applied sciences

MDPI

Article

A Deep Learning Framework Performance Evaluation to Use YOLO in Nvidia Jetson Platform

Dong-Jin Shin [1] **and Jeong-Joon Kim** [2,*]

[1] Department of Computer Engineering, Anyang University, Anyang-si 14058, Korea; djshin@gs.anyang.ac.kr
[2] Department of ICT Convergence Engineering, Anyang University, Anyang-si 14058, Korea
* Correspondence: jjkim@anyang.ac.kr

Abstract: Deep learning-based object detection technology can efficiently infer results by utilizing graphics processing units (GPU). However, when using general deep learning frameworks in embedded systems and mobile devices, processing functionality is limited. This allows deep learning frameworks such as TensorFlow-Lite (TF-Lite) and TensorRT (TRT) to be optimized for different hardware. Therefore, this paper introduces a performance inference method that fuses the Jetson monitoring tool with TensorFlow and TRT source code on the Nvidia Jetson AGX Xavier platform. In addition, central processing unit (CPU) utilization, GPU utilization, object accuracy, latency, and power consumption of the deep learning framework were compared and analyzed. The model is You Look Only Once Version4 (YOLOv4), and the dataset uses Common Objects in Context (COCO) and PASCAL Visual Object Classes (VOC). We confirmed that using TensorFlow results in high latency. We also confirmed that TensorFlow-TensorRT (TF-TRT) and TRT using Tensor Cores provide the most efficiency. However, it was confirmed that TF-Lite showed the lowest performance because it utilizes a GPU limited to mobile devices. Through this paper, we think that when developing deep learning-related object detection technology on the Nvidia Jetson platform or desktop environment, services and research can be efficiently conducted through measurement results.

Keywords: deep learning; embedded system; Nvidia Jetson platform; TensorFlow; TensorRT; YOLO

check for updates

Citation: Shin, D.-J.; Kim, J.-J. A Deep Learning Framework Performance Evaluation to Use YOLO in Nvidia Jetson Platform. *Appl. Sci.* **2022**, *12*, 3734. https://doi.org/10.3390/app12083734

Academic Editors: Bhanu Shrestha, Seongsoo Cho and Changho Seo

Received: 16 March 2022
Accepted: 6 April 2022
Published: 7 April 2022

1. Introduction

With the development of information and communication technology, various fields such as big data, Internet of Things (IoT), and AI are developing [1]. Machine learning (ML), related to AI, is a technology that computers can learn on their own to create and predict models. Furthermore, deep learning is a field of machine learning using deep neural network theory, using the principle of neural network corresponding to the human brain [2]. The fields of deep learning can be largely divided into image classification, object detection, natural language processing, and voice/speech recognition [3]. In particular, object detection can be divided into one-stage detectors and two-stage detectors, in which a one-stage detector has a YOLO, SSD-based model, and a two-stage detector has an R-CNN-based model. Among them, YOLO is a one-stage detector model that is very fast with a simple processing process, but has relatively low accuracy for small objects [4].

Deep learning can make inferences through computation on the CPU, but it requires GPU hardware for performance benefits. To this end, Nvidia embedded a core called Computed Unified Device Architecture (CUDA) in the GPU, which can be calculated faster than the CPU by utilizing GPUs during deep learning. Therefore, it is possible to perform computations on the GPU by using the CUDA Cores located in the GPU, so that the learning and result inference required for object detection using deep learning can be performed more efficiently [5,6].

Recently, as IoT and embedded environments are in the spotlight, not only are deep learning-related studies being conducted in desktop-like environments, but deep learning-related studies in embedded environments are also continuously being conducted. The

Jetson platforms sold by Nvidia as embedded systems are small modules, very small compared to desktops, with slightly less performance and accuracy. However, they have good power efficiency and provide at least four times higher performance when compared to CUDA cores, through built-in cores called Tensor Cores. In addition, TensorFlow and PyTorch are representative frameworks used for deep learning-based object detection. Each deep learning framework can operate in various programming languages such as C++, Java, and Python, so you can use it easily even if you are not familiar with a specific programming language.

Therefore, CUDA and Cuda Deep Neural Network Library (cuDNN) were installed using the JetPack package into AGX Xavier, one of the embedded systems provided by Nvidia. It then compared TensorFlow running on CUDA Cores with TRT running on Tensor Cores. The model used for comparison is YOLOv4, the dataset used COCO and PASCAL VOC, and the source code implemented to operate in TensorFlow was used. In TensorFlow, it was converted to TF-Lite, which operates on a mobile basis, and TF-TRT, which uses Tensor Cores. In addition, various frameworks were prepared by applying Mixed Precision's float32 and float16 in the converted deep learning framework and basic TRT (pure TensorRT that does not use TensorFlow). Experiments were conducted to quickly infer objects, even at the risk of some performance degradation when compared to desktop environments in embedded systems.

Section 2 of this paper examines the YOLO model used for deep learning-based object detection and introduces Nvidia Jetson platforms. Section 3 examines cases of deep learning in embedded systems and similar research cases comparing deep learning frameworks in various environments. Section 4 introduces some modified source codes for architecture comparison analysis and performance inference measurements for the deep learning framework. Section 5 compares CPU utilization, GPU utilization, object accuracy, latency, and power consumption through performance evaluation comparisons using the modified deep learning framework. Section 6 concludes this paper with conclusions.

