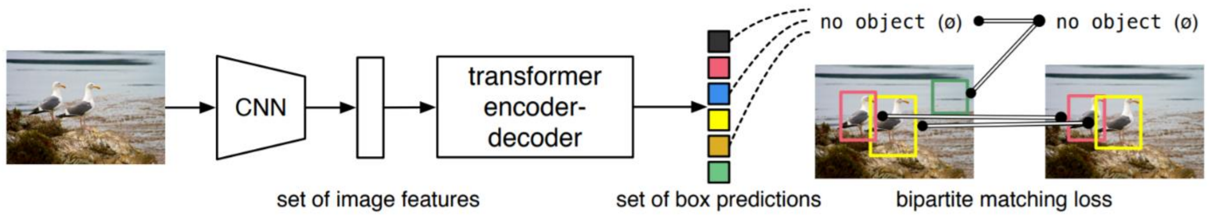


Fig. 1: DETR directly predicts (in parallel) the final set of detections by combining a common CNN with a transformer architecture. During training, bipartite matching uniquely assigns predictions with ground truth boxes. Prediction with no match should yield a "`no object`" (∅) class prediction.

Challenge

As a final project in the W251 Deep Learning course I set out to 'productionize' the DETR model from Facebook's research state into something that can run on an edge device. I selected the Nvidia Jetson Xavier NX specifically for its low power consumption as well also for its support for deep learning acceleration via Tensor Cores.

The task breakdown are as follows:

- Modify the training code to support mixed precision training.
- Adapt the CNN back bone to an edge friendly architecture (akin to TinyYolo).
- Depoly this trained model onto the Xavier NX using TensorRT.

Tools

Several tools and requirements were used in this project to bring everything together. Below is an itemized list of the main tools and their usage:

Tool	Usage
PyTorch	The deep learning SDK used to train and evaluate the model
TensorRT	An optimized library from Nvidia for deep learning inference
OpenCV	Used for grabbing images from a webcam to pass to the model.
Docker	As a method of providing containerized environments for running scripts
Nvidia Apex	A library run with PyTorch to support mixed precision training
Jetson Xavier NX	Our low-power deep learning device
IBM/AWS Cloud	Training was performed on large multi-GPU servers in the cloud

Dataset

The COCO 2017 dataset was used for this project to ensure similar results to the Facebook AI's research. COCO is one of the largest, commercial friendly machine learning datasets that is sponsored by Facebook, Microsoft and others. It is comprised of 18 GB of image data just for the training set, and 7 GB for validation and testing.

To assist with the download and setup of this large dataset a tool from the GluonCV toolkit was containerized into its own Docker image. The script required a couple dependencies to run so both *gluoncv* and *mxnet* packages were installed in the docker image. The */data/* folder of the cloud server was also passed to the docker image as a volume and the docker image creates the */data/coco* subfolder to pass along to the script. Once run this docker image created the three required folders for the COCO dataset: train, validation and test.

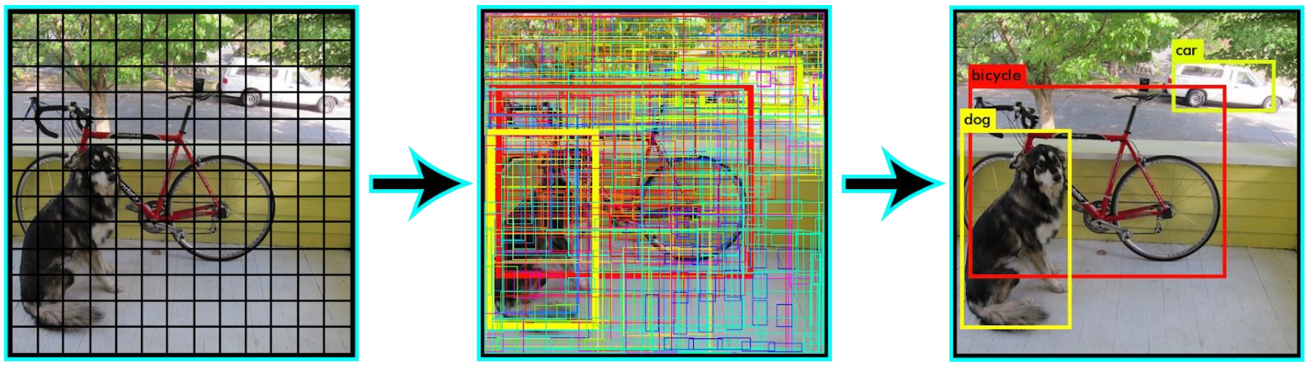
Running the Docker.mscoco container:

```
> docker build -t mscoco -f Dockerfile.mscoco .

> docker run --net=host -v /data:/data mscoco
```

