Exploring TensorRT to Improve Real-Time Inference for Deep Learning

Yuxiao Zhou
Department of Computer Science
Texas State University
San Marcos, TX, USA
y_z37@txstate.edu

Kecheng Yang
Department of Computer Science
Texas State University
San Marcos, TX, USA
yangk@txstate.edu

***Abstract*—Deep learning (DL) has dramatically evolved and become one of the most successful machine learning techniques. A variety of DL-enabled applications have been widely integrated into software systems, including embedded ones. Although having achieved very successful results in accuracy, the large size of deep neural networks could require significant runtime and computing resource consumption. To overcome these drawbacks, TensorRT has been developed and may be incorporated into popular DL frameworks such as PyTorch and Open Neural Network Exchange (ONNX). In this paper, focusing on inference, we provide a comprehensive evaluation on the performances of TensorRT. Specifically, we evaluate inference output validation, inference time, inference throughput, and GPU memory usage. Our results demonstrate that TensorRT is able to significantly improve the inference efficiency metrics without compromising inference accuracy. Furthermore, TensorRT may be adopted via several alternative workflows. Our evaluation also shows the pros and cons of these TensorRT workflows. We analyze that for each workflow and discuss the workflow selection for different application scenarios.**

***Index Terms*—deep learning, real-time inference, learning-enabled embedded systems, TensorRT**

I. Introduction

In the past few years, deep learning (DL) has dramatically evolved and become one of the most successful machine learning techniques. A variety of DL-enabled applications, such as computer vision, natural language processing, autonomous control, *etc.*, have been widely integrated into software systems, including embedded ones. In contrast to classical machine learning methods, which are prone to overfitting when their size is at a large scale, deep networks can achieve high accuracy with large and over-parameterized models. According to the ImageNet classification leader board provided by the papers with code [1], the parameter number in the state-of-the-art model for image classification has been increased from 61 million to 2100 million since 2013. A similar trend occurs in natural language processing (NLP). Since 2018, alone, we have seen the emergence of Bidirectional Encoder Representations for Transformers (BERT) and its variants [2]. The BERT base model, published by researchers at Google AI Language [6], contains over 109 million parameters, while the BERT large contains 345 million parameters.

Although such large, sometimes even huge, deep neural networks often result in very successful accuracy and precision, not only their training but also their inference runtime could become extremely slow and sluggish. Furthermore, such large model architectures often require significant amount of computing resources even just to make inferences. However, many applications and systems, especially embedded ones, require inference in real-time while utilizing a limited amount of hardware resources due to size, weight, power, and cost (SWaP-C). For example, autonomous vehicles need to process data from different sensors such as cameras and lidars, and make proper control decisions promptly. Also, a video surveillance system must perform real-time video analysis to detect abnormal activities such as accidents, burglaries, explosions, fighting, robberies, and other violent events. On the other hand, for privacy and reliability reasons, such computation is often required to be done at the embedded platform with limited computing resources.

This inference computation barrier opens a massive gap between the success of neural networks and their applicability in many real-world scenarios. To mitigate this gap, several technologies have been proposed and developed, including low-power, highly efficient SoC chips specialized for deep learning inference such as Google's Tensor Processing Unit (TPU) [9] and Intel's Vision Processing Unit (VPU) [13], model compression methods such as quantization [27], the pruning [33] and neural architecture search (NAS) [15] of deep learning models for resource-constrained devices, and lightweight models with reduced weights and parameters such as MobileNets [11] and SqueezeNet [12].

Despite such efforts and advances, the common, general-purpose DL framework, such as PyTorch [17], is not particularly optimized for the computing resource and time consumption of inferences. To address this issue, NVIDIA published the TensorRT, a high-performance DL inference engine for production deployments of deep learning models. TensorRT can improve the inference throughput and efficiency, enabling developers to optimize neural network models trained on all major frameworks such as PyTorch and TensorFlow, and deploy them to various devices such as embedded platforms and automotive product platforms [4].

In this paper, we examine the effectiveness of TensorRT

by comparing it to the vanilla (without TensorRT) PyTorch framework. In particular, there are three workflows to integrate the TensorRT engines with PyTorch-compatible models. We evaluate the performance of these three TensorRT integration workflows under a variety of workloads. We identify the performance bottlenecks in the inference using TensorRT. And we find that there is still room for fully maximizing the deep learning performance in term of GPU memory utilization.

Contribution. Our main goal is to bring attention to the emerging TensorRT for learning-enabled real-time embedded systems. In such systems, the inference often needs to be performed online and using embedded hardware and therefore may be subject to real-time constraints and computing resource limitations. We provide a comprehensive evaluation of the inference performances of TensorRT engines which are converted and deployed from different workflows via various software tools. Specifically, we evaluate inference output validation, inference time, inference throughput, and GPU memory usage for each workflow. Our results demonstrate that TensorRT is able to significantly improve the inference efficiency metrics without compromising inference accuracy. Nonetheless, TensorRT may be adopted via several alternative workflows. Our evaluation also shows the pros and cons of these TensorRT workflows. We analyze that for each workflow and discuss the workflow selection for different application scenarios.