2. Deep Learning Model and Embedded Systems

This section introduces deep learning algorithms and models used to detect objects, as well as Nvidia Jetson platforms that have recently become popular as embedded environment-based equipment.

2.1. Deep Learning in Image Detection

Object detection can classify a type for objects included in an image or video and at the same time determine where the object is located. Object detection expresses the location information of a specific object in images and videos as X, Y coordinate values. Additionally, the width and height values, which are the size of the object, are used as label information, and usually X, Y coordinate values and data of width and height are expressed as bounding boxes [7].

The types of deep learning-based object detection models that have appeared since 2012 can be divided into one-stage detectors and two-stage detectors [4]. To understand the two types, one must understand the concepts of classification and region proposal. Classification is the classification of objects for a particular object, and region proposal is an algorithm that quickly finds areas where objects can be. The two-stage detector performed well in terms of accuracy in detecting objects, but it is limited to real-time detection due to slow prediction. To solve this speed problem, a one-stage detector that performs classification and region proposal at the same time has been proposed. The one-stage detector is a method of obtaining results by simultaneously performing classification and regional proposal.

Figure 1 shows that after the image is input as a model, the features of the image are extracted using the Conv Layer, and region proposal and classification are simultaneously performed to output the result. Representative models include YOLO, RetinaNet, RefineDet, etc. [8].

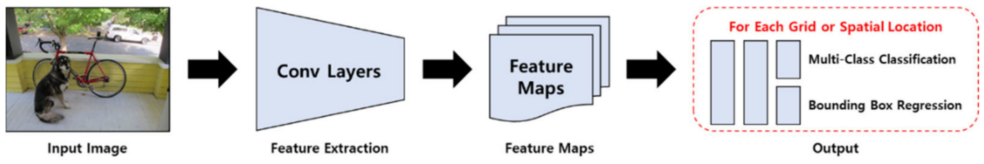


Figure 1. One-stage detector image processing process.

The two-stage detector is a method of obtaining results by sequentially performing classification and regional proposal.

Figure 2 extracts features using region proposal after inputting an image as a model, and predicts candidate areas using CNN operation and sliding-window in feature map extracted through region proposal network. Through classification, it classifies what name the object has and shows the output of the result. Representative models of the R-CNN series include R-CNN, Fast R-CNN, Faster R-CNN, and Mask R-CNN [9].

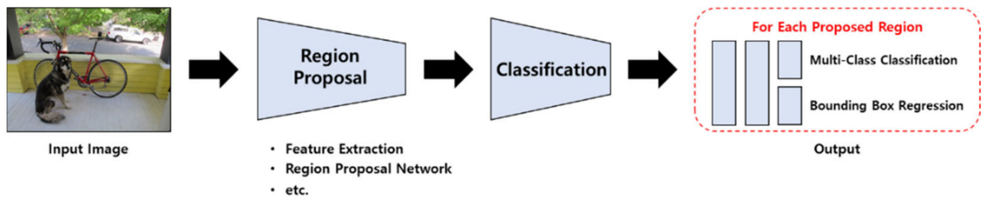


Figure 2. Two-stage detector image processing process.

Among them, YOLO is a one-stage detector and does not perform the existing region proposal and classification steps separately, but performs them at once. In other words, the type and location of the object are predicted at once by considering bounding box and class probability as one problem. The image is divided into grids of a certain size to predict the bounding box for each grid, and the confidence score of the bounding-box and the class score of the grid cell are trained [10].

Figure 3 shows the processing process of YOLO. First, the input image is divided into an S X S grid area. In each grid area, the bounding box is predicted as many as the number of Bs corresponding to the area where there is an object. This is represented by (x, y, w, h), where (x, y) is the center point coordinate of the bounding box, and w, h are the width and height of the bounding box.

$$\Pr(Object) \times IoU_{pred}^{truth} \tag{1}$$

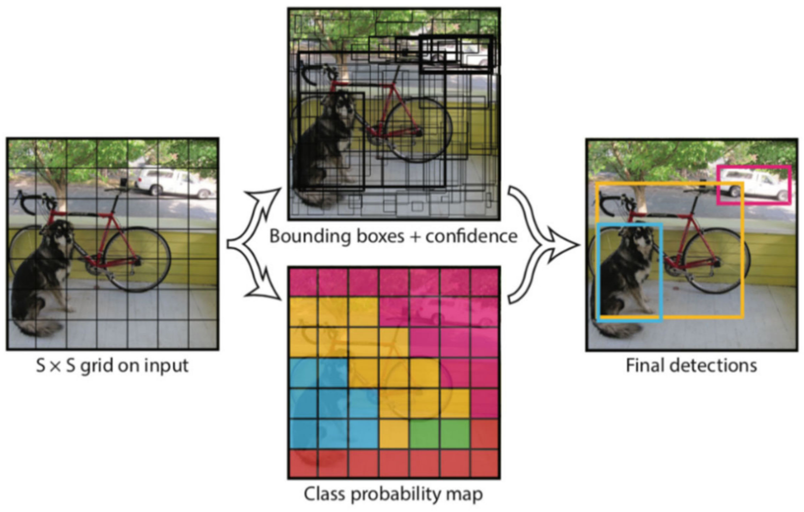


Figure 3. Object detection processing process of YOLO model.

Second, as in Equation (1), the confidence, representing the reliability of the box, is calculated. It is calculated by multiplying the probability Pr(*Object*) of the presence of an object in the grid by *IoU*, which represents the ratio of the overlapping area between the predicted box and the ground truth box.

