Feasibility Analysis of Machine Learning Optimization on GPU-based Low-cost Edges

Jiashun Suo*†, Xingzhou Zhang‡§, Shilei Zhang*†, ✉Wei Zhou*†,Weisong Shi¶
*Engineering Research Center of Cyberspace, Yunnan University, Kunming 650091, China
†School of Software, Yunnan University, Kunming 650091, China
‡Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China
§University of Chinese Academy of Sciences, Beijing 100190, China
¶Department of Computer Science, Wayne State University, Detroit, MI, USA 48202

***Abstract*—Many AI algorithms have been deployed on edge devices as edge computing has the advantages of reducing latency, saving network bandwidth, and protecting data privacy. Whether edge devices can run AI algorithms is an important challenge due to the low-power and low-cost characteristics of edge devices. Therefore, this paper analyzed the performance of optimization techniques by running YOLOv3 on a typical GPU-based low-cost edge device, NVIDIA Jetson Nano. YOLOv3 is a representative object detection algorithm, which is widely used as the benchmark in AI scenarios. We compared latency, memory, and power consumption of three deep learning frameworks, TensorFlow, PyTorch, and TensorRT. Then we squeezed the extreme performance using multiple optimization techniques, including model quantization, model parallelization, and image scaling on TensorRT. The running speed of YOLOv3 increases from 3.9FPS to 13.1FPS on NVIDIA Jetson Nano. It proves that the resource-limited edge device can run AI applications with high computing power requirements in a real-time manner. Moreover, we summarized nine observations and five insights to guide the selection and design of optimization techniques and verified the generalization of these rules on NVIDIA Jetson Xavier NX. We also provided a series of suggestions to help developers choose the appropriate method to deploy AI algorithms on edge devices.**

***Keywords*-Edge computing, Edge intelligence, YOLOv3, NVIDIA Jetson Nano, NVIDIA Jetson NX, Quantification**

I. Introduction

Edge computing calls for processing the data at the edge of the network, which has the potential to reduce latency and bandwidth charges, address the limitation of computing capability of a cloud data center, increase availability as well as protect data privacy and security [12]. Benefit from the above advantages, edge computing has developed rapidly. The combination of edge computing and artificial intelligence has promoted the landing of more application scenarios, including video analytics, smart home, autonomous driving, and so on [13]. These edge intelligence scenarios usually have the low-power and low-cost requirements for the edge devices, for example, inexpensive robots [4], unmanned aerial vehicles (UAVs) [5], and Internet-of-things (IoT) devices [11]. In order to reduce the cost and power consumption of edge devices without affecting the real-time requirements of edge intelligent tasks, many techniques have been proposed, including designing lightweight deep learning frameworks, compressing the computational requirements of the AI models, resizing and scaling the image, and so on.

This work was supported in part by the National Natural Science Foundation of China under Grant 61762089, Grant 61863036, and Grant 71972165, and Yunnan Province Science Foundation for Youths under Grant No.202005AC160007, No.202001BB050034.

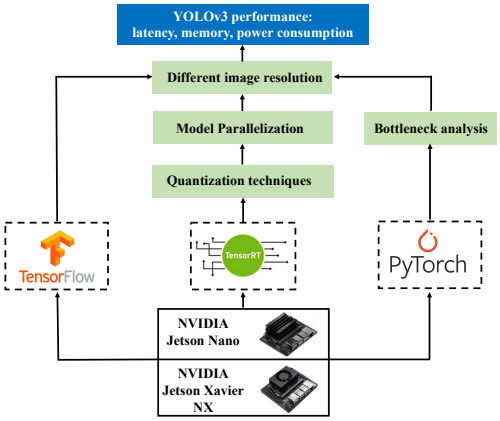


Fig. 1. The overview of the experiments.

However, while many research efforts have been devoted to edge intelligence, several open challenges remain when developing and deploying AI applications on edge in real practice:

- *Effectiveness.* The edge AI-oriented optimization innovations are being introduced at such a rapid pace that researchers are hard-pressed to know their effectiveness when running on specific edge devices, which caused a lot of trial-and-error costs.
- *Performance limitation.* With the reduction of power consumption and cost, the performance of edge devices also decreases, which affects their ability to perform AI tasks. It is difficult for the community to know whether the edge device could meet the real-time requirements.
- *Selection diversity.* The optimization for edge intelligence has multiple dimensions, including lightweight frameworks, model compression, etc. It is difficult for developers to make a suitable hardware and software selection.