Organization. The rest of this paper is organized as follows: Sec. II provides a background overview of PyTorch, ONNX, and TensorRT. Sec. III describes the methodology for our experiments, including evaluated models, workflows, and performance measuring. Experiment results and discussion are provided in Sec. IV and Sec. V. Sec. VI provided an overview of exsiting work. We conclude our work in Sec. VII.

II. Overview of Deep Learning Frameworks and TensorRT

In this section, we provide an overview of DL framework, compiler and runtime that we used in this research.

A. PyTorch

PyTorch [17] is an open-source machine learning framework that accelerates the path from research prototyping to production deployment. It is a Python package being primarily used as a deep learning research platform that aims at providing maximum flexibility and speed.

PyTorch supports the operation of Tensors (multi-dimension array) on both CPU and GPU, and this may accelerate the computation by a significant amount. PyTorch provides a variety of tensor routines to fit different scientific computation needs, including mathematical operations and linear algebra.

For most frameworks, the user has to build a neural network and reuse the same structure repeatedly. PyTorch uses reverse-mode auto-differentiation, a technique allowing the user to change the way the network behaves arbitrarily without any significant overheads.

PyTorch is integrated with acceleration libraries such as Intel MKL and NVIDIA (cuDNN, NCCL) to maximize speed. Therefore, it is quite fast to run network of varying sizes. Also, PyTorch is designed to use memory efficiently compared to other alternatives, so that it enables the user to train extremely large deep learning models.

B. ONNX

The Open Neural Network Exchange (ONNX) [8] is an open-source artificial intelligence ecosystem. It is an open standard established by many technology companies and research organizations for representing machine learning algorithms and software tools to promote innovation and collaboration in the AI sector.

ONNX targets at interoperability. It defines a common set of operators — the building blocks of machine learning and deep learning models — and a common file format, which make it possible for AI developers to use models with various frameworks, tools, compilers, and runtimes. ONNX supports multiples software frameworks such as PyTorch, TensorFlow, Caffe2 and Apache MXNet.

ONNX also facilitates model optimization on different hardware devices. Users can deploy an ONNX model using runtimes designed for particular hardware to accelerate the inference execution on the device.

C. TensorRT

TensorRT is a Software Development Kit (SDK) for high-performance deep learning inference. It is a part of NVIDIA CUDA X AI Kit. It comes with a deep learning inference optimizer and runtime that delivers low latency and high throughput for deep learning inference.

TensorRT performs six types of optimizations to reduce latency and increase the throughput of deep learning models: 1. Weight and activation precision calibration: maximize throughput by quantizing model to 8-bit integer while keeping the same level of accuracy. 2. Layer and tensor fusion: optimizes the use of GPU memory and bandwidth by fusing nodes in a kernel vertically or horizontally(or both), which reduces the overhead and the cost of reading and writing the tensor data for each layer. 3. Kernel auto-tuning: provides kernel-specific optimization which selects the best layers, algorithms, and optimal batch size based on the target GPU platform. 4. Dynamic tensor memory: improves memory re-usage by allocating memory to the tensor only for the duration of its usage. This helps in reducing memory consumption and avoiding allocation overhead for efficient execution. 5. Multi-stream execution: processes multiple input streams in parallel. 6. Time fusion: optimizes recurrent neural networks (RNN) over time steps with the dynamically generated kernel.

The ability to take a variety of DL models as input makes TensorRT applicable to a wide range of artificial intelligence (AI) based applications such as computer vision, automatic speech recognition, natural language understanding (BERT), text-to-speech, and recommender systems. TensorRT can provide deployment-ready inference engines for multiple computer vision models used in autonomous driving systems. It

can also deliver real-time,low-latency video analytics at scale for hyper-scale data centers.TensorRT optimizes DL model so it can achieve accurate and real-time inference on edge devices and Internet of Things (IoT).

III. Methodology

A. Neural Network Models to Evaluate

Image classification is a fundamental task in computer vision. It attempts to comprehend an entire image as a whole, and the goal of it is to classify the image by assigning it to a specific label. Convolutional Neural Network (CNN) is a special multi-layer neural network designed to recognize visual patterns directly from pixel images. It has been widely used for image classification because it can automatically detect significant features without human supervision. In this paper, we focus on the TensorRT inference of three types of image classification CNN models: Residual neural networks [10] (ResNet) models, MobileNet, and SqueezeNet. The ResNet model was firstly proposed in "Deep Residual Learning for Image Recognition". It uses the skip connection to improve the performance and the convergence of deep neural networks. There are many variants of ResNet architectures that run on the same concept but with different number layers. Our experiment is tested against five variants of ResNet: ResNet-18, ResNet-34, ResNet-50, ResNet-101 and ResNet-152.

