Enhanced Deployment of YOLO V8 on Nvidia Jetson Nano through Model Compression Techniques

Urmil Jagad
School Of Engineering and Applied Science
Ahmedabad University
Ahmedabad, India
urmil.j@ahduni.edu.in

Harsh Loriya
School Of Engineering and Applied Science
Ahmedabad University
Ahmedabad, India
harsh.l2@ahduni.edu.in

Rahul Patel
School Of Engineering and Applied Science
Ahmedabad University
Ahmedabad, India
rahul.p2@ahduni.edu.in

***Abstract*— This report describes the steps in the project "Enhanced Deployment of YOLO V8 on Nvidia Jetson Nano through Model Compression Techniques". The project was initiated by gaining a solid understanding of the key technologies involved. This involved looking into YOLOv8, its architecture, the Nvidia Jetson Nano platform, and various model compression approaches. To obtain real experience with model compression, post-training quantisation was applied to the YOLOv8 models (YOLOv8-n and YOLOv8-s) on VSCode Python environment with the help of Pytorch tutorials, YOLOv8 documentations, and TensorRT documentations. Then the models were deployed on JetSon Nano as well. The comparisons of models on the Vscode python environment and JetSon Nano are further discussed in the result section of the report.**

***Keywords*—DL, YOLOv8, Model Compression Techniques, Quantization**

I. Introduction

The need for effective object identification systems has increased recently in parallel with the quick development of computer vision technologies. One notable real-time object detection system that is well-known for its remarkable speed and accuracy is YOLO (You Only Look Once). This work explores the details of YOLOv8, the most recent version of the YOLO architecture, in order to maximise its implementation on the Nvidia Jetson Nano, a popular edge computing platform.

Furthermore, model compression techniques play an important role in optimizing the deployment of deep learning models to edge devices. Model compression techniques make efficient inference possible by reducing model size and computational complexity while maintaining accuracy. Throughout the past month, we have focused on understanding YOLOv8's architecture, learning about the potential of Nvidia Jetson Nano, and experimenting with various model compression techniques.

In particular, we explored to implement post-training quantisation to some extent. Post-training quantisation is a popular model compression technique. Post-training quantisation reduces numerical representation precision by quantifying the model's weights and activations, resulting in reduced model sizes and faster inference times. PyTorch tutorials and YOLO documentation provided by Ultralytics were used to construct post-training quantisation, with experiments conducted on Google Colab and Kaggle platforms.

In order to improve the YOLOv8 model for deployment on the Jetson Nano, our research methodology includes inves tigating model compression techniques.We have generated re sults in the ONNX file format despite encountering TensorRT requirements issues. These findings provide a solid basis for optimising real-time object detection in edge computing settings, along with our comprehension of YOLOv8, the Jetson Nano, and model compression methods.

II. Understanding of yolo v8

Ultralytics has released YOLOv8, the most recent version of YOLO. As a cutting-edge, state-of-the-art (SOTA) model, YOLOv8 builds on prior versions' success by offering new features and upgrades that boost performance, flexibility, and efficiency. YOLOv8 covers a wide range of visual AI tasks, including as detection, segmentation, posture estimation, tracking, and classification. This versatility enables users to apply YOLOv8's capabilities to a wide range of applications and areas. UNDERSTANDING OF MODEL COMPRESSION TECHNIQUES

Model compression reduces the size of the neural network while maintaining the accuracy. The resulting models are memory and energy-efficient.

A. Weight Quantization

Quantization refers to techniques for carrying out computations and storing weight/tensors with lower bitwidths than floating point precision. High speed vectorized operations on a variety of hardware platforms and a more condensed model representation are possible with it. The INT8 quantization feature of PyTorch allows for a 4x reduction in model size and memory bandwidth needs. Being limited to the forward pass for quantized operators, this method is mainly used to accelerate inference.

Post-training static quantization involves converting weights from float to int and feeding data through the network. It computes distributions of different activations by inserting observer modules at different points. These distributions are used to determine quantization at inference time. A simple technique is to divide the entire range of activations into 256 levels, but more sophisticated methods are supported. This step allows for passing quantized values between operations, resulting in a significant speed-up by converting them to floats and back to ints.

III. Understanding Jetson Nano

Image 1 [8].

The Jetson Nano is a small and powerful computer designed for artificial intelligence and embedded systems. It is a popular choice for developers wishing to create AI-powered devices due to a variety of capabilities. The Jetson Nano boasts a compact design and low power consumption, using as little as 5 watts of power while being only slightly bigger than a credit card. This makes it ideal for applications where size and power are limited, such as smart cameras, drones, and robotics.

With its powerful NVIDIA GPU, the Jetson Nano can run many neural networks at once. This makes it ideal for high-demanding applications requiring a lot of processing, such as speech recognition, object identification, and image classification. It's easy to start developing AI applications with the Jetson Nano thanks to the comprehensive software development kit (SDK). The SDK contains all the necessary components for developing, launching, and utilizing AI apps on the Jetson Nano.