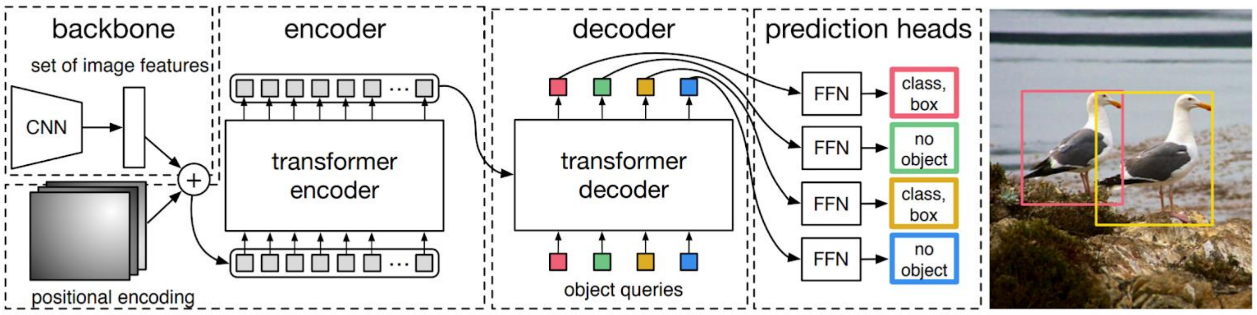
DETR vs Yolo

Yolo has become the industry standard for object detection models. It is a computer vision model that can detect several objects contained within an image and place both a bounding box around each object as well as provide object classes for each box.

It accomplishes regional detection by dividing the image into 19x19 sections (see left most image below) and providing a classification per box. It is then up to the data scientist to parse through this model's output to display only the relevant bounding boxes.



DETR differs in that there is little post processing code required in order to display its model outputs. By employing the use of a transformer encoder/decoder block, DETR instead is trained to output a <u>*set*</u> of bounding boxes and also supports a null classification if fewer objects are detected in the image than requested.



Attention for Detection

Diving deeper into the transformer structure we can see how the layers are constructed for both the encoder and decoder layers:

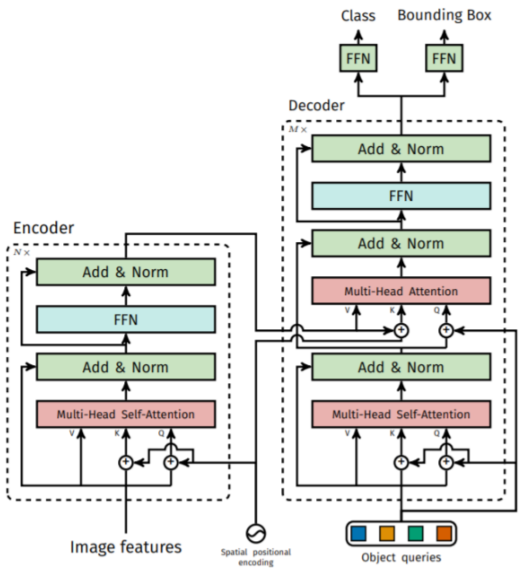


Fig. 10: Architecture of DETR's transformer.