We also experiment with two network architectures that are well suited for platform with limited resources: SqueezeNet and MobileNet. They both apply smart tricks in their architecture to keep the models small and efficient without sacrificing too much accuracy. Our test includes two versions of Squeezenet, where SqueezeNet 1.1 has even less computation and fewer parameters than SqueezeNet 1.0.

B. Workflows

We designed our experiments to evaluate the performance of all possible workflows to speed up the PyTorch DL model inference using TensorRT. Fig. 1 provides an overview of our experiment workflows and the software tools used in each stage.

Pre-trained model loading. As shown in Fig. 1, all workflows share this starting step: pre-trained model loading. In this stage, we load the pre-trained models using the PyTorch torch.hub. The pre-trained model includes the definition of the model architecture and pre-trained weights. The model architecture provides the working parameters, such as the number, size and type of layers in a neural network. The pre-trained weight, often trained on an open dataset such as ImageNet, is downloaded to a cache directory when instancing a pre-trained model.

Inference implementation. To implement the inference in an already trained model, there are multiple alternative *workflows* to choose. In this work, we compare the conventional, default workflow in PyTorch that does not involve TensorRT at all (denoted as W0) with three workflows that do integrate TensorRT (tagged by W1, W2, W3, respectively). These four workflows we evaluate in this work are explained in more details as follows.

W0: PyTorch Default

By default, the pre-trained models are loaded on CPU. We transfer the models from CPU to GPU to ensure the inference executes on GPU. The same logic applies to the input data. We then perform inference through the PyTorch model by running a Python API [17].

W1: Torch-TensorRT

In this workflow, we accelerate inference using Torch-TensorRT, an integration of PyTorch with NVIDIA TensorRT. Torch-TensorRT acts as an extension to TorchScript. It optimizes and executes compatible subgraphs, letting PyTorch execute the remaining graph. The Torch-TensorRT compiler's architecture consists of three phases to optimize compatible subgraphs: 1) reducing the TorchScript module, 2) conversion and 3) execution.

During the first phase, Torch-TensorRT [20] simplifies implementations of frequent operations to representations that map more directly to TensorRT. In the conversion phase, Torch-TensorRT identifies TensorRT compatible portion of graphs and translates them to TensorRT operations. The remaining nodes stay in TorchScripting, forming a hybrid graph returned as a standard TorchScript module [28]. In the last phase when executing the compiled module, Torch-TensorRT sets up the TensorRT engine ready for execution. During execution, the TorchScript interpreter calls the TensorRT engine, which performs inference with inputs passed from the interpreter. The engine then returns the running results back to the interpreter after the inference completed [28].

Torch-TensorRT provides a simple Python API for both model compilation and optimization. To compile the PyTorch model with Torch-TensorRT, we only need to provide the PyTorch model and inputs to Torch-TensorRT API that returns an optimized TorchScript module to run. Input is a list of torchtensorrt Input classes which define input's shape, data type, and memory format. We specify operating precision as floating point 32 as our hardware doesn't support half precision. After compilation, we save the module as a .ts file for later use. We then run inference on top of the optimized TorchScript module [28].

W2: ONNX to TensorRT Conversion

There are three steps in this workflow: 1) exporting the PyTorch model to the ONNX file, 2)TensorRT engine building 3)TensorRT engine deployment in addition to model loading.

In the first step, We use the torch.onnx.export() function in PyTorch library exporting the PyTorch model to ONNX files. The torch.onnx.export() function takes an input tensor to run the model tracing its execution and then exports the traced model to an ONNX file. Note that the input size is fixed in the exported ONNX graph for all inputs' dimensions by default. To export an ONNX model that accepts input tensor of dynamic batch size, we need to specify the first dimension as dynamic in the dynamic_axes parameter in torch.onnx.export() function. We export PyTorch models to ONNX files with both static shape and dynamic shape. We prefer the latter option

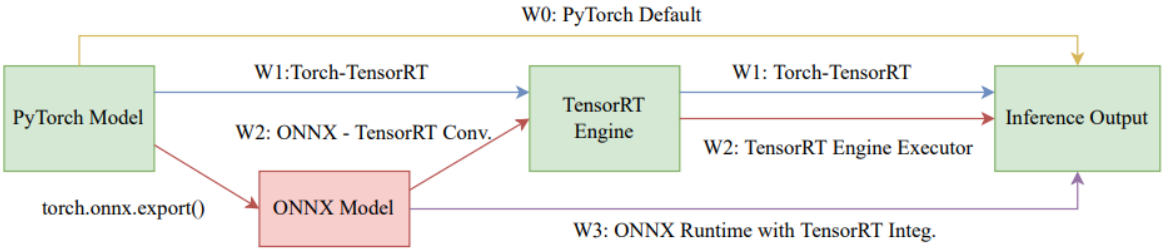

Fig. 1. An illustration of experiment workflows.

since it has the obvious advantage in GPU memory usage and we only need to create one ONNX file for multiple input batch size [19].