Finally, for each grid, the probability of C classes is calculated, and Equation (2) is shown.

$$\Pr(Class_i|Object) \tag{2}$$

In this case, what is peculiar is that in the existing Object detection, the number of classes + 1 (background) is always classified as an input to a neural network model, but YOLO does not. In this way, YOLO divides the input image into grids, and simultaneously performs bounding box and classification for each grid.

2.2. Nvidia AI Embedded Systems

Deep learning can be sufficiently performed through GPU computation in a desktop environment. However, IoT and embedded systems are widely used because they have the advantage of relatively inferior processing performance and accuracy when compared to desktop environments, but have a small size and use less power. Therefore, Nvidia sells Jetson platforms, an embedded system. Jetson platforms are largely divided into Nano, TX2, and AGX Xavier. Similar to desktops, hardware performances such as those for CPU, GPU, RAM, and HDD are better, and the more features it provides, the more expensive it is [11]. Table 1 shows a comparison of the specifications of Jetson products.

Table 1. A comparative analysis of the Jetson platform provided by Nvidia.

	Jetson NANO	**Jetson TX2**			**Jetson AGX Xavier**	
		4 GB	**8 GB**	**Industrial**	**8 GB**	**16 GB**
GPU	128-Core Maxwell GPU with CUDA Cores	256-Core Pascal GPU with CUDA Cores			384 Core Volta + NVDLA	512 Core Volta + NVDLA
CPU	Quad-core ARM Cortex-A57	Quad-core Arm Cortex-A57 Quad-core ARM A57 complex			6-core Carmel ARM CPU 1.3 GHZ	8-core Carmel ARM CPU 2.26 GHz
Memory	4 GB 64-bit LPDDR4	4 GB 128-bit LPDDR4	8 GB 128-bit LPDDR4		8 GB 256-bit LPDDR4x	16 GB 256-bit LPDDR4x
Storage	16 GB eMMC 5.1 (Module) Not Include (Dev-Kit)	16 GB eMMC 5.1	32 GB eMMC 5.1		32 GB eMMC 5.1	
Video Encode	4K @ 30 (H.264/H.265)	2 × 4K @ 30 (HEVC)			2 × 4K @ 60 (HEVC)	4 × 4K @ 60 (HEVC)
Video Decode	4K @ 60 (H.264/H.265)	2 × 4K @ 30, 12-bit support			2 × 4K @ 30 (HEVC)	2 × 8K @ 30 (HEVC)
Camera	12 lanes (3 × 4 or 4 × 2) MIPI CSI-2 DPHY 1.1 (1.5 Gbps)	12 lanes MIPI CSI-2, D-PHY 1.2 (30 Gbps)			16 lanes MIPI CSI-2 D-PHY 1.2 (40 Gbps)	
Connectivity	Gigabit Ethernet	Gigabit Ethernet	Wi-Fi Gigabit Ethernet	Gigabit Ethernet	Gigabit Ethernet	
Display	HDMI 2.0 | DP 1.2 eDP 1.4 | DSI (1 × 2) 2	HDMI 2.0 | DP 1.2 eDP 1.4 | DSI (1 × 2) 2			HDMI 2.0 | eDP 1.4, DP HBR3	
UPHY	1 × 1/2/4 PCIE, 1 × USB 3.0, 3 × USB 2.0	Gen2 | 1 × 4 + 1 × 1 OR 2 × 1 + 1 × 2, USB 3.0 + USB 2.0			(8×) PCIe Gen4 | (8×) SLVS-EC (3×) USB 3.1 Single Lane UFS	
I/O	SDIO, SPI, I2C, I2S, GPIOs	CAN, UART, SPI, I2C, I2S, GPIOs			UART, SPI, CAN, I2C, I2S, DMIC, GPIOs	

Table 1. *Cont.*

	Jetson NANO	Jetson TX2			Jetson AGX Xavier	
		4 GB	8 GB	Industrial	8 GB	16 GB
Module Size	69.6 mm × 45 mm	87 mm × 50 mm			87 mm × 100 mm	
Mechanical	260-pin Connector	400-pin connector			699 pi Connector	
Performance	472 GFLOPs	1 TFLOPs	1.3 TFLOPs		10 TFLOPs	16 TFLOPs
Power	5/10 W	7.5/15 W	10/20 W		10/20 W	10/15/30 W

The Jetson Nano platform is a small AI computer that runs a deep learning neural network in parallel and has the performance and power efficiency required to process data from multiple high-resolution sensors simultaneously. It is the perfect level of platform to add advanced AI to embedded products and is used in embedded IoT applications, including network video recorders, home robots, and intelligent gateways with full analysis capabilities.

The Jetson TX2 delivers unparalleled speed and power efficiency in embedded AI computing devices, enabling true AI computing through a wide range of standard hardware interfaces. The TX2 embedded module for installing AI applications on Edge Devices comes in three versions: TX2 (4 GB), TX2 (8 GB), and TX2i (Industrial). In particular, even if a TX1-based platform is purchased, it has the advantage of being able to migrate to TX2 4 GB.

The Jetson AGX Xavier is an embedded system specifically designed for autonomous machines and has the best hardware performance on the market today. It can easily process sensor data and run AI software, and provides the best performance and power efficiency among Jetson platforms. AGX Xavier provides the best performance among Jetson platforms for high-speed computing, energy efficiency, and AI-based inference functions.

3. Related Work and Contribution

This section introduces research cases that grafted deep learning technology in an embedded environment and studies that compared and analyzed various deep learning frameworks.

