CS230

Real-Time Object Detection on an Edge Device (Final Report)

Elias Stein, Siyu Liu, John Sun
Department of Electrical Engineering
Stanford University
{eliastein, siyuliu3, js44}@stanford.edu

Abstract

The large model size of modern Deep Learning tasks such as object detection presents challenges for model deployment on edge devices due to significant resource constraints of the embedded hardware. In this project, we explored techniques such as loop fusion and post-training quantization in an effort to achieve real-time performance while maintaining prediction accuracy under limited computational resources. We started with a baseline model of YOLOv3 pretrained on COCO dataset and measured its prediction metrics using mean average precision(mAP) and inference time. Next, we applied transfer learning to a smaller dataset to make a comparison. Lastly, we provide a detailed analysis of experiment results, measured both on Desktop and edge environment.

1 Introduction

Due to the advancement of technologies such as autonomous driving, human vision augmentation, humanoid robotics, there is a rapidly increasing demand for fast and accurate object detection algorithm that can run on edge devices in real-time. The state-of-the-art solution uses huge deep neural network models which require billions of operations per inference. Moreover, such large models are extremely memory intensive, putting more pressure on resource limited mobile devices.

Addressing this challenge through tuning deep learning architectures to strike an optimal balance between performance and accuracy has been a popular research field over the past few years. We sought to explore this trade-off by deploying object detection models onto an edge device for measurement of real-time performance. We selected NVIDIA's Jetson Nano. The Nano is the smallest offering in NVIDIA's Jetson line of specialized SOCs for AI systems. It offers a CPU and GPU integrated into the SOC and has a computational power of 472 GFLOPs, at a power draw of 5 Watts and price of 99 dollars. These specifications fall within what one might reasonably consider for a mass-available edge device, while offering significantly more computational power than other small embedded devices like the RaspberryPi 3 B+.

2 Related Work

We looked into several real-time object recognition architectures that could potentially be used for a Jetson Nano, including publications that present work in MobileNetv2, YOLOv3 and their variants, as well as Faster R-CNN. In general all of these object detection models struggle with the trade-offs between speed and accuracy. Indeed, some models, like MobileNetV2 were designed with speed in mind utilizing Depthwise Separation Convolutions are used to reduce the computational overhead.

CS230: Deep Learning, Autumn 2019, Stanford University, CA.

Similar work has been done by Hoang et al and published in 2019, where a YOLOv3 model was established and retrained while maintaining real-time performance and original accuracy on a laptop setup [1]. On Rasbery Pi, they achieved the same accuracy at the cost of real-time performance.

Stemming from this, we opted to use the YOLO model and attempt to achieve the opposite: increasing real-time performance at the cost of hopefully minimal accuracy loss. We liked that it had not previously taken steps to greatly sacrifice accuracy for speed, although YOLOv3 did start to explore this trade-off with the input size [2]. Additionally, we could not find any official documentation on NVIDIA's website listing performance benchmarks on the NVIDIA hardware [3].

Recall that YOLO is a single neural network that does only 1 inference per input image, therefore easier to train and has more potential to generate real-time inference. YOLO contains 24 convolutional layers connected to 2 fully connected layers [4]. Literature [5] has identified several limitations of YOLO, such as its strong spatial constraints on bounding box predictions since each grid cell only predicts two boxes and can only have one class, as well as it penalizing errors the same regardless of the size of its corresponding box and its impact on IOU. The most recent update, YOLOv3, uses a logistic classifier to calculate the probability of the object that matches a certain label instead of softmax functions. This removes the implicit assumption that each output only belongs to exactly one class, therefore enabling the model to generalize better [6].

3 Dataset and Features

As our main goal is not to realize any particular object detection task, collecting and labelling data manually is out of the scope of this project. We found potential datasets from several sources: KITTI , MS COCO, and VOC datasets, all providing labeled images for deep learning network training. Due to better accessibility and smaller size, we chose to start with VOC dataset for training and validation. The dataset comprises 16.5k training images and 5k validation images in 20 object classes. Input images are normalized and reshaped into fixed shapes that are multiples of 32 in order to accommodate YOLOv3's grid-cell based predictions. In our work, we experimented with three different input shapes: (320, 320), (416, 416) and (608, 608).