We build the TensorRT engine with an ONNX model using the TensorRT Python API. The building stages involve operations such as creating a network definition, importing a model through ONNX parser, and building a TensorRT engine with a builder [25].

TensorRT engine inference consists of the following six sub-steps: 1) create an inference execution context, 2) allocate memory for input and output on CUDA device 3) transfer input data from host into input memory allocated on CUDA device, 4) perform TensorRT engine inference using the asynchronous execute API, 5)transfer the output back into host memory and 6)synchronize the stream used for data transfers and inference execution to make sure all operations are finished [26].

W3: ONNX Runtime with TensorRT Integrated

This workflow also starts with exporting a model from PyTorch to ONNX using torch.onnx.export() function in PyTorch library. After obtaining the ONNX model, we run the model inference using the ONNX Runtime execution provider(EP).

ONNX Runtime works with various hardware acceleration libraries through its EP framework that enables flexibility to deploy ONNX models in different environments and optimize the execution by taking advantage of the computation capabilities of the platform. ONNX Runtime works with the EP(s) using an API to allocate specific nodes or sub-graphs for execution by the EP library in supported hardware [22]. The EP libraries are pre-installed in the execution environment process and execute the ONNX sub-graph on the hardware. With the TensorRT EP, the ONNX Runtime delivers better inference performance on the same hardware in comparison to normal CUDA acceleration [22].

We load and run the ONNX model in ONNX Runtime through the InferenceSession class. When initializing the InferenceSession we explicitly register TensorRT EP to it so the model is executed through TensorRT engine [21].

To reduce memory copy between CPU and devices during execution, ONNX Runtime introduces the notation of IOBinding. With IOBinding, we can copy the input to the GPU and pre-allocate the output on the GPU before model execution. In our experiments, we run the created InferenceSession with IOBindng [23].

C. Inference Performance Measuring

Output validation. During the inference, we feed models with four-dimension Pytorch tensors which can be thought of as four-dimension arrays filled with random numbers from a uniform distribution on the interval [0, 1). The shape of the input tensor is determined by the batch size and model architecture. For all image classification models, we set the shape of input tensors as [batch size, 3, 224, 224], where 3 is the number of channels in the input image while last two numbers represent the width and height of input images. All image classification models are pre-trained on ImageNet [30] datasets which span on 1000 object class. Therefore, the outputs of these model, which contain probability values for all classes, have a shape of [batch size, 1000].

In the ideal condition, we want the absolute differences between outputs from each PyTorch model and its TensorRT engine as minimal as possible. We copy the output Tensors from GPU to CPU and then convert them into NumPy arrays. We apply a Numpy function, numpy.isclose() [5] to compare the output numpy arrays. The input parameters for this function include two arrays to compare, relative tolerance and absolute tolerance. The relative difference (rtol * abs(array 2)) and the absolute difference (atol) are added together to compare against the absolute difference between array 1 and array 2. In our experiment, we set the relative tolerance (rtol) to 0.0001 and use the default absolute tolerance (atol): 1e-8. The function numpy.isclose() returns a boolean array where two arrays are element-wise equal within a tolerance. We report the percentage of non-equal elements by dividing the number of non-equal elements by the size of the array.

Execution time measuring. In our experiments, we measured the end-to-end execution time of the forward inference cycle. This timing does not include any data retrieval, model initialization, and pre-processing of input. To accurately measure the inference time of the evaluated model, we ran a few warm-up operations since the execution speed usually takes some time to reach the maximum speed. We first run the model inference for 50 hot runs and track the execution time following the warm-up steps [31]. The execution time in each step is measured using 'time' [7] function in the 'time' module in Python. We record two timestamps using the 'time' function; one is just before the inference execution and the other is when the inference completes. The inference time is recorded as the difference between these two timestamps.

Model	ResNet18	ResNet34	ResNet50	ResNet101	ResNet152	MobileNet	SqueezeNet1.0	SqueezeNet1.1
Torch-TensorRT	256	256	128	128	64	256	256	256
ONNX - TensorRT Conv.	256	256	256	256	256	256	256	256
ONNX Runtime w/ TensorRT Integ.	256	256	256	128	128	256	256	256

TABLE I
MAXIMUM BATCH SIZE FOR EACH NETWORK ARCHITECTURE AND WORKFLOW COMBINATION

Throughput measuring. Throughput is used to indicate the amount of data that can be processed or the number of executions of a task that can be completed in a given period such as 1 second. To calculate the throughput, we divide the number of inference inputs by the processing time.

As described in this paper, 'Deep Learning at Scale on NVIDIA V100 Accelerators' [32], increasing the batch size may increase throughput. So, we measure the inference throughput using different batch sizes. We start our testing with batch size as one and multiplying by 2 until it reaches 256 or the maximum batch size that doesn't cause GPU memory errors.