In order to address these challenges and analyze the fea-

sibility of the optimization techniques, this paper designed a series of experiments to explore the *effectiveness* of the edge AI-oriented techniques, squeeze the extreme *performance* of an edge device, and provide *selection* suggestions when deploying edge AI applications. This paper chooses YOLOv3 [10], an object detection algorithm, as the edge AI tasks. YOLOv3 is a representative object detection algorithm, which is widely used as the benchmark in AI scenarios. NVIDIA Jetson Nano, an embedded CPU-GPU heterogeneous architecture [6] development device of high cost–performance ratio, is chosen as the target edge device in this paper. The price of Jetson Nano is $99, the power is 5W-10W, and it will provide the peak computing power of 472GFLOPs. If YOLOv3 can be run on Jetson Nano in a real-time manner, we can conclude that object detection tasks can be deployed at the edge at a meager cost. Therefore, **this paper mainly explores whether running YOLOv3 on Jetson Nano can meet real-time requirements**. If not, what is the fastest speed, and which is the best deployment method? In addition, we hope that the experience and characteristics obtained from Jetson Nano can transfer to other GPU-based edge devices.

The overall design of the experiment is shown in Figure 1. We analyze three factors that affect the performance of YOLOv3 on Jetson Nano: frameworks, quantization techniques, and image resolution. The performance indicators include latency, memory usage, and power consumption. Firstly, we compare the performance of three deep learning frameworks, TensorFlow, PyTorch, and TensorRT and found that the latency of PyTorch is much faster than TensorFlow and TensorRT except the post-processing. Therefore, more detailed experiments are used to analyze the performance bottleneck of post-processing for PyTorch. The best overall performance is the TensorRT framework, which has significant advantages in end-to-end speed and memory usage. Consequently, we leverage three optimized methods on TensorRT: 16-bit quantization, adjusting the resolution, and model parallelization. To validate the generalization of these observations, we analyze the performance on another GPU-based edge, NVIDIA Jetson Xavier NX. The experimental results show that most of the rules on Jetson Nano are also suitable for Jetson Xavier NX. After utilizing the optimization techniques, the running speed of YOLOv3 increases from 3.9FPS to 13.1FPS on NVIDIA Jetson Nano and increases from 12.6FPS to 30.3FPS on NVIDIA Jetson Xavier NX, which is sufficient for most edge intelligence task.

The contributions in this paper are as follows:

- We obtained the extreme performance of low-cost edge devices running AI tasks by utilizing various optimization technologies, including model quantization, model parallelization, and image scaling. The maximum speed can reach 13.1FPS on Jetson Nano, which proves that the edge device has the ability to meet the real-time requirements of AI applications.
- We verified the feasibility of edge AI techniques and obtain five insights by comparing and analyzing the performance of deep learning frameworks on NVIDIA Jetson Nano and Jetson Xavier NX. They pointed out the performance bottlenecks and the further optimization directions.
- We provided suggestions when developing and deploying AI tasks on edge, which help developers to choose the suitable configurations.

The rest of the paper is organized as follows. In Section II, we introduced the related work of this paper and the background of edge intelligence. In Section III, we explained the four experimental methods. The observations, insights, and suggestions from the experiment are reported in Section IV and Section V. We summarized the paper in Section VI.

II. Related work and background

A. Edge Intelligence

Edge intelligence calls for the capability to enable edges to execute artificial intelligence algorithms. The edge intelligence capability includes the accuracy of AI models, the latency, memory footprint, and energy of running the models on the edge [15]. Edge intelligence has already been widely used in many scenarios, such as smart home, smart city, and industrial Internet [16]. In [14], the development of edge intelligence on the Internet of Vehicles and the typical use cases were presented.

B. Hardware for the edge intelligence

Many edge hardware has been designed to contribute to the development of edge intelligence. The Google Coral[3] is an inference accelerator at the edge which contains an Edge TPU. Edge TPU is a useful supplement to CPU, GPU, FPGA, and other ASIC solutions that run AI at the edge. The Raspberry Pi 4 [9] is a tiny desktop computer based on ARM. Some simple edge tasks can be deployed on the Raspberry Pi 4. But some powerful AI algorithms are difficult to run because it does not have a GPU to accelerate the algorithm. The NVIDIA Jetson series are edge computing devices with GPUs, including Jetson AGX, Jetson NX, Jetson TX2, Jetson Nano [7], etc.

C. Deep learning packages on the edge

The lightweight deep learning packages are used to speed up the execution, such as TensorFlow, PyTorch, and TensorRT. TensorFlow [1] is developed by Google in 2016 and becomes one of the most widely used deep learning frameworks. PyTorch [2] is published by Facebook, which provides tensor computation with strong GPU acceleration and Deep Neural Networks built on a tape-based auto-grad system to optimize the running time. Developed by NVIDIA company, TensorRT [8] is designed to reduce the latency and increase the throughput when executing the inference task on NVIDIA GPU.