To get a higher mAP, we tried to pretrain YOLOv3 on a larger dataset such as MS COCO which has 83k training, 5k validation images and 80 object classes, and apply transfer learning on the relatively smaller VOC dataset. Same pre-processing steps are applied to input images.

4 Methods

4.1 Learning Algorithm

We use the latest version of YOLOv3, whose architecture is shown below in Figure 1.

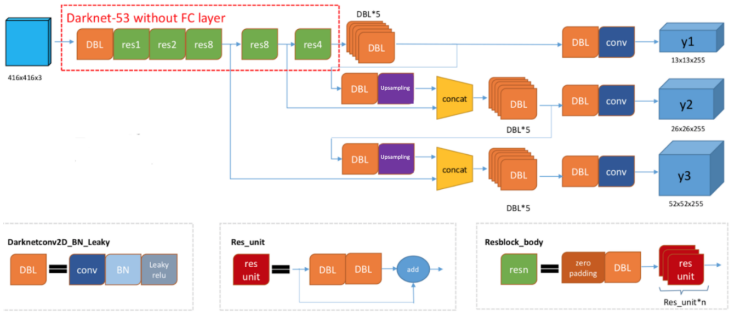

Figure 1: YOLOv3 architecture (text modified by our team) [7]

Loss functions are an important criterion for evaluating the performance of a model. Evolved from the multi-part loss function of YOLO:

$$\lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] + \lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] + \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2$$
$$+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2 + \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2$$

where $\mathbb{1}_i^{obj}$ denotes if object appears in cell i and $\mathbb{1}_{ij}^{obj}$ denotes that the jth bounding box predictor in cell i is "responsible" for that prediction [5].

For calculating class confidence, YOLOv3 switched from softmax to independent logistic regressions(sigmoid), enabling each box to have multiple classes. Moreover, YOLOv3 changed the last part of the loss function responsible for class mismatch from MSE to cross-entropy loss function.

We set up the training to iterate over 20,000 batches to minimize the above loss function. Training starts from scratch with randomly initialized data. Learning rate starts with 0.001 with momentum = 0.9 and decay = 0.0005. Data augmentation is significantly used by randomly changing the exposure and saturation of the input images by a factor of 1.5 and hue by a factor of 0.1. We also randomly rotated and scaled the inputs up to 15%.

4.2 Training Via Transfer Learning

To improve mAP, we trained our model on VOC dataset from pretrained weights on COCO, which contains the backbone of Darknet-53 [8] for feature extraction. Since the dimension of the last layer of YOLO depends on the number of classes, we have to modify the configuration of output shape correspondingly according to the shape $(n_{batch}, S, S, (4 + 1 + n_{class}) * n_{anchor})$. where S is the size of the grid-cell.

4.3 Tools and Framework Pipeline

We used Darknet[9] for mAP calculation and training our models in full precision on the VOC dataset.

For optimizations, we used TensorRT[10], an real-time inference framework from NVIDIA. We started from NVIDIA example code which converts YOLOv3-608 model/weights into ONNX format and then builds a TensorRT engine for inference speed testing. We modified this code to additionally build the YOLOv3-320, and YOLOv3-416 size models and YOLOv3 models trained on VOC. We then updated it to process the COCO and VOC datasets and do the appropriate post processing to save out predictions.

To calculate the mAP for a given test, utilized code from another public github[7], adjusting our post processing to provide a compatible format.

The same software packages / environment were setup on a desktop with an RTX 2060 GPU and on the Jetson nano for inference testing. Setup on the Nano proved to be particularly challenging as many off-the-shelf SW packages are not compatible with embedded HW structure. This was the main driver in choosing TensorRT for optimization vs. other popular packages such as Tensorflow-lite. Due to struggles with package management, especially around compatible NVIDIA GPU libraries, we learned about Docker and used it to set up environments.

5 Results and Analysis

5.1 Metrics

We used mean average precision(mAP) to evaluate our model. It is calculated by taking the mean of the average precision (AP) of all object classes. AP can be calculated by taking the area under precision-recall curve(AUPRC) for all classes, where True Positives and False Positives are evaluated using IoU between predictions and ground truths:

$$AP = \int_0^1 p(r)dr, \quad mAP = \frac{1}{C} \sum_{i=1}^{C} AP_i$$

Inference time is a measure of the inference execution only, and does not include the pre or post processing steps. Measurements are made on a RTX 2060 GPU and a Jetson Nano, and inference time (ms) and mAP (%) are recorded in Figures 2 - 4. As a comparison, we also measured pretrained weights performance on COCO dataset.