What is interesting to note is the hard coded size **N** of the object queries permitted. In their project Facebook trained with an ***N****=100* on the COCO dataset as they observed no image containing more than 70-80 objects. They also outline in their paper how each object query once trained appears to develop a specific "skillset", or favouring bounding boxes of a certain size and region.

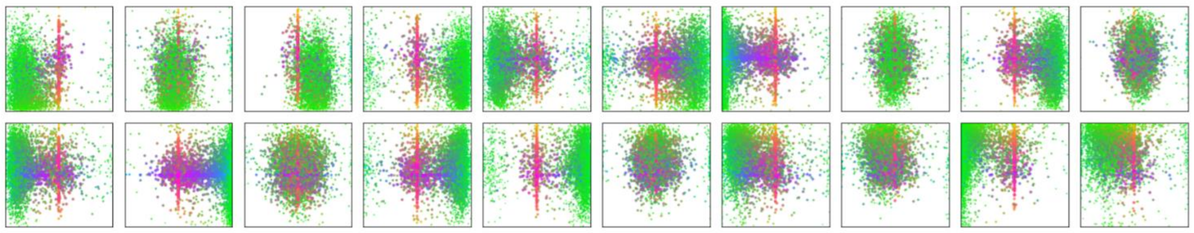


Fig. 7: Visualization of all box predictions on all images from COCO 2017 val set for 20 out of total $N = 100$ prediction slots in DETR decoder. Each box prediction is represented as a point with the coordinates of its center in the 1-by-1 square normalized by each image size. The points are color-coded so that green color corresponds to small boxes, red to large horizontal boxes and blue to large vertical boxes. We observe that each slot learns to specialize on certain areas and box sizes with several operating modes. We note that almost all slots have a mode of predicting large image-wide boxes that are common in COCO dataset.

Attention for Segmentation

Once the model has been trained for object detection an optional step that can be performed in a separate, secondary training process is the ability to render instance (or panoptic) segmentations of an input image.

In the following figure the authors of the paper illustrate how the same outputs from the transformer decoder can be passed instead to a CNN head, performing convolutional operations on attention maps to generate separate masks for each detected object. These can then be combined back into a single image using pixel-wise argmax to determine the class each pixel belongs to.

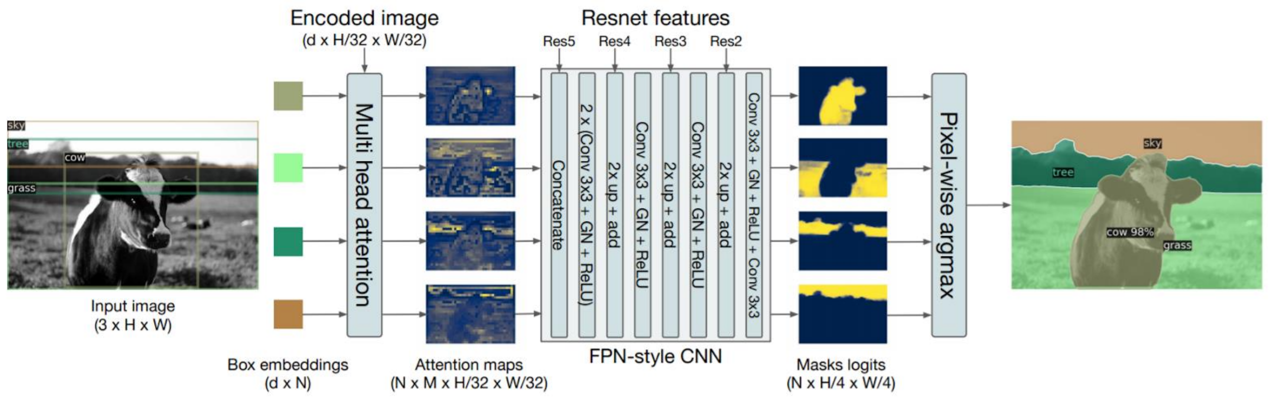


Fig. 8: Illustration of the panoptic head. A binary mask is generated in parallel for each detected object, then the masks are merged using pixel-wise argmax.