3.1. Deep Learning Study in Embedded Environment

In [12], the author proposed milliEye, a lightweight mmWave radar and camera fusion embedded system for robust object detection on edge platforms in low-light environments by fusing radar and images. In [13], the author proposed a training plan to detect objects, using drones, with NVIDIA Jetson TX2 for real-time drone detection using pretrained weights and YOLOv3, which is capable of transfer learning. The detection result after training proved that the average accuracy was 88.9% at the input image size of 416 × 416. In [14], the author compared Visual Odometry and Visual–Inertial Odometry algorithms in several NVIDIA Jetson platforms, such as NVIDIA Jetson TX2, Xavier NX, and AGX Xavier. Additionally, the author proposed a new dataset, the KAIST VIO dataset, for an unmanaged aerial vehicle. In [15], the author proposed a Lane Department Warning System (LDWS) based on CNN encoder-decoder and long short-term memory networks in Nvidia Jetson Xavier NX embedded environments. LDWS demonstrated high predictive performance of 96.36% average accuracy, 97.54% recall, and 97.42% F1 score. In [16], the author evaluated the performance of Nvidia Jetson Nano in the Dew computing approach using ML applications. Experimental evaluation metrics measured processing resources (CPU, GPU), device temperature, power consumption, and RAM usage used in ML operations. In [17], the author introduced the advantages of using machine learning in NVIDIA's Jetson embedded system, and the results were provided by investigating the work of evaluating and optimizing neural network applications on the Jetson platform. In addition, hardware and algorithm optimization, performed to execute neural network algorithms in Jetson, was reviewed and actual applications to which these algorithms were applied were introduced. In [18], the author examined the architecture of Full-Convolutional Neural Networks for depth reconstruction, and proposed several improvements to increase the efficiency of inference. In addition, frame speed was evaluated for an input of 320 $\times$ 240 network size to provide the best performance and accuracy for NVidia Jetson TX2.

3.2. Deep Learning Framework Performance Evaluation Study

In [19], the authors proposed a comparative study of GPU-accelerated deep learning software frameworks such as PyTorch and TensorFlow. Three different neural networks were implemented through MNIST, CIFAR10, and Fashion MNIST datasets to benchmark the performance of the framework. In [20], the authors analyzed the performance of the three frameworks, Caffe, TensorFlow, and Apache SINGA, in various hardware environments to better understand the performance impact of the deep learning framework on various resources and provide guidelines for future hardware technologies. In [21], the authors used a fashion image dataset and compared the performance of single-board computers on NVIDIA Jetson Nano, NVIDIA Jetson TX2, and Raspberry PI4 through CNN algorithms. Metrics for performance analysis were defined as consumption (GPU, CPU, RAM, power), accuracy, and cost. In [22], the authors compared and analyzed the performance of various deep learning frameworks in terms of inference time and power consumption in Asus Tinker Edge R, Raspberry Pi 4, Google Coral Dev Board, and Nvidia Jetson Nano, the System-On-Chip (SoC)-type embedded systems. In the case of the MobileNetV2 neural network, the Jetson Nano reduced inference time to less than 29.3%. In [23], the authors introduce theories related to optimization technology for the proposed deep learning framework (TF-Lite, TRT) for edge computing. Additionally, authors proposed a detailed performance study of TF-Lite and TF-TRT in edge devices on various hardware platforms. The main comparisons are throughput, latency, and power consumption.

3.3. Contribution

Research cases of converging IoT technologies, such as drones with deep learning by utilizing SoC-type embedded systems for edge computing, are increasing. In addition, there are many cases of testing neural network models in various environments, but there are not many comparative analysis studies that can utilize the deep learning framework in Jetson platforms. Therefore, this paper compared and analyzed the TensorFlow, TF-Lite, TF-TRT, and TRT deep learning frameworks based on the Jetson AGX Xavier 16 GB platform. The source code required for detection was further developed to measure metrics generated during performance inference based on those provided on Github. In other words, YOLOv4-based TensorFlow, TRT source code can be developed by integrating with the Jetson monitoring tool, and the optimized deep learning framework can be selected by checking CPU and GPU utilization, object accuracy, and power consumption, which are necessary information for developing deep learning-related services in various IoT and embedded environments. The dataset used for evaluation downloads the weights of the pre-trained model of the COCO dataset that detects 80 objects, converts them into various deep learning frameworks, and compares and analyzes them [24]. In the case of the PASCAL VOC dataset that detects 20 objects, since the pretrained YOLOv4 model is not provided, the dataset is downloaded directly, and weights are obtained through training [25,26]. This study is expected to provide guidelines on which deep learning framework to use when research using the Jetson platform or developing applications related to TensorFlow and TRT based on it.

4. Overview and Development of Deep Learning Framework

This section introduces what was developed for architecture and performance evaluation for the four deep learning frameworks (TensorFlow, TF-Lite, TF-TRT, and TRT) compared and analyzed in this paper.

4.1. Deep Learning Framework Overview

4.1.1. TensorFlow

TensorFlow 2.0 provides a simple and user-friendly Keras standard API for building and learning high-level models [27]. The overall structure of TensorFlow 2.0 is shown in Figure 4.