III. Experimental methods

In this section, we will introduce the experimental methods. As shown in Figure 2, this paper designed four experiments to explore the feasibility of the machine learning optimization

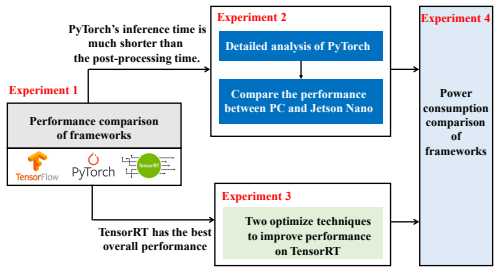


Fig. 2. The design of the experiments.

techniques. In experiment 1, we compared the performance of TensorFlow, PyTorch, and TensorRT running YOLOv3 on Jetson Nano. In experiment 2, we analyzed PyTorch's performance because PyTorch exhibited an unusual performance in experiment 1. Experiment 3 used the optimization techniques to accelerate the running time of TensorRT to find the best configuration for deploying YOLOv3 on Jetson Nano. In experiment 4, we measured the power of TensorFlow, PyTorch, and TensorRT. In addition, we measured the power consumption of TensorRT when leveraging the 16-bit quantization optimization techniques.

The structure of YOLOv3 is shown in Figure 3, which is a deep learning model consisting of Convolutional Neural Network(CNN) layers. The YOLO layer processes the bounding box after the deep neural network. The Post-Processing layer includes filtering(remove the invalid bounding box) and non maximal suppression(NMS).

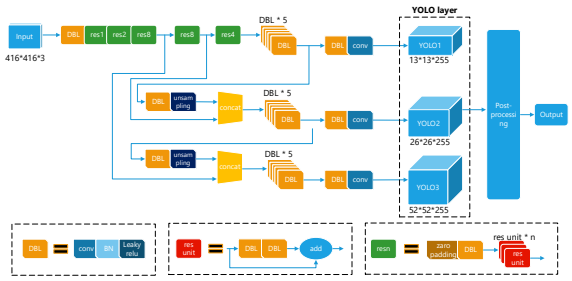


Fig. 3. The structure of YOLOv3(take 416*416 input as an example).

A. Experiment 1: Framework comparison

We compared the latency and memory usage of YOLOv3 under different computational complexity.

For TensorFlow, we selected seven different resolution, 320*320, 416*416, 480*480, 544*544, 608*608, 704*704 and 800*800, to measured the latency and memory usage. In the latency side, we only measured the total end-to-end time because TensorFlow only supports static graphs. In the memory side, its because that TensorFlow will take up all available memory that we can not accurately measure the memory used by models. we only measured a suitable memory by limiting TensorFlow memory to make sure all models used can run.

For PyTorch, we selected ten different resolution, 320*320, 416*416, 448*448, 480*480, 512*512, 544*544, 576*576, 608*608, 640*640 and 672*672, to measured the latency and memory usage. The number of the resolution is larger than TensorFlow because we found some non-linear results between 480*480 to 608*608 resolution. The maximum resolution is 672*672 since Jetson Nano cannot run the YOLOv3 when the resolution is bigger than 672*672. In terms of latency, we measured the model forward time(The running time before the YOLO layer, including the YOLO layer) and post-processing time to analyze the performance bottlenecks.

For TensorRT, the resolution is the same as TensorFlow. We also measured the model forward time and post-processing time. However, unlike PyTorch, the model forward time of TensorRT does not include the YOLO layer because the TensorRT model does not support YOLO layer conversion. The YOLO layer is included in the post-processing process and implemented using NumPy.

B. Experiment 2: PyTorch performance

In experiment 1, we found that PyTorch has some unusual features. First, the forward time is not directly related to the computational complexity of the model and the post-processing time is much longer than the forward time. Second, the end-to-end time has some noticeable nonlinear changes as the computational complexity increase.

We measured the different operations of the post-processing part to find the most time-consuming process. We found that the time is mainly consumed by the filtering(Filter out untargeted boxes based on a threshold) consumption of `torch.where()`. Further, we want to determine whether we can use `numpy.where()` to optimize this operation. Experiments show that `numpy.where()` is much faster than `torch.where()`. But to use `numpy.where()` to process data, we need first to convert the GPU data in PyTorch into CPU data. This conversion time is almost equivalent to the time consumed by `torch.where()`.

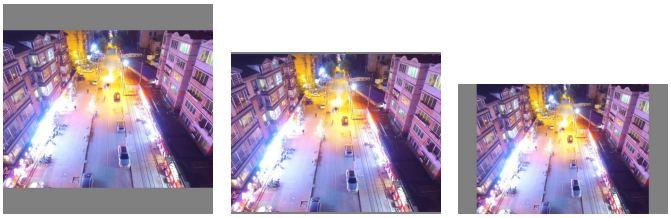
We guess that Jetson Nano's CPU's performance may significantly impact the end-to-end time. To verify this conjecture, we ran the same program on a PC in the same software environment and compared the ratio of model forward time and post-processing time on Jetson Nano and PC to analyze the impact of CPU performance on latency.

C. Experiment 3: TensorRT optimization

In experiment 1, we found that the performance of TensorRT is the best. In this experiment, we leverage three methods to optimize the inference speed of TensorRT to get the best deployment strategy of YOLOv3 on Jetson Nano.

The first method is the 16-bit quantization. TensorRT supports an engine that generates 16-bit quantization. To measure the performance after 16-bit quantization, we used the VisDrone dataset [17] to compare the latency and accuracy of the 16-bit quantization model and the original model.