YOLOv3 - COCO - RTX 2060	No TensorRT (Baseline)		Loop Fusion + Kernel Tuning Optimizations				LF + KT + FP16 Quantization			
Size	Time / Inf (ms)	mAP	Time / Inf (ms)	Speedup Multiplier	mAP	mAP Change	Time / Inf (ms)	Speedup Multiplier	mAP	mAP Change
320	27	52.48%	10	2.70x	35.35%	-32.64%	6	4.50x	35.37%	-32.60%
416	33	54.59%	16	2.06x	38.37%	-29.71%	8	4.13x	38.37%	-29.71%
608	64	55.16%	26	2.46x	38.60%	-30.02%	11	5.82x	38.60%	-30.0%

Figure 2: YOLOv3 Performance on COCO dataset on RTX 2060 GPU

YOLOv3 - COCO - Jetson Nano	No TensorRT (Baseline)		Loop Fusion + Kernel Tuning Optimizations				LF + KT + FP16 Quantization			
Size	Time / Inf (ms)	mAP	Time / Inf (ms)	Speedup Multiplier	mAP	mAP Change	Time / Inf (ms)	Speedup Multiplier	mAP	mAP Change
320	323	52.48%	249	1.30x	35.36%	-32.62%	154	2.10x	35.36%	-32.62%
416	500	54.59%	392	1.28x	38.24%	-29.95%	248	2.02x	38.24%	-29.95%
608	909	55.16%	805	1.13x	38.60%	-30.02%	501	1.81x	38.60%	-30.02%

Figure 3: YOLOv3 Performance on COCO dataset on Jetson Nano

YOLOv3 - VOC - RTX 2060	No TensorRT (Baseline)		Loop Fusion + Kernel Tuning Optimizations				LF + KT + FP16 Quantization			
Size	Time / Inf (ms)	mAP	Time / Inf (ms)	Speedup Multiplier	mAP	mAP Change	Time / Inf (ms)	Speedup Multiplier	mAP	mAP Change
320	26	75.10%	13	2.00x	64.33%	-14.34%	5	5.20x	64.32%	-14.35%
416	34	77.40%	20	1.70x	68.43%	-11.59%	7	4.86x	68.44%	-11.58%
608	62	76.64%	35	1.77x	68.92%	-10.07%	12	5.20x	68.95%	-10.03%

Figure 4: YOLOv3 Performance on VOC dataset on RTX 2060 GPU

5.2 Testing of YOLOv3 on COCO Dataset on Desktop RTX 2060 (RTX) and Jetson Nano (Nano)

From Figures 2 and 3 above, we see that our baseline measurements of YOLOv3 are as expected, roughly reproducing the mAP results reported by Redmon's paper [2]. Since the CNN weights are independent of input shape, we can directly measure the trade-off between inference speed and mAP by adjusting the input resolution without re-training the weights. We can see that on both the RTX and Nano we pay significant speed costs, approximately 2x and 3x respectively, for marginal increases in mAP. The Nano fares worse given it's limited compute power and resulting sensitivity to increases in FLOPS.

Next we measured YOLOv3 performance with for non-quantization optimizations like layer fusion (LF) and kernel tuning (KT). The speed increased significantly (2-3x for RTX and 1.2x for Nano) but the mAP score dropped significantly. This was quite surprising, since the applied optimizations should

only work to increase arithmetic intensity and reduce loads and stores to memory. Interestingly, a review of NVIDIA discussion boards indicated that this may be due to a bug in TensorRT's implementation. It was potentially fixed in the latest release (6.0), however, we retested in a new environment using 6.0 and got the same results.

Lastly, we measure performance including 16-bit floating point quantization. We observed a negligibly small decrease in mAP with another significant boost to inference speed (2.5x for RTX and Nano). This result is reasonable as we would expect that the error introduced by precision loss to be minimal in a large NN like YOLOv3 since less importance will be given to any one weight value. Furthermore, reducing precision loss will have a greater impact on small weight values (due to the nature of floating point), but these values inherently contribute less to the output calculation.