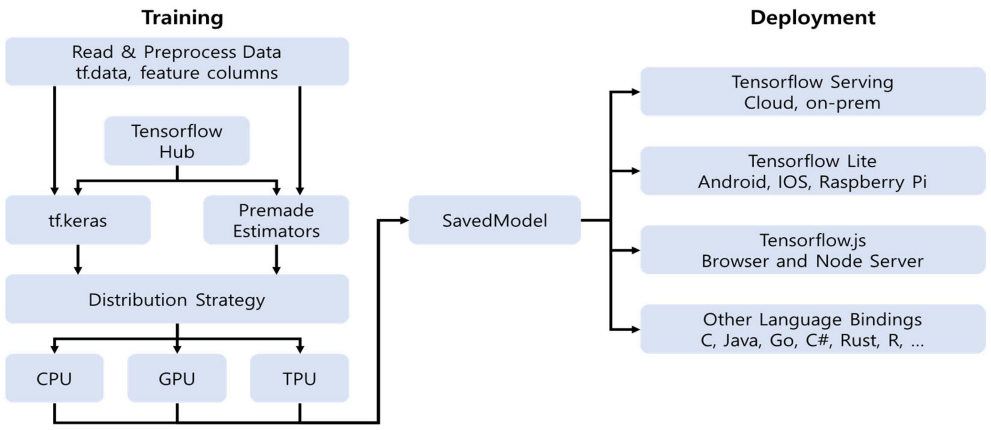


Figure 4. TensorFlow 2.0 version architecture and processing process.

Use Tf.data to load data. Build and train models using Tf.keras, validate, or use the Premade Estimator. Execute using Eager execution, debug and use tf.function to utilize the graph later. Distribution Strategy API is used for distributed learning. Save the model and export it according to the required environment. TensorFlow 2.0 can export regardless of target frameworks such as servers, edge devices, and the web. That is, the user may use TensorFlow 2.0 to train and export the model regardless of the programming language.

4.1.2. TensorFlow-Lite

TF-Lite is a deep learning framework for mobile devices developed by Google. Only the TensorFlow model can be optimized, and TF-Lite consists of a Converter and Interpreter. The Converter module serves as an optimization module to maintain performance in the TensorFlow model so that it can be used efficiently. The Interpreter module serves to help the optimized model be executed on the mobile device. The processing process of TF-Lite is shown in Figure 5.

After training TensorFlow, the trained model is optimized using Converter and converted to TF-Lite format. Through Interpreter, TF-Lite can be used on other devices [28,29]. The biggest feature of TF-Lite is that during conversion from a trained model to TF-Lite, Converter supports mixed precision quantization such as float and integer. Mixed-precision transformations can reduce model size by 2× at the cost of minimal impact on latency and accuracy.

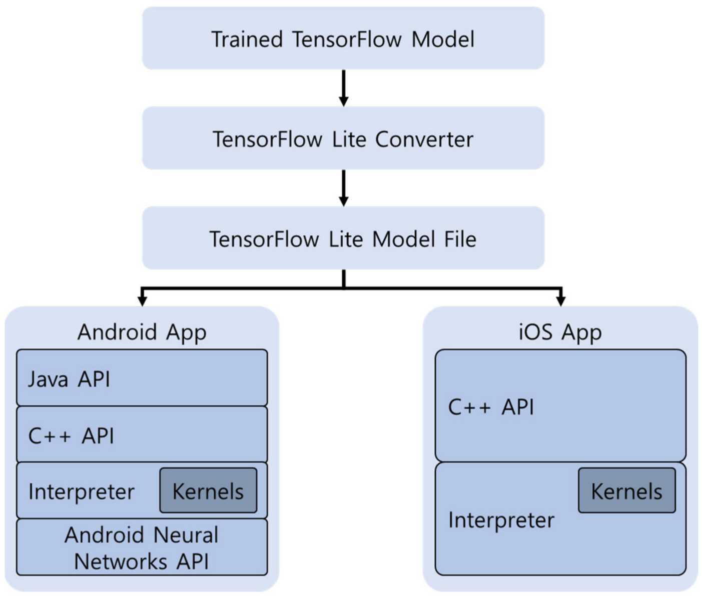


Figure 5. TensorFlow-Lite architecture and processing process.

4.1.3. TensorFlow-TensorRT and TensorRT

TRT is a deep learning framework optimization engine that can help improve deep learning services by optimizing learned deep learning models and improving inference speeds in NVIDIA GPUs by tens of times. Models generated through deep learning frameworks such as Caffe, PyTorch, and TensorFlow are optimized through TRT and re-recognized on NVIDIA GPU platforms (TESLA T4, JETSON TX2, TESLA V100, etc.). TRT supports models learned in most deep learning frameworks and supports optimal deep learning model acceleration. Because it supports C++ and Python at the API level, it can be easily used by developers in the deep learning field without much knowledge of CUDA or GPU programming. In addition, it builds a runtime binary to automatically use the optimal computational resources supported by GPUs, which makes it easy to improve latency and throughput, enabling the efficient execution of deep learning applications and services [30].

The fusion of TensorFlow and TRT optimizes and executes compatible subgraphs, creating a model for TensorFlow to execute the rest of the graphs. In other words, TensorFlow's extensive and flexible functions can continue to be used, and TRT parses the model and applies optimization to the graph portion as much as possible. The processing process of TensorFlow and TF-TRT is shown in Figure 6.