Many PyTorch users have reported converting a PyTorch model to ONNX using torch.onnx.export function may raise CUDA out of memory error as the converter may create new tensors may as copies of the tensors in the PyTorch model [16]. We face similar memory issues if we explicitly set a large batch size (64 or 128) on an ONNX model during the model conversion. This issue is solved by setting the batch size, the first dimension of the input Tensor, as dynamic axes when exporting a PyTorch to ONNX. Table I lists the maximum batch size for each inference speeding up workflow.

For Torch-TRT, and TensorRT the inference engine is generated independently for each batch size as different types of kernels may be used, while PyTorch and ONNX Runtime make such decisions at run-time, without hardware probing.

GPU memory usage measuring. We use a Python module GPUtil [24] to estimate the GPU memory consumption during the inference. This module acquires the GPU status from NVIDIA System Management Interface (nvidia-smi) [3], a command line utility which reports the GPU device states. We keep the model inference as the only process that are using GPU memory each time when tracking the GPU utilization to ensure the GPU utilization estimates are as accurate as possible.

D. Hardware Specifications

While desktop- and server-level GPUs nowadays, such as Nvidia GeForce RTX 3090, may be equipped with 24 GB or more GPU memory, GPUs on portable and embedded devices are typically subject to more restrictive memory constraints. Such restrictions could impose limitations on both the complexity of neural networks and the size of the data samples that can be processed. To this end, we carry out our experiments on a Dell Precision 3551 mobile workstation laptop equipped with the NVIDIA Quadro P620 with 4 GB memory.

E. Software Specifications

Each software tool used in this experiment has its support matrices including the specific version of platforms, features, and operating systems. For example, TensorRT 8.2.4 is only compatible with cuDNN 8.2.1 while TensorRT Execution Provider in ONNX Runtime 1.10 requires CUDA to be 11.4. We need to verify the compatibility between these libraries before performing the test. Table II summarizes the software tools and their versions used in our experiments.

CUDA	ONNX	ONNX Runtime	PyTorch	TensorRT	Torch-TensorRT
11.4	1.12.0	1.10.0	1.11.0	8.2.4	1.1.0

TABLE II
SOFTWARE TOOLS AND VERSIONS IN OUR EXPERIMENTS

IV. EXPERIMENT RESULTS

We present our experiments results in the following four subsections: inference accuracy, inference execution time, inference throughput, and GPU memory usage.

A. Inference Output Validation

We verify the TensorRT engine inference output by comparing them with the raw outputs from the PyTorch models. Table III presents the percentage of non-matching elements in two output Tensors where a relative threshold 1.0×10^{-3} and an absolute threshold 1.0×10^{-8} are applied. Three TensorRT engine workflows show similar performance. They all achieve the best results on SqueezeNet while the inference results on MobileNet contain more non-matching elements than the other network architectures. With the number of layers in ResNet models increasing, the difference between the inference outputs from the PyTorch model and TensorRT engines is reduced.

Observation 1. Adopting TensorRT incurs minimum inference outputs difference compared to the default PyTorch framework. In other words, the benefits TensorRT brings in are *not* at the cost of noticeable sacrifice in inference accuracy.

B. Inference Execution Time

Minimum forward execution time is often targeted for time-critical applications, where having minimal latency has more priority than having higher throughput. For such deployments, the batch size is often set to minimum.

Our results in Fig. 2 show that ONNX Runtime with TensorRT Integrated and ONNX - TensorRT Conversion have comparable execution times on all network architecture. They both have better performance than the Torch-TensorRT.

In the Table IV, we list the maximum inference speed up that can be obtained through all workflows. It is observed that the Resnet variants deployed with the TensorRT engine perform 1.6-1.7 faster than PyTorch running on the same GPU.

Model	ResNet18	ResNet34	ResNet50	ResNet101	ResNet152	MobileNet	SqueezeNet1.0	SqueezeNet1.1
Torch-TensorRT	0.49%	0.48%	0.32%	0.34%	0.26%	0.99%	0.000003%	0.0001%
ONNX - TensorRT Conv.	0.48%	0.43%	0.34%	0.32%	0.25%	1.07%	0.002000%	0.0180%
ONNX Runtime w/ TensorRT Integ.	0.48%	0.45%	0.34%	0.32%	0.25%	1.08%	0.000034%	0.0200%

TABLE III
PERCENTAGE OF ON-EQUAL ELEMENT IN OUTPUT TENSORS

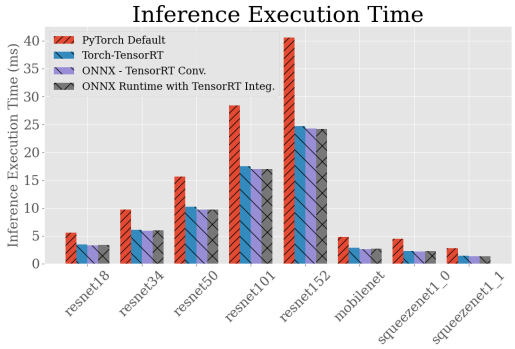

Fig. 2. Mean inference execution time for all network architectures.