IV. ONNX and TensorRT

ONNX (Open Neural Network Exchange) is an open format for representing deep learning models, allowing interoperability between various frameworks like PyTorch, TensorFlow, and others. It simplifies the process of transferring models between different environments, facilitating collaboration and deployment across diverse platforms.

NVIDIA created the high-performance deep learning inference library known as TensorRT (TensorRT Runtime). Deep learning models for inference are accelerated and optimised, especially on NVIDIA GPUs. To achieve quick and effective inference performance, TensorRT uses several strategies, including layer fusion, dynamic tensor memory allocation, and precision calibration.

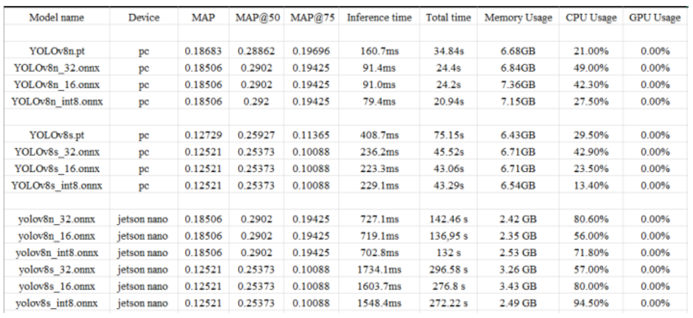
TensorRT is essential for deployment on NVIDIA platforms such as the Jetson Nano. It can use a format called.engine files to optimize models for deployment froTensotRT in Jetson Nano, so we generated all of these results on the Onnxse.engine files offer a quicker and less latency representation of the model that is tuned for effective inference on NVIDIA GPUs. For resource-constrained devices like the Jetson Nano, where optimizing performance while conserving resource utilization is crucial, this optimization is especially crucial.

Due to a dependency fault in the library and versions, we were unable to use TensotRT in Jetson Nano, so we generated all of these results on the Onnx file format.

V. Methodology

The team started by understanding the problem. Our method was to apply the model compression techniques on quantised Yolov8 models. We obtain the trained model Yolov8n and Yolov8s versions after understanding the model and the methods of weight quantisation. We first exported the original models in the onyx format for better implementations on Jetson Nano. Then, the exported models were quantised from float32-bit floating integers to float16-bit floating integers to int8 floating integers. We were also given a pedestrian detection dataset consisting of 160 images and had annotations in XML format. We converted this dataset in the validation format of YOLOv8 i.e. YAML. We validate this dataset on the exported models and quantised models..

VI. Results and reasoning



Model name	Device	MAP	MAP@50	MAP@75	Inference time	Total time	Memory Usage	CPU Usage	GPU Usage
YOLOv8n.pt	pc	0.18683	0.28862	0.19696	160.7ms	34.84s	6.68GB	21.00%	0.00%
YOLOv8n_32.onnx	pc	0.18506	0.2902	0.19425	91.4ms	24.4s	6.84GB	49.00%	0.00%
YOLOv8n_16.onnx	pc	0.18506	0.2902	0.19425	91.0ms	24.2s	7.36GB	42.30%	0.00%
YOLOv8n_int8.onnx	pc	0.18506	0.292	0.19425	79.4ms	20.94s	7.15GB	27.50%	0.00%
YOLOv8s.pt	pc	0.12729	0.25927	0.11365	408.7ms	75.15s	6.43GB	29.50%	0.00%
YOLOv8s_32.onnx	pc	0.12521	0.25373	0.10088	236.2ms	45.52s	6.71GB	42.90%	0.00%
YOLOv8s_16.onnx	pc	0.12521	0.25373	0.10088	223.3ms	43.06s	6.71GB	23.50%	0.00%
YOLOv8s_int8.onnx	pc	0.12521	0.25373	0.10088	229.1ms	43.29s	6.54GB	13.40%	0.00%
yolov8n_32.onnx	jetson nano	0.18506	0.2902	0.19425	727.1ms	142.46 s	2.42 GB	80.60%	0.00%
yolov8n_16.onnx	jetson nano	0.18506	0.2902	0.19425	719.1ms	136,95 s	2.35 GB	56.00%	0.00%
yolov8n_int8.onnx	jetson nano	0.18506	0.2902	0.19425	702.8ms	132 s	2.53 GB	71.80%	0.00%
yolov8s_32.onnx	jetson nano	0.12521	0.25373	0.10088	1734.1ms	296.58 s	3.26 GB	57.00%	0.00%
yolov8s_16.onnx	jetson nano	0.12521	0.25373	0.10088	1603.7ms	276.8 s	3.43 GB	80.00%	0.00%
yolov8s_int8.onnx	jetson nano	0.12521	0.25373	0.10088	1548.4ms	272.22 s	2.49 GB	94.50%	0.00%

- Model Performance (MAP): The variations in MAP (Mean Average Precision) among different models and quantization techniques are minimal, suggesting that the model compression techniques applied do not significantly impact the accuracy of the YOLO V8 model.
- Inference Time: Introducing model compression techniques, particularly quantization, significantly reduces inference time. For instance, the inference time decreased from around 370ms to approximately 70ms for YOLO8s.pt and int8.onnx, respectively.