The second method is to find the most suitable input size according to the picture's ratio to reduce computational complexity. In practical applications, the pictures taken by the camera are mostly 4:3 or 16:9 ratio. Taking a 4:3 image as an example, if the input resolution of YOLOv3 is 416*416, the picture will be scaled in the pre-processing process. The



(a) Scale the image at a ratio of 1:1 (b) Scale the image at the closest ratio of 4:3 (c) Scale the image at the closest ratio of 16:9

Fig. 4. Adjust the length-width ratio of the image.

actual picture's sufficient resolution is 416*312, and the other parts will be filled with monochrome blocks to 416*416, as shown in Figure 4(a). However, the above method will waste some computing power. If we zoom to 416*320 with the resolution closest to 4:3(The minimum input resolution of YOLOv3 is a multiple of 32), the sufficient resolution of the picture is still 416*312, which will reduce computational complexity without the loss of image information, as shown in the Figure 4(b). In the same way, the 16:9 image zoomed in the ratio closest to 16:9 will also reduce a lot of computational complexity. However, scaling the 4:3 picture to 16:9 will lose some image information. By scaling the 4:3 picture to 16:9, the picture's sufficient resolution is 312*234, which results in the loss of some image information, as shown in Figure 4(c). In the experiment, we used the VisDrone dataset to measure the latency and accuracy of the two scaling methods to verify whether the end-to-end time can be accelerated without affecting the accuracy.

Finally, we tested the parallel performance of the optimized model on Jetson Nano, divided into two parts: the maximum number and the average throughput of models in parallel. To illustrate, if two models are running simultaneously, the average throughput is twice the runtime throughput of a model.

D. Experiment 4: Power measurement

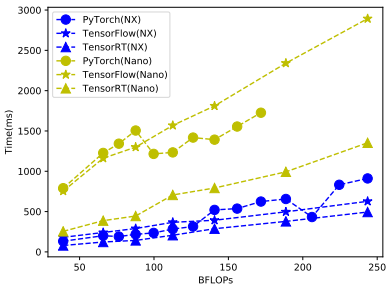
This experiment is designed to measure the power consumption of different frameworks and optimization techniques when running YOLOv3 on Jetson Nano. The experiment includes two aspects. First, We measured the average power of three frameworks running YOLOv3 with different resolutions. Second, We obtain power fluctuations by using high-frequency sampling. We sampled the power consumption of running the YOLOv3 for 100 times on Jetson Nano.

IV. Observations

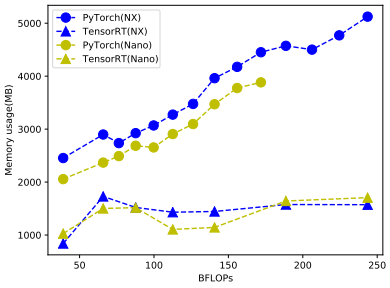
Section III introduced the methods and configuration of the four experiments. In this section, we will show our experimental results and observations.

The relationshiop of end-to-end time and memory usage of Jetson Nano and Jetson Xavier NX with computational complexity are demonstrated on Figure 5(a) and Figure 5(b). We first analyze the results of Jetson Nano and the analysis of Jetson Xavier NX will be written in subsection D.

We can find that the end-to-end time of TensorRT is shorter than that of TensorFlow and PyTorch. The relationships of



(a) Compare the end-to-end time



(b) Compare the memory usage

Fig. 5. Compare the end-to-end time and memory usage of Jetson Nano and Jetson Xavier NX.

running time and the computational complexity for TensorFlow and TensorRT are linear. The function relationships are described in the formula (1) and (3), respectively. The relationship of PyTorch conforms to the formula (2).

$$T = 10.19C + 411.64 \tag{1}$$

$$T = \begin{cases} 14.66C + 230.89, 38.97 \leq C \leq 87.68 \\ 7.70C + 430.06, 99.76 \leq C \leq 126.26 \\ 10.72C - 116.07, 140.69 \leq C \leq 171.86 \end{cases} \tag{2}$$

$$T = 5.32C + 37.62 \tag{3}$$

In formulas (1), (2) and (3), T is time which is represented by ms, and C is computational complexity which is represented by Billion floating-point operations ($BFLOPs$).

With the increase of the computational complexity, the memory usage of TensorRT is stable without increasing. The memory footprint of TensorRT is about 1378MB, respectively. PyTorch's memory will increase linearly with the increase of the computational complexity, which conforms to the formula (4). In formula (4), M is memory, represented by MB. In our experiment, 2.5GB available memory can make sure all TensorFlow models of different resolution to run.

$$M = 14.36C + 1393.52 \tag{4}$$

Table I shows the detailed model forward time and post-processing time of PyTorch because the unusual features that the post-processing time is much more than model forward time.