Mixed Precision: Training

In order to perform mixed precision training on a cloud VM I resorted to using a server instance on the AWS cloud platform. Initial attempts on IBM Cloud virtual machines did not have the proper driver support installed for the version of PyTorch required for this project.

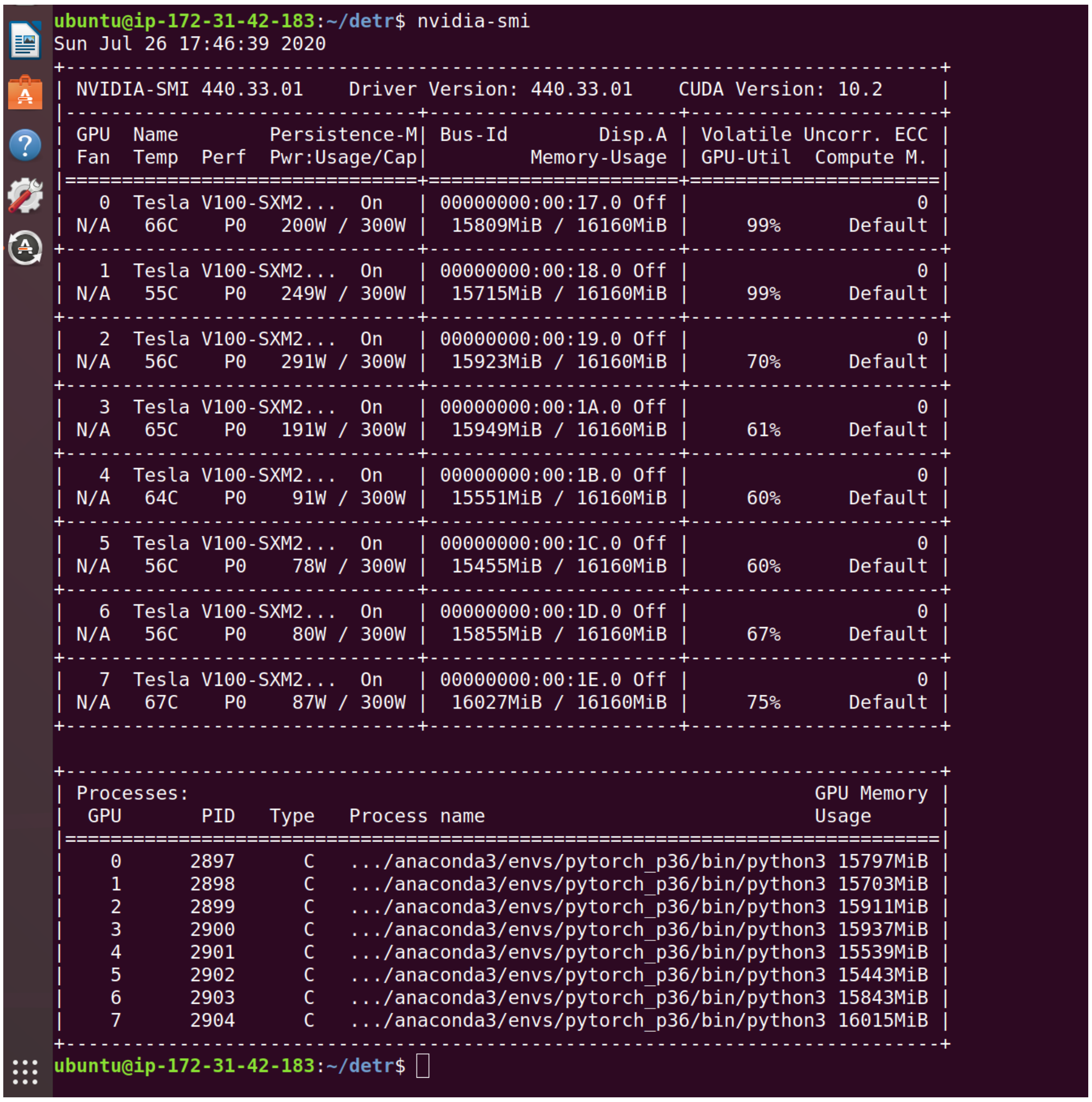
Amazon's EC2 provides some prebuilt machine images with deep learning support, which I selected to ensure a compatible environment. Once running you are presented with several Conda environments to select, including the selected Python 3 and Pytorch version. The downside of this setup is that more of the disk space on the instance has already been taken up to include all these deep learning packages and environments, so much so that during my first attempt I ran out of hard drive space downloading and expanding the COCO dataset.