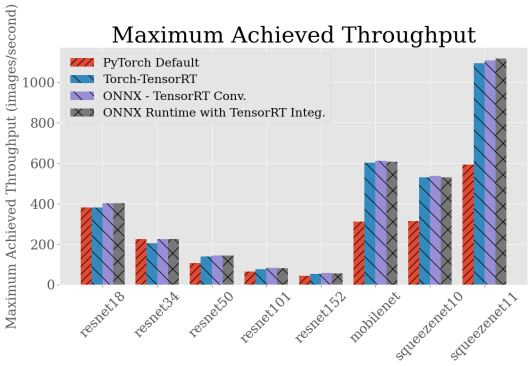

Fig. 3. Maximum achieved throughput for all network architectures.

For efficient architectures such as MobileNet and SqueezeNet, the TensorRT engine can make about 2 times speed up.

Observation 2. Applying TensorRT can significantly improve the inference time. Light-weighted model architecture, such as MobileNet and SqueezeNet, can benefit even more from TensorRT.

C. Inference Throughput

Inference throughput is important for applications that involve multiple inference operations in a single time frame. In certain instances, delayed batch processing is acceptable. An application can benefit from increased throughput by increasing the inference batch size.

Fig. 3 shows maximum achieved throughput using each workflow with varying batch size. In maximum throughput calculation, all available batch sizes are considered. As in the previous execution evaluation, ONNX Runtime with TensorRT Integrated have similar performance with ONNX to TensorRT Conversion. Torch-TensorRT has the overall lowest throughput.

Fig. 4 shows a subset of the resulting throughput-vs-batch size curves. It is observed that increasing the batch size increases throughput remarkably until performance converges at a certain batch size, typically 32. A further increase provides a limited gain in throughput. This occurs because further parallelization inside the GPU is not possible.

Observation 3. By applying TensorRT, improvement on inference throughput can be obtained in most cases and may be very significant for certain network architectures. Inference throughput increases with batch size increases until reaches hardware limits.

D. GPU Memory Usage

As the networks go wider and deeper, the limited GPU memory becomes a bottleneck restricting the size of networks

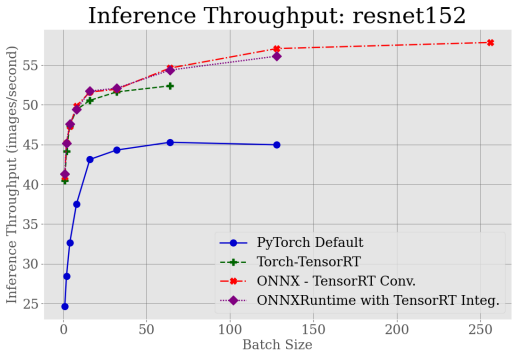

Fig. 4. Inference throughput for ResNet152 with varying batch size.

to be inferenced. It is important to track the GPU memory consumption by these three workflows.

As shown in both Fig. 5 and Fig. 6, the ONNX to TensorRT Conversion consumes the least GPU memory while ONNX Runtime with TensorRT Integrated has the largest memory usage. It is noteworthy there is a positive correlation between GPU memory usage and batch size for all three workflows for all models. This is not always the case for PyTorch. The memory reserved by PyTorch includes memory occupied by tensors and the cached memory managed by the caching allocator [18]. While the occupied memory increases with the increasing batch size, we find that the cached memory may not be affected as much by the batch size in some cases. In our experiment, the GPU memory usage remains the same after the batch size is increased to 4 for ResNet50 with PyTorch.

Observation 4. TensorRT, especially via the workflow W2: ONNX to TensorRT Conversion, utilizes GPU memory in a significantly more efficient and scalable manner.

V. DISCUSSION

Due to different requirements and priorities, the use of a single performance metric for all inference scenarios is not

Model	ResNet18	ResNet34	ResNet50	ResNet101	ResNet152	MobileNet	SqueezeNet1.0	SqueezeNet1.1
Speed Up	1.67	1.64	1.61	1.67	1.68	1.83	2.00	2.05

TABLE IV
MAXIMUM SPEED UP FOR EACH MODEL ARCHITECTURE

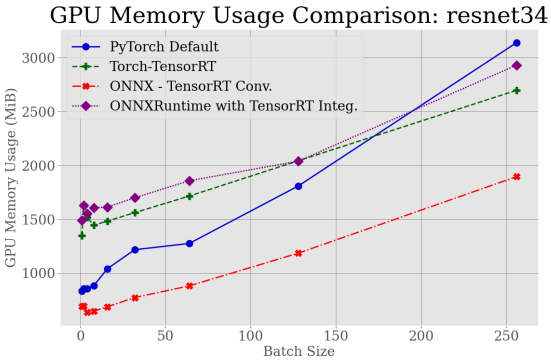

Fig. 5. GPU memory usage for ResNet34.