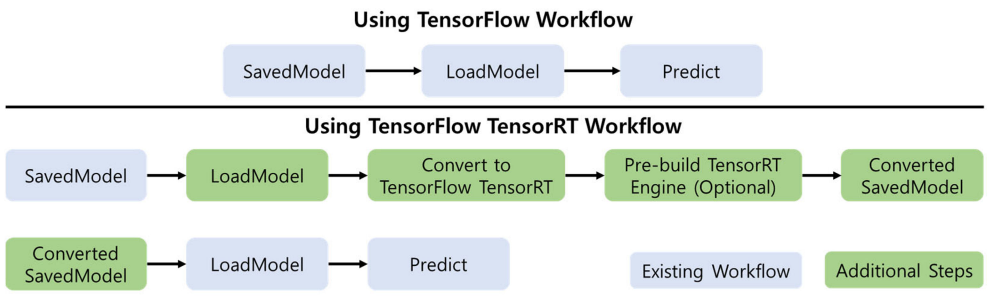


Figure 6. The workflow comparison processing process of TensorFlow and TensorFlow-TensorRT.

Figure 6 shows a comparison of the inference workflow of the basic TensorFlow and TRT. TensorFlow workflow typically loads the saved model and executes inference using TensorFlow runtime. For TF-TRT, TRT optimization is applied to the model's TRT support subgraph. Optionally, several additional steps are included, including pre-building the TRT engine. The precision mode is used to represent the mixing precision (e.g., float32, float16) that can be used to implement TF-TRT operations. Then, in the saved model, a convert object is created to obtain transformation parameters and inputs. In TensorFlow 2.0, TF-TRT only supports models stored in the TensorFlow SavedModel format. Next, when the converter convert() method is called, TF-TRT converts the graph by replacing the TRT-compatible part of the graph with TRT EngineOps. The TRT execution engine must be built on a GPU of the same device type as the device type on which the inference will be run, because the build process is GPU-specific. For example, if you generate TF-TRT based on the T194 GPU used by Jetson AGX Xavier, it will not work with T4 GPU of other Jetson equipment [31].

4.2. *Deep Learning Framework Development*

Source codes based on measuring inference performance are [32,33]. Authors in [32] ran the YOLOv4 model in TensorFlow and converted it into TF-Lite and TF-TRT. Reference [33] describes a source code that uses pure TRT and can convert YOLO4 models using pycuda and onnx libraries. The source code for monitoring CPU, GPU, and power consumption of the Nvidia Jetson platform that can measure inference performance using [34] was applied to [32,33]. However, since the entire source code that has been further developed and modified for measuring inference performance cannot be reflected in this paper, only the main parts are introduced.

4.2.1. TensorFlow, TensorFlow-Lite, TensorFlow-TensorRT

Development 1 shows the main part of the utils.py source code for measuring the performance inference of TensorFlow, TF-Lite, and TF-TRT.

Development 1. /core/utils.py

```
155  if “coco” in FLAGS.weights:
156      if class_ind == 0:
157              value_0 = score*100
158      elif class_ind == 2:
159          value_1 = score*100
160      else:
161          continue
162   if “voc” in FLAGS.weights:
163      if class_ind == 14:
164              value_0 = score*100
165      elif class_ind == 6:
166          value_1 = score*100
167      else:
168          continue

180  Return image, value_0, value_1
```

In Development 1, modify the draw_bbox() function in /core/utils.py. When detecting, the object number of the predefined text file is different, so it is separated and executed according to the dataset used through the IF syntax. In the case of COCO dataset, when the person object is detected (“0”) according to the object number of the COCO.names text file, it is stored in the value_0 variable, and when the car object (“2”) is detected, it is stored in value_1. In the case of PASCAL VOC dataset, since the object number of the text file is different, when the person object is detected (“14”), it is stored in the value_0 variable, and when the car object (“6”) is detected, it is stored in value_1. Since the accuracy has a value between 0.1 and 1, multiply by 100 (Line 155~168). The variables that store the object’s accuracy are returned to the main function of detectvideo_jtop.py (Line 180). Then, copy detectvideo.py, the main source code for measuring inference performance, and create a new detectvideo_jtop.py.

Development 2 shows the main part of the source code for measuring TensorFlow’s performance inference. Related libraries for saving metrics generated during performance inference measurements are input (Line 15 to 17). For performance inference measurements, a DictWriter function of the csv library is added to load an image using the read() method of the VideoCapture function of the OpenCV library at the same time as the code starts, and detection starts (Line 67~78). If you input "q" when measuring inference performance, inference stops and calculates and outputs the average accuracy of Person and Car, framework name used, model name used, and average delay time. (Line 140~155). The following copies de-tectvideo_jtop.py, written in Development 2 to detectvideo_jtop_tflite.py for TF-Lite performance inference measurement.

Development 2. /detectvideo_jtop.py

```
15  from jtop import jtop, JtopException
16  import csv
17  import argparse
```

```
67  try:
68      with jtop() as jetson:
69      with open(FLAGS.csvfile, 'w') as csvfile:
70            stats = jetson.stats
71            writer = csv.DictWriter(csvfile, fieldnames = stats.keys())
72            writer.writeheader()
73            writer.writerow(stats)
74            while True:
75            stats = jetson.stats
76            writer.writerow(stats)
77            return_value, frame = vid.read()
78        . . . # (obmitted—it's the same as detectvideo.py)
```

```
140 if cv2.waitKey(1) & 0xFF == ord('q'):
141     del info_list[0]
142     del info_list[1]
143     del info_list[2]
144     best_list_0_sum = sum(best_list_0)
145     best_list_0_avg = best_list_0_sum/len(best_list_0)
146     best_list_1_sum = sum(best_list_1)
147     best_list_1_avg = best_list_1_sum/len(best_list_1)
148     info_list_sum = sum(info_list)
149     info_list_avg = info_list_sum/len(info_list)
150     print("person average: {:.2f}".format(best_list_0_avg))
151     print("car average: {:.2f}".format(best_list_1_avg))
152     print("framework: {}".format(FLAGS.framework))
153     print("weights: {}".format(FLAGS.weights))
154     print("average ms: {:.2f}".format(info_list_avg))
155     break
```