I also wanted to recreate the training environment that the original researchers performed their training in and requested an 8 V100 GPU server instance. While the dedicated pricing for these servers is $27 an hour their spot instance pricing is much more reasonable at ~$7/hr. My budget allowed for only 48 hours of training time, translating to 100 epochs of the 300 Facebook managed to perform.

Training commands on AWS:

```
> source activate pytorch_p36

> git clone https://github.com/sirakzg/detr.git

> docker build -t detr -f Dockerfile .

> docker run --runtime=nvidia --net=host -v /data:/data -v
/home/ubuntu/detr:/workspace/detr -ti detr bash

> time python3 -m torch.distributed.launch --nproc_per_node=8  --use_env
main.py --epochs 100 --coco_path /data/coco/
```

GPU utilization was terribly low with default training settings, and I managed to increase it by doubling the batch size. It is still not optimal due to memory limits of the V100 GPUs used from AWS.



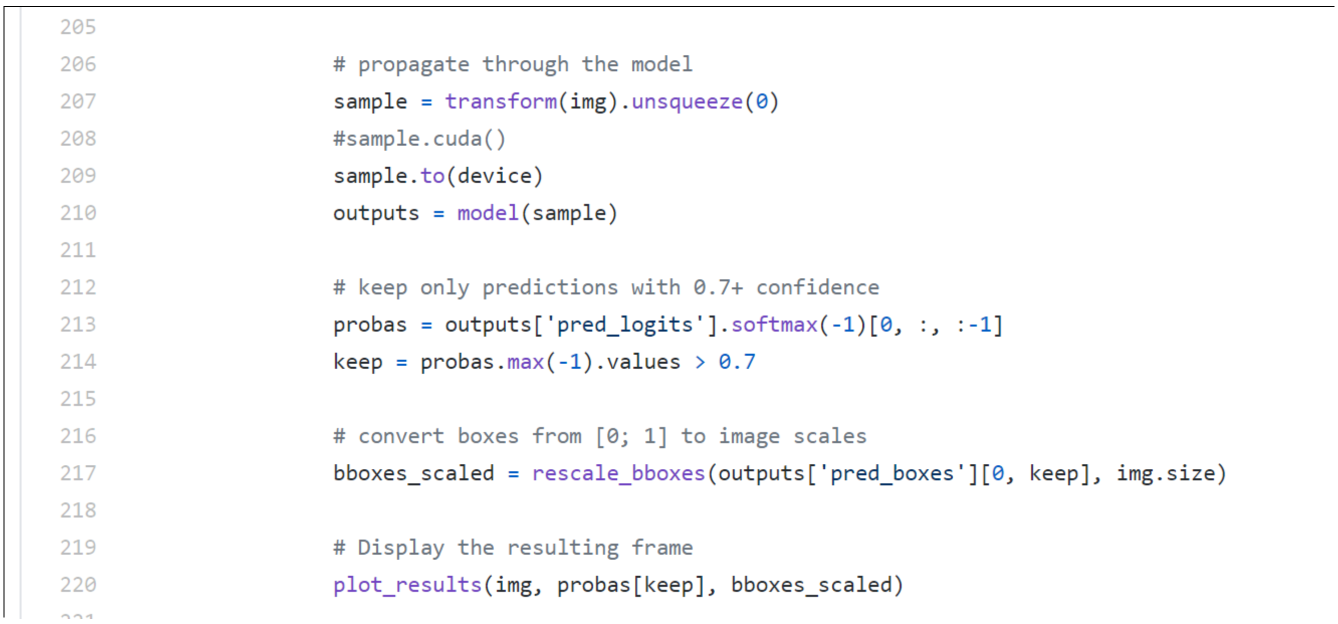
```
ubuntu@ip-172-31-42-183:~/detr$ nvidia-smi
Sun Jul 26 17:46:39 2020
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.33.01    Driver Version: 440.33.01    CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  On   | 00000000:00:17.0 Off |                    0 |
| N/A   66C    P0   200W / 300W |  15809MiB / 16160MiB |     99%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla V100-SXM2...  On   | 00000000:00:18.0 Off |                    0 |
| N/A   55C    P0   249W / 300W |  15715MiB / 16160MiB |     99%      Default |
+-------------------------------+----------------------+----------------------+
|   2  Tesla V100-SXM2...  On   | 00000000:00:19.0 Off |                    0 |
| N/A   56C    P0   291W / 300W |  15923MiB / 16160MiB |     70%      Default |
+-------------------------------+----------------------+----------------------+
|   3  Tesla V100-SXM2...  On   | 00000000:00:1A.0 Off |                    0 |
| N/A   65C    P0   191W / 300W |  15949MiB / 16160MiB |     61%      Default |