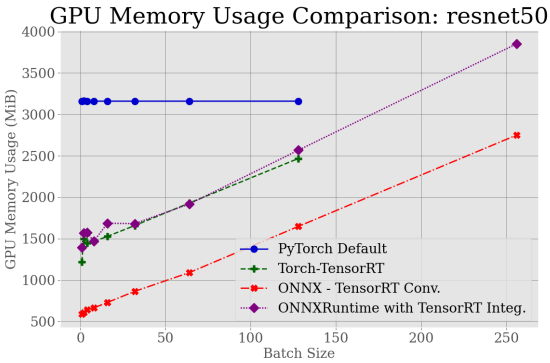

Fig. 6. GPU memory usage for ResNet50.

feasible. Overall, the ONNX to TensorRT Conversion delivers the highest throughput and smallest latency, although there are cases where it is outperformed by others. Its GPU memory consumption is also less than the other two workflows. However, to benefits from the TensorRT often involves the conversion from ONNX model first. The ONNX conversion is all-or-nothing, meaning all operations in the model must be supported by TensorRT (or the user must provide custom plug-ins for unsupported operations). Once the ONNX model got exported, one had to then write a TensorRT client application, which would feed the data into the TensorRT engine. This increases the burden on the user and reduces the likelihood of reproducibility.

Our results show that ONNXRuntime with TensorRT Integrated can deliver less latency and more throughput on average, as compared to the Torch-TensorRT. But it consumes most GPU memory during the inference among three workflows. The high GPU memory usage can impose an upper limit on the inference throughput. Since ONNX targets interoperability, it provides converters for a variety of frameworks such as Keras, TensorFlow and Apache MXNet. Therefore, the ONNX Runtime with TensorRT Integrated can be used together with many other frameworks in addition to PyTorch.

The Torch-TensorRT has the lowest overall throughput but consumes less GPU than ONNX Runtime with TensorRT Integrated. It provides a user-friendly API which accelerates the inference by one line of the code. While the other workflows require several sub-stages such as memory allocation and management of data transferring between host and device, the workflow for Torch-TensorRT is much simpler. Also, it doesn't involve any explicit model conversion. Model conversion may cause blocking issues if there is no valid mapping between some operators in different network frameworks.

Considering our results and observations, it is impossible to conclude that one of the evaluated workflows is superior in all aspects. Latency and throughput characteristics provided in Fig. 2 and Fig. 3 show which workflows have better performance for certain network architectures. However, this comparative result does not only depend on workflow selection. Application requirements may require specific functionality, which may be supported at various levels with different frameworks and tools. Furthermore, dependency on additional software libraries or tools and ease of use also need to be considered in the workflow selection process. According to our experience, this decision requires a well-defined scenario, which includes available hardware and resources, required capabilities, range of desired throughput and execution time.

VI. Related Work

Xu *et al.* use quantifies the inference performance using TensorRT. They compared the TensorRT inference for Resnet50 with INT8 vs FP32, which shows that INT8 mode is -3.7x faster than FP32. The experiments that they did also concluded that INT8 can also achieve the comparable accuracy with FP32 [32]. Jeong *et al.* proposed present a TensorRT-based parallelization method that uses both GPU and NPUs to maximize the throughput of a single DNN application. They reported a 81% -391% performance improvement over the baseline inference that is performed on a GPU [14]. Ulker *et al.* presented an evaluation of the inference performance of deep learning software tools using CNN architectures for multiple hardware platforms. They benchmark these hardware-software pairs for a broad range of network architectures, inference batch sizes, and floating-point precision, focusing on latency and throughput. Their results reveal that TensorRT delivers minimum average execution time and highest throughput for the network models that can be translated into TensorRT engines [31]. In Shin and Kim's recent work, they introduce a performance inference method that fuses the Jetson monitoring tool with TensorFlow and TRT source code on the Nvidia Jetson AGX Xavier platform. The CPU utilization, GPU utilization, object accuracy, latency, and power consumption of the deep learning framework were also compared and analyzed [29].

VII. Conclusion

This paper presents an extensive comparative inference performance evaluation of a set of workflows accelerating PyTorch models with TensorRT on hardware platforms with limited resource. We focus on the local computation of CNN model inference. Based on our evaluation results, we discuss framework performance in terms of latency, throughput and GPU memory usage characteristics.

We supplemented our interpretation with an investigation of weakness and strength in each workflow. Our discussions include workflow selection for common scenarios in deep learning inference deployment for computer vision tasks. Results show that ONNX to TensorRT Conversion has the best overall performance for improving PyTorch model inference. ONNX Runtime with TensorRT Integrated can deliver less latency and more throughput on average, as compared to the Torch-TensorRT. The Torch-TensorRT provides a user-friendly API which accelerates the inference by one line of the code.