- Total Time: Total time includes both inference and preprocessing/postprocessing times. While inference time reduced significantly with quantisation, total time showed varying degrees of reduction, indicating other factors such as preprocessing and postprocessing contribute to the overall execution time.

- Memory Usage: Interestingly, memory usage fluctuates across different models and quantisation techniques. No consistent trend has been observed, suggesting that memory consumption is influenced by various factors beyond just model architecture and quantisation.

- CPU and GPU Usage: The CPU and GPU usage percentages provide insights into the resource utilization during inference. Despite variations in model architecture and quantization, there's no significant difference observed in CPU and GPU usage, implying that the computational load is distributed evenly across these resources.

Regarding the changes experienced during the project, the variations in performance metrics could be attributed to several factors, such as:

- The bottleneck in Other Components: The inference time and CPU usage may not change significantly because the bottleneck in the system might not be the model inference itself. Other components such as data loading, preprocessing, or postprocessing could be consuming a significant portion of the overall time. If these components remain constant across different models or quantization techniques, the changes in inference time or CPU usage may not be substantial.

- Hardware Limitations: The hardware on which the inference is being performed, such as the CPU and GPU, may have limitations that prevent significant changes in performance. For instance, if the CPU or GPU is already operating at maximum capacity during inference with one model, switching to another model may not lead to notable improvements in performance due to hardware constraints.

- Optimization Level: The optimization level applied during inference can also impact the extent of changes in inference time and CPU usage. If the models are already heavily optimized or if the optimization techniques used do not vary significantly across models, the differences in performance metrics may be minimal.

- Quantization Overhead: While quantization generally reduces model size and inference time, there may be overhead associated with the quantization process itself. This overhead could offset some of the gains achieved through quantization, resulting in smaller changes in inference time or CPU usage than expected.

- Complexity of Model: If the models being compared are of similar complexity or if the changes in architecture or quantization techniques do not significantly alter the computational requirements of the model, the differences in performance metrics may be marginal.

- Measurement Noise: In some cases, the observed changes in performance metrics may be within the margin of measurement noise. Variability in hardware performance, system load, or other external factors could introduce noise into the measurements, making it challenging to discern small changes in performance accurately.

VII. Conclusion

In conclusion, the project has advanced the study of model compression methods—weight quantisation in particular—for effective Nvidia Jetson Nano platform deployment. We successfully implemented weight quantisation in the ONNX format on PC and Jetson Nano systems through careful development and experimentation.

Nevertheless, difficulties occurred in the implementation stage on Jatson nano, mainly due to version dependencies and import problems, which made it difficult for us to include the model compression method straight into TensorRT. Despite these obstacles, the project has yielded insightful information on the intricacies of model deployment on edge devices and the significance of resolving compatibility problems to ensure seamless integration.

VIII. Future work

It will be easier to include weight quantisation techniques into TensorRT in the future if the version and import requirements are met. This involves carrying out in-depth testing and debugging in order to find and address any dependencies between the components.

Moreover, future versions of the project can also focus on improving the deployed model's efficiency and performance on the Jetson Nano through adaptation of the TensorRT framework. This may involve experimenting with various model compression techniques, adjusting parameters, and utilising TensorRT's accelerated inference capabilities.

References

[1] *Convert and Optimize YOLOv8 with OpenVINO™ — OpenVINO™ documentation*. (n.d.). https://docs.openvino.ai/2023.3/notebooks/122-yolov8-quantization-with-accuracy-control-with-output.html#:~:text=%2C%20log%3DFalse

[2] LoriyaHarsh. (n.d.-a). *GitHub - LoriyaHarsh/CV_2024_3_Group_12*. GitHub. https://github.com/LoriyaHarsh/CV_2024_3_Group_12

[3] Krishnamoorthi, R. (2018, June 21). *Quantizing deep convolutional networks for efficient inference: A whitepaper*. arXiv.org. https://arxiv.org/abs/1806.08342

[4] Ultralytics. (2024, March 13). *Home*. Ultralytics YOLOv8 Docs. https://docs.ultralytics.com/

[5] *Quantization — PyTorch 2.2 documentation*. (n.d.). https://pytorch.org/docs/stable/quantization.html

[6] Pokhrel, S. (n.d.). *4 popular model compression techniques explained*. https://xailient.com/blog/4-popular-model-compression-techniques-explained/

[7] (beta) Static Quantization with Eager Mode in PyTorch — PyTorch Tutorials 2.2.1+cu121 documentation. (n.d.). https://pytorch.org/tutorials/advanced/static_quantization_tutorial.html

[8] *Amazon.in*. (n.d.). https://www.amazon.in/Nvidia-Jetson-Development-Cores-Cortex-A57/dp/B084DSDDLT