+-------------------------------+----------------------+----------------------+
|   4  Tesla V100-SXM2...  On   | 00000000:00:1B.0 Off |                    0 |
| N/A   64C    P0    91W / 300W |  15551MiB / 16160MiB |     60%      Default |
+-------------------------------+----------------------+----------------------+
|   5  Tesla V100-SXM2...  On   | 00000000:00:1C.0 Off |                    0 |
| N/A   56C    P0    78W / 300W |  15455MiB / 16160MiB |     60%      Default |
+-------------------------------+----------------------+----------------------+
|   6  Tesla V100-SXM2...  On   | 00000000:00:1D.0 Off |                    0 |
| N/A   56C    P0    80W / 300W |  15855MiB / 16160MiB |     67%      Default |
+-------------------------------+----------------------+----------------------+
|   7  Tesla V100-SXM2...  On   | 00000000:00:1E.0 Off |                    0 |
| N/A   67C    P0    87W / 300W |  16027MiB / 16160MiB |     75%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0      2897      C   .../anaconda3/envs/pytorch_p36/bin/python3 15797MiB |
|    1      2898      C   .../anaconda3/envs/pytorch_p36/bin/python3 15703MiB |
|    2      2899      C   .../anaconda3/envs/pytorch_p36/bin/python3 15911MiB |
|    3      2900      C   .../anaconda3/envs/pytorch_p36/bin/python3 15937MiB |
|    4      2901      C   .../anaconda3/envs/pytorch_p36/bin/python3 15539MiB |
|    5      2902      C   .../anaconda3/envs/pytorch_p36/bin/python3 15443MiB |
|    6      2903      C   .../anaconda3/envs/pytorch_p36/bin/python3 15843MiB |
|    7      2904      C   .../anaconda3/envs/pytorch_p36/bin/python3 16015MiB |
+-----------------------------------------------------------------------------+
ubuntu@ip-172-31-42-183:~/detr$
```

Mixed Precision: Inference

In order to convert the Torch checkpoint we just trained on the cloud a helper script 'detr2tensorrt.py' was developed that takes in the checkpoint and a destination filename as arguments. I was reliant on the original build_model function that built the training model in order to cast the saved weights from the checkpoint onto the model.