The observations are summarized as follows:

Observation 1: *With the increase of the model complexity, the end-to-end time of TensorFlow and TensorRT conforms to*

TABLE I
PYTORCH LATENCY PERFORMANCE

Resolution	BFLOPs	Model forward time(ms)	Post-processing time(ms)
320*320	38.97	62.44	727.77
416*416	65.86	62.82	1164.61
448*448	76.38	59.83	1282.47
480*480	87.68	61.04	1444.48
512*512	99.76	60.95	1154.6
544*544	112.62	60.95	1173.2
576*576	126.26	61.48	1356.56
608*608	140.69	59.38	1332.79
640*640	155.89	59.67	1496.43
672*672	171.86	74.68	1651.64

a linear relationship. The end-to-end time of PyTorch has no absolute relationship with the model complexity and, the peak end-to-end time occurs at 87.68BFLOPs.

Observation 2: *With the increase of the model complexity, the memory usage of TensorRT is stable in an interval while PyTorch's memory usage conforms to a linear relationship.*

Observation 3: *As shown in Table I, unlike common sense, the PyTorch's end-to-end time is mainly spent on post-processing rather than model forward, and the model forward time does not increase with the increase of the model complexity.*

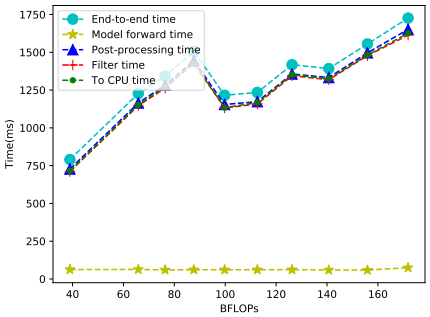


Fig. 6. The running time profiling of PyTorch.

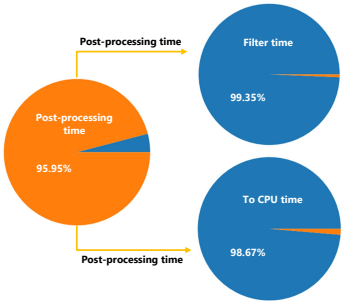


Fig. 7. The running time breakdown of PyTorch when the input resolution is 480*480.

A. The experimental results of PyTorch

According to experiment 1, we found some unusual features of PyTorch. Experiment 2 is designed to further understand the performance of PyTorch. Figure 6 shows the relationship between the time consumed by each operation of PyTorch and model complexity. It shows that the end-to-end time is mainly consumed in the post-processing stage, and the post-processing time is primarily consumed in filtering. The time for converting GPU data to CPU data is the same as the filtering time. The time of model forward is much less the time of post-processing and doesn't change with the increase of FLOPs. Figure 7 shows the proportion of each operation time when the resolution is 480*480 in detail.

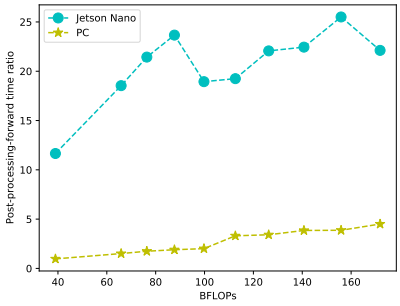


Fig. 8. The ratio of post-processing time and model forward time on PC and Jetson Nano.

Because the model forward part, mainly executed on the GPU, has a short running time, we want to explore whether the performance of the CPU affects the end-to-end time. We used another PC to do the same experiment in the same software environment. The CPU of the PC is Inter(R) Core(TM) i5 9400F, and the GPU is NVIDIA GeForce GTX 1660 SUPER. The Figure 8 shows the ratio of post-processing time to model forward time on Jetson Nano and PC. It is found that the value on PC is much smaller than that on Jetson Nano, which shows that the performance bottleneck of using the PyTorch framework to run YOLOv3 on Jetson Nano is mainly on the CPU. If the CPU performance is improved, the end to end speed will also be significantly improved.

The observations are summarized as follows:

Observation 4: *PyTorch's end-to-end time is mainly spent on post-processing, and the post-processing time is mainly spent on filtering. In the process of filtering with NumPy, the time for converting GPU data to CPU data is almost the same as the time of* `torch.where()`*. Taking 480*480 resolution input as an example, the post-processing time accounts for 95.95% of the end-to-end time, the filtering time accounts for 99.35% of the post-processing time, and the time from GPU data to CPU data accounts for 98.67% of the post-processing time.*

Observation 5: *Improving the performance of CPU is beneficial in improving the deployment performance of PyTorch.*

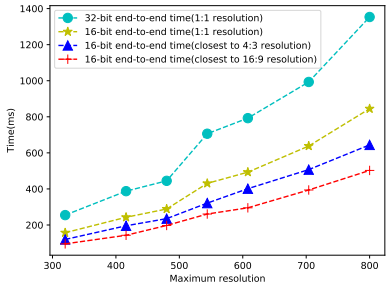
B. The optimization of TensorRT

We used the VisDrone dataset to compare the optimization methods of TensorRT introduced in section III. The pictures in the VisDrone dataset have two styles of 4:3 and 16:9. As shown in Table II, compared with the resolution of 416*416,