In terms of speedup, we see that the Nano derived limited speedup benefit from this from LF and KT optimizations, as opposed to the RTX. This makes sense as the Nano is likely already compute limited and thus can't derive the same benefits from increased arithmetic intensity. Additionally, the RTX hardware includes TensorCores allowing TensorRT to to select more efficient kernel implementations.

5.3 Testing of YOLOv3 on VOC Dataset

As mentioned in 5.2, loop fusion seems to reduce mAP significantly on COCO dataset. We are curious to see how the same transformation will impact performance on a different dataset. Therefore, we took the pretrained weights on COCO and retrained them on VOC dataset which has a smaller number of classes. For this dataset, the model does not need to learn subtle changes between similar classes, thus our theory was that it would be more robust to the error introduced by optimization methods. We repeated our inference time and accuracy measurements on the YOLOv3 model trained for VOC. The results can be seen in Figure 4. An average mAP of 76% is obtained as baseline. After optimization, we observe a similar improvement in the inference execution time, and a smaller drop in accuracy (albeit still relatively large at around 12%). The smaller drop in accuracy would imply that this model was less sensitive to the TensorRT optimizations. Since VOC dataset has only 20 classes, it is relatively easier to get a higher mAP on VOC dataset compared to COCO dataset because the model does not need to recognize similar classes that are hard to distinguish. Moreover, having fewer classes did not help with inference speed as expected because the reduction in computation of the last layer is negligible compared to the entire model.

6 Conclusion and Future Work

Overall, we found that applying optimization techniques such as quantization could provide for a 2x - 6x increase in inference execution speed. Such performance gains could be critical for real world deployment on an edge device. Of course, in our testing this came with a significant drop in accuracy, which may ultimately outweigh such speed gains.

Our primary next step would be to setup a new optimization toolchain that does not use the TensorRT framework. The large drop in mAP we observed is highly suspect and we would want to independently recreate it. Afterwards, future work could be directed into the exploration of 8-bit integer quantization. This is a more aggressive precision reduction method that will further enhance the performance, but the exact degree of trade-off remains to be examined. The Jetson Nano does not support INT8 operations, but another embedded device, like Google's Coral, could be used instead.

7 Contributions

Elias Stein: SW environment setup on Desktop and Jetson Nano. Primary packages for support include, Tensorflow-GPU, CUDA-10.0, Cudnn 7.5, TensorRT 5.1.6. Setting up the ONNX-TensorRT pipline and reworking the python scripts to support multiple permuations of YOLOv3 models (sizes and datasets), and to handle processing of the COCO dataset and VOC dataset.

Siyu Liu: researched and explored different dataset, trained model and tuned hyperparameter using Darknet on AWS, measured model metrics mAP on desktop, wrote script to generate ground truth bounding box from coco website.

John Sun: literature review on TensorRT various models including MobileNetv2, YOLOv3, Faster R-CNN, and ResNet, and their trial implementation in Colab, AWS setup and management, results post-processing and presentation, communication with course staff .

References

[1] N. Hoang and A. Minh, "Computer vision deployment on edge devices," 2019.

[2] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," in *arXiv preprint arXiv:1804.02767*, 2018.

[3] "Jetson nanol deep learning inference benchmarks," Dec 2019. [Online]. Available: https://developer.nvidia.com/embedded/jetson-nano-dl-inference-benchmarks

[4] H.-J. Jeong, K.-S. Park, and Y.-G. Ha, "Image preprocessing for efficient training of yolo deep learning networks," in *2018 IEEE International Conference on Big Data and Smart Computing (BigComp)*. IEEE, 2018, pp. 635–637.

[5] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[6] A. Ammar, A. Koubaa, M. Ahmed, and A. Saad, "Aerial images processing for car detection using convolutional neural networks: Comparison between faster r-cnn and yolov3," *arXiv preprint arXiv:1910.07234*, 2019.

[7] YunYang1994, "Yunyang1994/cv-notebooks." [Online]. Available: https://github.com/YunYang1994/cv-notebooks/blob/master/ai_algorithm/YOLOv3.md

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[9] AlexeyAB, "Alexeyab/darknet," Dec 2019. [Online]. Available: https://github.com/AlexeyAB/darknet

[10] "Nvidia tensorrt," Oct 2019. [Online]. Available: https://docs.nvidia.com/deeplearning/sdk/tensorrt-developer-guide/index.html