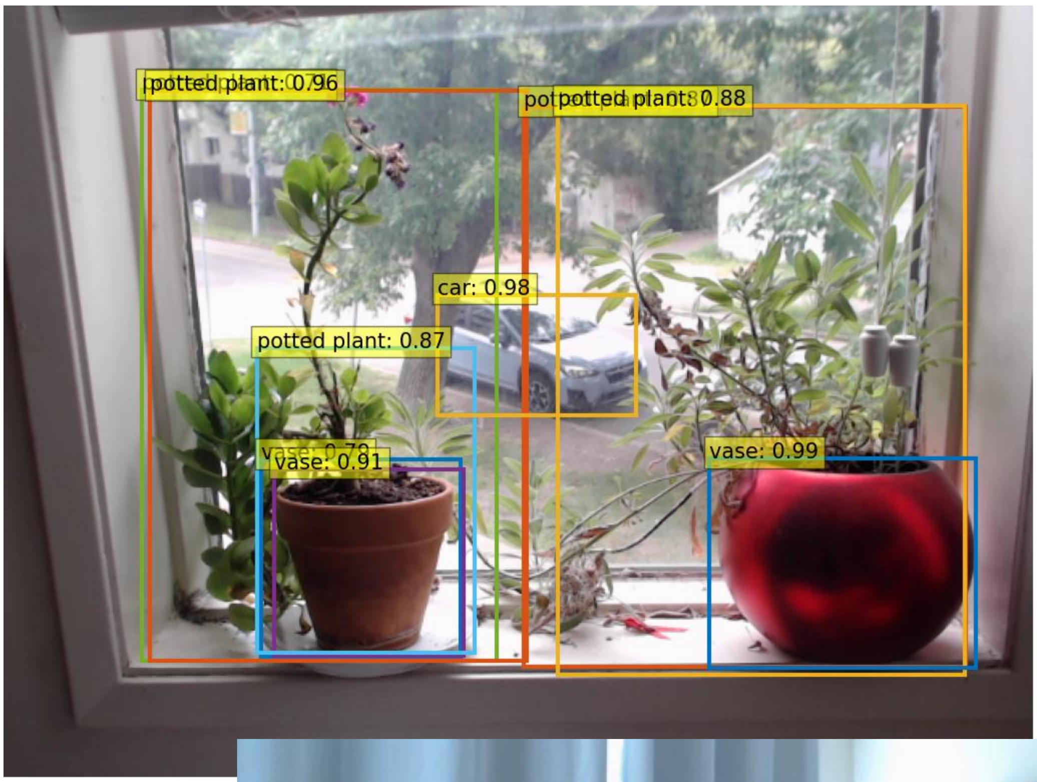
And finally, the script 'webcam2detr.py' was created to open a webcam feed and pass it to the model for inference. Similar to the previous script we are dependant on the original build_model function to recreate the model with weights. Once that has been performed it's a relatively simple task to:

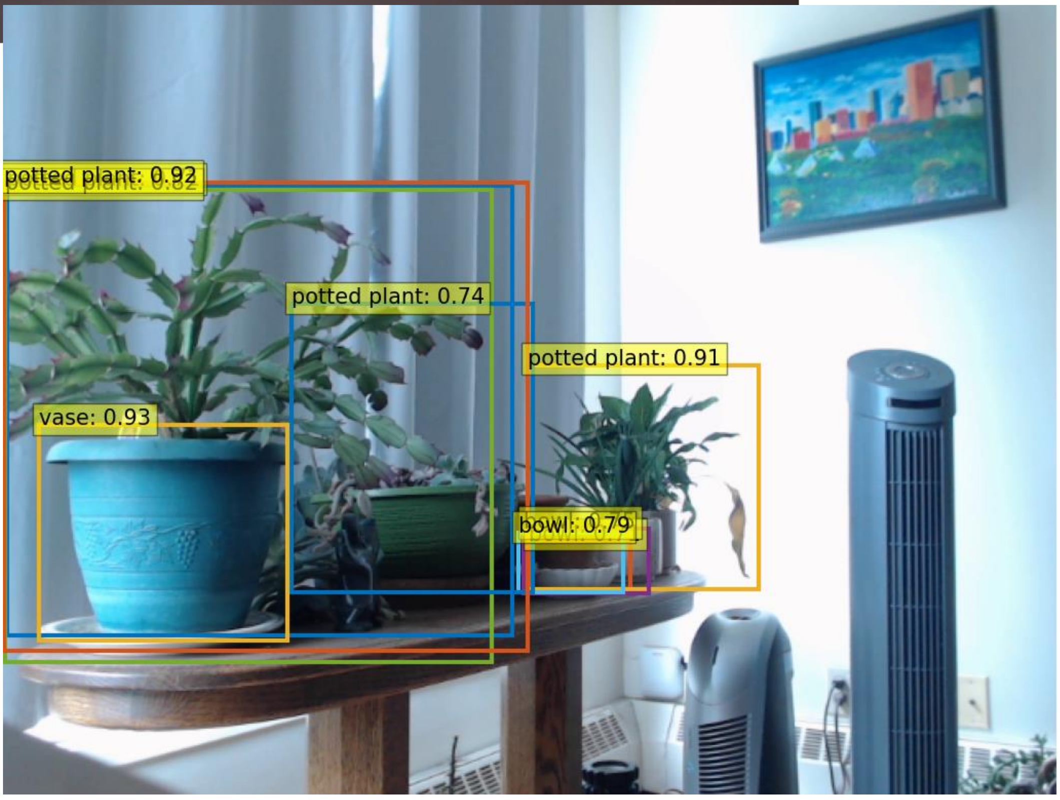
- Transform the image and send it to the GPU
- Evaluate the mode on the image
- Parse the outputs of the model by taking a softmax of class predictions
- Then filtering on a desired confidence level 0.7 (down from their 0.9)
- Rescale these bounding boxes back to image resolution and display results



```
                # propagate through the model
                sample = transform(img).unsqueeze(0)
                #sample.cuda()
                sample.to(device)
                outputs = model(sample)

                # keep only predictions with 0.7+ confidence
                probas = outputs['pred_logits'].softmax(-1)[0, :, :-1]
                keep = probas.max(-1).values > 0.7

                # convert boxes from [0; 1] to image scales
                bboxes_scaled = rescale_bboxes(outputs['pred_boxes'][0, keep], img.size)

                # Display the resulting frame
                plot_results(img, probas[keep], bboxes_scaled)
```

Example images taken from a webcam feed and passed into the model:





Conclusions

The goals of this project were for the most part achieved: convert a model from an academic, pure research state into something that can be deployed on an edge device. While there were challenges along the way in training with 16-bit mixed precision and inference, the model was successfully trained and deployed to the Xavier NX device in order to run predictions off a webcam feed.

Challenges

As anticipated with any bleeding edge release of software/technologies there were several challenges faced during this process.

- GPU drivers were not recent enough for required version of PyTorch
- Supporting Apex install required NGC Docker containers
- GPU utilization on AWS was ~70% with default training settings
- Torch2TensorRT converter library was not intuitive, unresolved errors
- Training on smaller and custom datasets appears to be a growing issue on the main project repository.

Future Goals

There is lots of room to revisit this project and explore the models capabilities. Below are just some of the tasks that could push this model from its research infancy into a fully productionized model:

- Train for Instance (Panoptic) Segmentation
- Multi-server Training support
- Reducing training time and length for desktops
- Smaller CNN backbone to support more edge devices
- Ensure Tensor Cores were engaged on the Xavier NX
- Convert layers to 4-8 bit integer precision for greater speedup

Reference Links

- https://ai.facebook.com/blog/end-to-end-object-detection-with-transformers/
- https://ai.facebook.com/research/publications/end-to-end-object-detection-with-transformers/
- https://www.youtube.com/watch?v=T35ba_VXkMY&t=1464s

- https://colab.research.google.com/github/facebookresearch/detr/blob/colab/notebooks/detr_demo.ipynb
- https://colab.research.google.com/github/facebookresearch/detr/blob/colab/notebooks/detr_demo.ipynb
- https://colab.research.google.com/github/facebookresearch/detr/blob/colab/notebooks/DETR_panoptic.ipynb

Github Repository

- https://github.com/sirakzg/detr/