Development 3 shows the main part of the source code for measuring performance inference of TF-Lite. Lines 15 to 17, Lines 67 to 78, and Lines 170 to 185 are the same as detectvideo_jtop.py. The code was modified and supplemented because the detectvideo_jtop.py created for TF-Lite performance inference measurement was executed, but a list index range error occurred and was not executed. In case of YOLOv4-Tiny and YOLOv4-Native, it is divided and executed through the if statement, and since the shape of the model input to TF-Lite is different, specify a different list index according to the precision of float32 and float16 and store the result in the output_tesnors variable through the decode function (Line 108~130).

Development 3. /detectvideo_jtop_tflite.py

```
15
~    ... # (obmitted—it’s the same as detectvideo_jtop.py 15~17)
17

67
~    ... # (obmitted—it’s the same as detectvideo_jtop.py 67~78)
78

     bbox_tensors = []
108  prob_tensors = []
109  if FLAGS.tiny:
110      for i, fm in enumerate(pred):
111          if i == 0:
112              output_tensors = decode(pred[1], input_size//16, NUM_CLASS, STRIDES, ANCHORS, i, XYSCALE,
113  ’tflite’)
114          else:
115              output_tensors = decode(pred[0], input_size//32, NUM_CLASS, STRIDES, ANCHORS, i, XYSCALE,
116  ’tflite’)
117          bbox_tensors.append(output_tensors[0])
118          bbox_tensors.append(output_tensors[0])
119  else:
120      for i, fm in enumerate(pred):
121          if i == 0:
122              output_tensors = decode(pred[2], input_size//8, NUM_CLASS, STRIDES, ANCHORS, i, XYSCALE,
123  ’tflite’)
124          elif i == 1:
125              output_tensors = decode(pred[0], input_size//16, NUM_CLASS, STRIDES, ANCHORS, i, XYSCALE,
126  ’tflite’)
127          else:
128              output_tensors = decode(pred[1], input_size//32, NUM_CLASS, STRIDES, ANCHORS, i, XYSCALE,
129  ’tflite’)
130          bbox_tensors.append(output_tensors[0])
             prob_tensors.append(output_tensors[1])
     pred_bbox = tf.concat(bbox_tensors, axis = 1)
     pred_prob = tf.concat(prob_tensors, axis = 1)
     pred = (pred_bbox, pred_prob)
170
~    ... # (obmitted—it’s the same as detectvideo_jtop.py Line 140~155)
185
```

TF-Lite and TF-TRT may convert precision to float32 and float16 through quantize mode argument of convert_tflite.py and convert_trt.py. In TF-Lite, saved_model.py is executed, and conversion is completed through convert_tflite.py. However, when TF-TRT executes saved_mode.py and attempts further conversion through convert_trt.py, the existing source code does not work and a max_batch_size error occurs. The following shows a modified version of convert_trt.py.

Development 4 shows the modified convert_txt.py. The max_batch_size portion corresponding to Lines 58 and 66 is annotated. The memory provided by Nvidia Jetson AGX Xavier is in the form of an SoC, so it was confirmed that the max_batch_size running in a typical desktop Linux environment cannot be read. That is, this is because the memory is not separately mounted on the mainboard, the memory is not used for each hardware (RAM, GPU), and the integrated memory in the form of SoC is shared and used together. If you comment the line and run convert_trt.py, there is no problem, because it checks the amount of memory available to Jetson AGX Xavier.

Development 4. /convert_txt.py
58 # max_batch_size = 8
66 # max_batch_size = 8

4.2.2. TensorRT

Modify onnx_to_tensorrt.py, which is the source code related to the onnx library for converting to TRT.

Development 5 shows a portion related to the conversion of the float32, float16. The fp16_mode corresponding to Line 139 in onnx_to_tensorrt.py is set to True by default. The default value of True is converted to a TRT model with float16 precision, inputting the variable value as False, and executing onnx_to_tensorrt.py, it can be converted to a TRT model with float32 precision.

Development 5. /yolo/onnx_to_tensorrt.py
139 builder.fp16_mode = True # float16 mode enable
139 builder.fp16_mode = False # float32 mode enable

The source code modification for measuring the inference performance of pure TRT modifies trt_yolo.py and visualization.py of [33]. As for the source code format, the utils.py part of Section 4.1.1 is the same as visualization.py, and the trt_yolo.py part is the same as detectvideo_jtop.py. Therefore, the accuracy measured according to the class number of the object modified in utils.py is input as two variables and returned as trt_yolo.py. In addition, trt_yolo.py finds the read() method part of the OpenCV library in the same way as detectvideo_jtop.py and adds code for performance inference measurement. Finally, when "q" is input, inference stops and calculates the average accuracy of Person and Car, the framework name used, the model name used, and the average delay time, and adds the output code. The method of modifying and adding the source code in the TRT will be omitted because it is the same as Section 4.1.1.