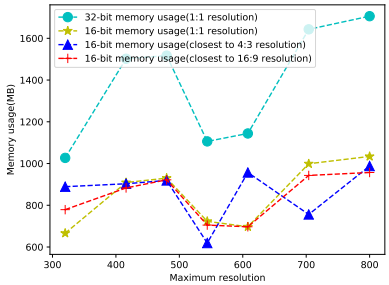
the accuracy of 416*320, which is the closest to 4:3, is slightly improved, which can be regarded as unchanged. When testing at the resolution of 416*256, which is the closest to 16:9, the accuracy is reduced by less than 1%, mainly because it will lose some image information when the 4:3 style image is scaled to 416*256. The fifth row indicates the accuracy of 16-bit quantization at 416*256 resolution, which can be regarded as unchanged compared to the 32-bit result.

TABLE II
COMPARISON OF ACCURACY UNDER DIFFERENT RESOLUTIONS AND QUANTIZATION BITS

Resolution	Bits	AP(%)	AP50(%)	AP75(%)
416*416	32	8	18.39	6.02
416*320	32	8.05	18.49	5.99
416*256	32	7.65	17.59	5.63
416*256	16	7.66	17.61	5.65



(a) End-to-end time



(b) Memory usage

Fig. 9. The end-to-end time and memory usage of TensorRT using different deployment methods with different input resolutions on Jetson Nano.

Figure 9(a) shows that the end-to-end time using TensorRT with different quantization conditions and different resolution ratios. The maximum resolution represents the max of resolution. For example, the maximum resolution of 416*416 and 416*320 are both 416.

Figure 9(b) shows the memory usage of the TensorRT model using different quantization accuracy and input resolution ratios to deploy YOLOv3.

In parallel experiments, the maximum running numbers of the TensorRT 16-bit model with 16:9 input resolution in parallel are three on Jetson Nano. Figure 10 shows the parallel performance.

The observations are summarized as follows:

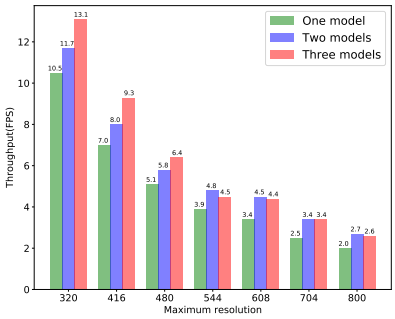


Fig. 10. The average throughput of YOLOv3 model on Jeton Nano with different parallelism.

Observation 6: *Using the 16-bit quantization model and scaling the input resolution of YOLOv3 will reduce memory usage and improve running speed without loss of accuracy. Compared with the original model, using 16-bit quantization and scaling input resolution to approach 16:9 can increase the speed from 3.9FPS to 10.5FPS and reduce memory usage from 1027MB to 779MB without loss of accuracy when the maximum input resolution is 320.*

Observation 7: *The latency of YOLOv3 (the unit is FPS) will improve when the model is running in parallel. Compared with the two models in parallel, the throughput of the three models in parallel has a higher increase when the maximum resolution is less than 480. As the resolution continues to increase, the performance of the multi-model parallel is no longer improved.*

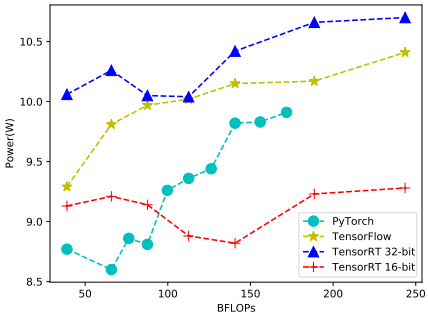


Fig. 11. The average power of TensorFlow, PyTorch, TensorRT 32-bit, and TensorRT 16-bit model.

C. Power measurement

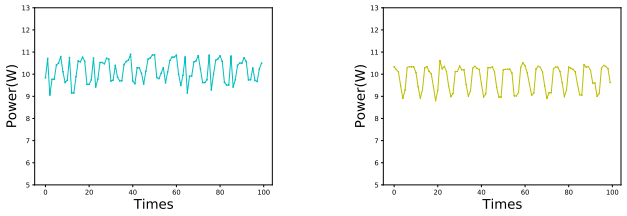
We measured the average power of three frameworks. Figure 11 shows the change in power of TensorFlow, PyTorch, TensorRT 32-bit, and TensorRT 16-bit with the model complexity.

We chose the 608*608 input resolution to show the results of power fluctuation as shown in Figure 12.