We conclude that none of these inference workflows as the best choice in all scenarios. We suggest using ONNX to TensorRT Conversion if the application requires the high inference performance with limited computation resource. ONNX Runtime with TensorRT Integrated can be used to accelerating inference in system consisted of multiple frameworks such as PyTorch, TensorFlow and Apache MXNet. We may utilize Torch-TensorRT for application where the fast development is first priority.

Future work. In the future work, we will include the inference performance evaluation with reduced precision such as the half floating-point precision and 8-bit integer. The lower precision can reduce the computational load and also speed up the inference while keep an acceptable accuracy. We will try to experiment with more model architectures such as Recurrent Neural Networks (RNN) and generative adversarial network (GAN) in addition to CNN. We would like to employ multiple threading to further improve the inference performance of TensorRT engine. We also interested in improving the GPU memory consumption during ONNX model conversion.

Acknowledgment

This work is supported in part by NSF grant CNS-2104181, and start-up and REP grants from Texas State University.

References

[1] Meta AI. Image classification on imagenet. Online at https://paperswithcode.com/sota/image-classification-on-imagenet.

[2] Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, pages 610–623, 2021.

[3] NVIDIA Corporation. Nvidia system management interface. Online at https://developer.nvidia.com/nvidia-system-management-interface.

[4] NVIDIA Corporation. Nvidia tensorrt. Online at https://developer.nvidia.com/tensorrt.

[5] NumPy Developers. numpy.isclose. Online at https://numpy.org/doc/stable/reference/generated/numpy.isclose.html.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[7] Python Software Foundation. time — time access and conversions. Online at https://docs.python.org/3/library/time.html.

[8] The Linux Foundation. Open neural network exchange. Online at https://onnx.ai/.

[9] Google. Cloud tensor processing units (tpus). Online at https://cloud.google.com/tpu/docs/tpus.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[11] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[12] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[13] Intel. Intel® vision accelerator design with intel® movidius™ vision processing unit (vpu). Online at https://www.intel.com/content/www/us/en/developer/topic-technology/edge-5g/hardware/vision-accelerator-movidius-vpu.html.

[14] EunJin Jeong, Jangryul Kim, Samnieng Tan, Jaeseong Lee, and Soonhoi Ha. Deep learning inference parallelization on heterogeneous processors with tensorrt. *IEEE Embedded Systems Letters*, 14(1):15–18, 2021.

[15] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European conference on computer vision (ECCV)*, pages 19–34, 2018.

[16] Piotr Marcinkiewicz. Onnx export doubles ram usage. Online at https://github.com/pytorch/pytorch/issues/61263.

[17] Meta. From research to production. Online at https://pytorch.org/.

[18] Meta. Memory management. Online at https://pytorch.org/docs/stable/notes/cuda.html#memory-management/.

[19] Meta. (optional) exporting a model from pytorch to onnx and running it using onnx runtime. Online at https://pytorch.org/tutorials/advanced/super_resolution_with_onnxruntime.html.

[20] Meta. Torch-tensorrt. Online at https://pytorch.org/TensorRT/.

[21] Microsoft. Get started with ort for python. Online at https://onnxruntime.ai/docs/get-started/with-python.html.

[22] Microsoft. Onnx runtime execution providers. Online at https://onnxruntime.ai/docs/execution-providers/.

[23] Microsoft. Onnx runtime performance tuning. Online at https://onnxruntime.ai/docs/performance/tune-performance.html.

[24] Anders Krogh Mortensen. Gputil. Online at https://github.com/anderskm/gputil.

[25] NVIDIA. Nvidia tensorrt developer guide — nvidia docs. Online at https://docs.nvidia.com/deeplearning/tensorrt/pdf/TensorRT-Developer-Guide.pdf.

[26] NVIDIA. tutorial runtime. Online at https://github.com/NVIDIA/TensorRT/blob/main/quickstart/SemanticSegmentation/tutorial-runtime.ipynb.

[27] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. *arXiv preprint arXiv:1802.05668*, 2018.

[28] Ashish Sardana. Accelerating inference up to 6x faster in pytorch with torch-tensorrt. Online at https://developer.nvidia.com/blog/accelerating-inference-up-to-6x-faster-in-pytorch-with-torch-tensorrt/.

[29] Dong-Jin Shin and Jeong-Joon Kim. A deep learning framework performance evaluation to use yolo in nvidia jetson platform. *Applied Sciences*, 12(8):3734, 2022.

[30] Princeton University Stanford Vision Lab, Stanford University. Imagenet. Online at https://www.image-net.org/.

[31] Berk Ulker, Sander Stuijk, Henk Corporaal, and Rob Wijnhoven. Reviewing inference performance of state-of-the-art deep learning frameworks. In *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems*, pages 48–53, 2020.

[32] Rengan Xu, Frank Han, and Quy Ta. Deep learning at scale on nvidia v100 accelerators. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 23–32. IEEE, 2018.

[33] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.