The observations are summarized as follows:

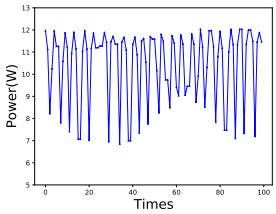
Observation 8: *When the resolution is less than 480*480, PyTorch is energy-saving. Meanwhile, when the resolution is greater than 480*480, TensorRT with the 16-bit quantization techniques is most energy-efficiency.*

Observation 9: *The power fluctuation of TensorRT is greater than that of TensorFlow and PyTorch.*

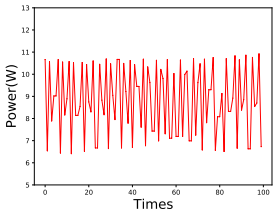


(a) TensorFlow power

(b) PyTorch power



(c) TensorRT 32-bit power



(d) TensorRT 16-bit power

Fig. 12. The power of TensorFlow, PyTorch, TensorRT 32-bit, and TensorRT 16-bit model in the 608*608 input resolution.

D. Generalization

This subsection explores whether the rules on Jetson Nano are suitable for other GPU-based edge devices. We compared the performance of TensorFlow, PyTorch and TensorRT on NVIDIA Jetson Xavier NX which is designed for high-performance compute and AI in embedded and edge systems. The theoretical peak computing power of Jetson Xavier NX is 21 TOPS, which is equipped with 6-core NVIDIA Carmel ARM CPU, 384-core NVIDIA GPU, and two NVIDIA Deep Learning Accelerators engines.

Figure 5(a) compares the end-to-end time on Jetson Nano and Jetson Xavier NX. The experimental results on Jetson Xavier NX are similar to Jetson Nano. The end-to-end time of TensorFlow and TensorRT is proportional to the model complexity on Jetson Xavier NX. TensorRT presents the shortest running time among the three packages. There is still a trough in the runtime of PyTorch on Jetson Xavier NX, but the model complexity at this time is 206.17BFLOPs instead of the 99.76BFLOPs on Jetson Nano.

Figure 5(b) compares the memory usage on Jetson Nano and Jetson Xavier NX. The experimental results on Jetson Xavier NX are also similar to Jetson Nano. The memory usage of PyTorch is proportional to the model complexity, while TensorRT's memory usage is relatively stable, remaining between 800MB and 1600MB.

In addition, we also explored the performance of optimization techniques and the power usage on Jetson Xavier NX. Using the quantization techniques and scaling the image to suitable resolution will improve the performance on the latency and memory usage. When running YOLOv3 on Jetson Xavier NX, the peak performance will achieve **30.3FPS**. TensorRT's power consumption is minimal between the three packages, while the power fluctuation is the biggest. The maximal average power consumption of TensorRT, TensorFlow, and PyTorch are **15.73W, 19.06W, and 20.07W,** respectively.

In summary, by comparing the running time, memory usage, power consumption, and optimization techniques, we found that most of the rules on Jetson Nano are also suitable for other GPU-based edges, Jetson Xavier NX, which validates that the generalization of these observations.

V. Insights and Suggestions

This section will introduce the insights and provide suggestions to deploy YOLOv3 on Jetson Nano.

A. Insights

The insights are summarized as follows:

Insight 1: The formula of end-to-end time with the model computational complexity of TensorFlow, PyTorch, and TensorRT is in the form of $T = aC+b$. a represents the increasing trend of time as computational complexity increases, and b represents the initial time consumption. Formulas 1 to 3 show that TensorRT has the smallest initial time consumption and time growth trend.

Insight 2: The initial memory usage of PyTorch and TensorRT are 1393MB and 1378MB, respectively. But unlike the memory usage of TensorRT is stable, the PyTorch's memory usage increases with the increase of the model computational complexity. The relationship is show in formula 4.

Insight 3: PyTorch's performance on edge devices is largely affected by the CPU. If the CPU is improved, the performance of PyTorch when running YOLOv3 will be improved.

Insight 4: The running time will be improved when the model is in parallel because the model forward needs to use GPU, and the post-processing just uses CPU. Parallelization can make full use of heterogeneous resources where a model uses GPU and other models use CPU. With the increases in the model computational complexity, the performance of parallelization decreases because the forward time increases more than the post-processing time. The latency can't be improved due to the insufficient computing resources.

Insight 5: TensorRT's power fluctuation is greater than TensorFlow and PyTorch because TensorRT uses NumPy to implement the post-processing part, which saves the power consumption of GPU. The TensorRT with the 16-bit quantization model has lower average power while the peak power is the same as TensorFlow model and PyTorch model.

B. Suggestions

Suggestions when using TensorFlow:

1) Paying attention to limit the available memory of models because TensorFlow will take up all available memory of system if the available memory is not set.

Suggestions when using PyTorch:

1) Avoiding using the input resolution from 416*416 to 480*480, because the speed in this range is even lower than the speed of higher input resolution.

2) Preparing enough swap space when the resolution is bigger than 416*416.

Suggestions when using TensorRT:

1) TensorRT is recommended for deployment because the performance is better than TensorFlow and PyTorch.
2) Deploying the model with the 16-bit quantization techniques because it will improve running time and reduce memory usage without reducing accuracy. Taking 416*416 (resolution) as an example, the 16-bit quantization model has 40% less end-to-end time and 39% less memory than the original model.
3) Using the resolution closest to the picture as the input will speed up by reducing computational complexity without affecting accuracy. The end-to-end time of 416*256 (resolution) is 41% of 416*416 (resolution) input when using the 16:9 picture.

Suggestions for power consumption:

1) Compared with the TensorRT 32-bit model, using TensorRT 16-bit quantization model to deploy YOLOv3 will save more energy and get faster running speed.
2) If only energy consumption is considered, using the PyTorch model when the resolution is less than 480*480 and using TensorRT 16-bit quantization model when the resolution is greater than 480*480.

VI. Conclusion

In this paper, we analyze the feasibility of the edge AI optimization techniques on the low-cost, low-power edge. Four experiments were designed to explore whether NVIDIA Jetson Nano could meet the real-time requirement when running YOLOv3 on Jetson Nano. Experiment 1 compared the performance of three deep learning frameworks, TensorFlow, PyTorch, and TensorRT, running YOLOv3 on Jetson Nano. Since the inference time of PyTorch is much faster than TensorFlow and TensorRT except for the post-processing, experiment 2 analyzed the performance bottleneck of post-processing for PyTorch. Experiment 3 leveraged optimization techniques to squeeze the extreme performance of TensorRT as TensorRT performs better than others. We compared the power of running YOLOv3 in experiment 4 and found that TensorRT with the 16-bit model has the lowest power consumption with the fastest running speed and the least memory usage.

After these experiments, we obtained extreme performance and provided suggestions when using the edge AI-oriented frameworks and optimization techniques. The fastest inference time of YOLOv3 on Jetson Nano is 13.1FPS with the TensorRT framework and the 16-bit model quantization technology. The resolution is 320*192, and the picture ratio is 16:9. In addition, we compared the performance of Jetson Nano and Jetson Xavier NX through the same experiments and found that most of the rules on GPU-based edges are similar. By leveraging the optimization techniques on Xavier NX, the running speed increases from 12.6FPS to 30.3FPS.

In conclusion, we found that the low-cost, GPU-based edge devices could meet the requirements of the mainstream AI applications with the help of optimization technology. In the future, we will study more optimization techniques and compare their effectiveness on a variety of edges to provide more systematic deployment recommendations for the community.

References

[1] Martın Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning." In: *OSDI*. Vol. 16. 2016, pp. 265–283.

[2] *From Research to production. https://pytorch.org*. 2019. URL: https://pytorch.org.

[3] *Google Coral.* 2020. URL: https://coral.ai/.

[4] Ramyad Hadidi et al. "Distributed perception by collaborative robots". In: *IEEE Robotics and Automation Letters* 3.4 (2018), pp. 3709–3716.

[5] Huimin Lu et al. "Motor anomaly detection for unmanned aerial vehicles using reinforcement learning". In: *IEEE internet of things journal* 5.4 (2017), pp. 2315–2322.

[6] Sparsh Mittal and Jeffrey S Vetter. "A survey of CPU-GPU heterogeneous computing techniques". In: *ACM Computing Surveys (CSUR)* 47.4 (2015), pp. 1–35.

[7] *NVIDIA Jetson Nano.* 2020. URL: https://developer.nvidia.com/embedded/jetson-nano-developer-kit.

[8] *NVIDIA TensorRT: Programmable Inference Accelerator. https://developer.nvidia.com/tensorrt*. 2019. URL: https://developer.nvidia.com/tensorrt.

[9] *Raspberry Pi 4 Model B+ (Raspberry Pi)*. 2020. URL: https://www.raspberrypi.org/products/raspberry-pi-4-model-b/.

[10] Joseph Redmon and Ali Farhadi. "Yolov3: An incremental improvement". In: *arXiv preprint arXiv:1804.02767* (2018).

[11] Omer Berat Sezer, Erdogan Dogdu, and Ahmet Murat Ozbayoglu. "Context-aware computing, learning, and big data in internet of things: a survey". In: *IEEE Internet of Things Journal* 5.1 (2017), pp. 1–27.

[12] Weisong Shi et al. "Edge computing: Vision and challenges". In: *IEEE internet of things journal* 3.5 (2016), pp. 637–646.

[13] Shi Weisong et al. "Edge computing: state-of-the-art and future directions". In: *Journal of Computer Research and Development* 56.1 (2019), p. 69.

[14] Jun Zhang and Khaled B Letaief. "Mobile edge intelligence and computing for the internet of vehicles". In: *Proceedings of the IEEE* 108.2 (2019), pp. 246–261.

[15] Xingzhou Zhang et al. "OpenEI: An Open Framework for Edge Intelligence". In: *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. July 2019.

[16] Zhi Zhou et al. "Edge intelligence: Paving the last mile of artificial intelligence with edge computing". In: *Proceedings of the IEEE* 107.8 (2019), pp. 1738–1762.

[17] Pengfei Zhu et al. "Vision Meets Drones: Past, Present and Future". In: *arXiv preprint arXiv:2001.06303* (2020).