5. Result and Discussion

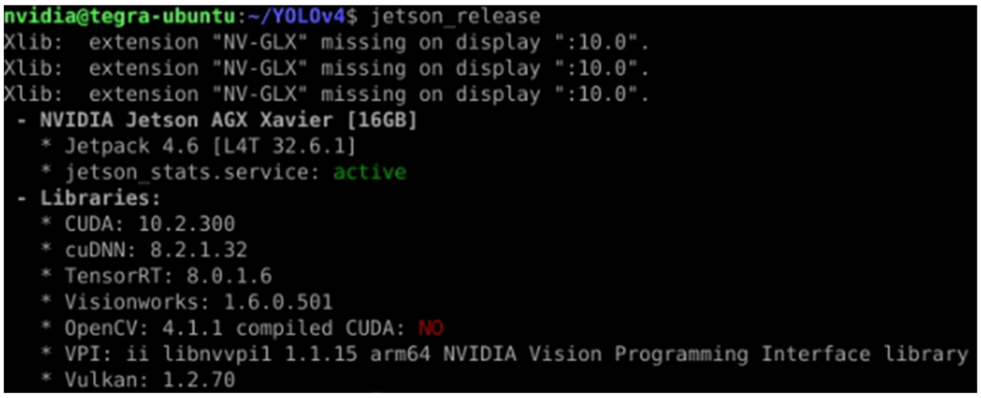
This section introduces the experimental results environment and results of the four deep learning frameworks that were comparatively analyzed in this paper.

5.1. Enviroment

For the deep learning framework comparative analysis, AGX Xavier 16 GB, sold by Nvidia, was used. The OS is installed through SDK Manager provided by Ubuntu-based Nvidia, and version 32.6.1 is applied. To install CUDA, cuDnn, and TRT, install the Jetpack package version 4.6 to complete the deep learning-related library environment setup. [35]. The library version installed through the Jetpack package is shown in Figure 7.

For high-performance inference speed and accuracy, MAXN mode is applied, which lifts the limit on CPU and GPU utilization [36]. TensorFlow was installed through the installation method provided by Nvidia, and version 2.6.2 applies [37]. Python version uses built-in 3.6.9 by default. The COCO dataset has 80 classes, and the number of images for training is 118,287 and the number of images for validation was 5000. Download the YOLOv4-Native model and the YOLOv4-Tiny model that have been trained through the COCO dataset [38,39]. The PASCAL VOC dataset has 20 classes, and the total number of images for training and validation was 17,125. Since the PASCAL VOC dataset does not provide a trained YOLOv4 model, the data were downloaded directly and trained through the darknet deep learning framework [25,26]. The environment used for training (Nvidia Jetson AGX Xavier for detection and performance inference measurements) is a desktop environment with Ubuntu 20.04 OS, DDR4 32 GB memory, and Nvidia Geforce RTX 2070

Super GPU. The learning process first converts the data annotated in the PASCAL VOC format into the YOLO format using the [40] source code. Second, the converted YOLO format file and PASCAL VOC image data are copied into one folder. Third, after copying, the data for training and verification are divided in an 8:2 ratio, and the value when divided into nine grid cells in the YOLO model is calculated through the calc_anchors command provided by darknet. Fourth, input the output anchors value in the configuration file yolov4-custom.cfg, input the classes factor in the configuration file as 20, and input the filters factor as 75, because it is calculated as (number of classes + 5) × 3. Fifth, the width and height of the neural network were input as 416 × 416, the same as COCO. Finally, we trained through darknet's train command, and default values were applied to other input parameters, such as learning rate and max batches [26].



```
nvidia@tegra-ubuntu:~/YOLOv4$ jetson_release
Xlib:  extension "NV-GLX" missing on display ":10.0".
Xlib:  extension "NV-GLX" missing on display ":10.0".
Xlib:  extension "NV-GLX" missing on display ":10.0".
 - NVIDIA Jetson AGX Xavier [16GB]
   * Jetpack 4.6 [L4T 32.6.1]
   * jetson_stats.service: active
 - Libraries:
   * CUDA: 10.2.300
   * cuDNN: 8.2.1.32
   * TensorRT: 8.0.1.6
   * Visionworks: 1.6.0.501
   * OpenCV: 4.1.1 compiled CUDA: NO
   * VPI: ii libnvvpil 1.1.15 arm64 NVIDIA Vision Programming Interface library
   * Vulkan: 1.2.70
```

Figure 7. Check various library versions that make up the experimental environment.

The source code uses [32], which can evaluate the YOLOv4 model using the TensorFlow deep learning framework. In [32], the YOLOv4-Native.weights and YOLOv4-Tiny.weights files were converted into models required for comparative analysis because the code that can be converted to TF-Lite and TF-TRT was built-in. In the case of pure TRT, the source code provided in [32] is not available because it is a different structure from TensorFlow, so another source code [33] was used. In order to check the metrics generated during inference performance measurement, the monitoring source code [34] provided by Nvidia was applied and modified to [32,33], and the values generated during inference were automatically saved in the form of a CSV file. When converting YOLOv4 to match the deep learning framework, the network size was configured as 416 × 416 all the same. Details of the differences and source codes of the deep learning framework can be found in Section 4.

The video used to measure inference performance has a resolution of 1280 × 720 at 30 FPS and is 30 s long. The video is a night image of a low-light environment, two to three people are detected, and three cars are detected fixedly. For readers who are curious about the image of the sample video used, the screen captured during inference performance measurement is shown in Figure 8.

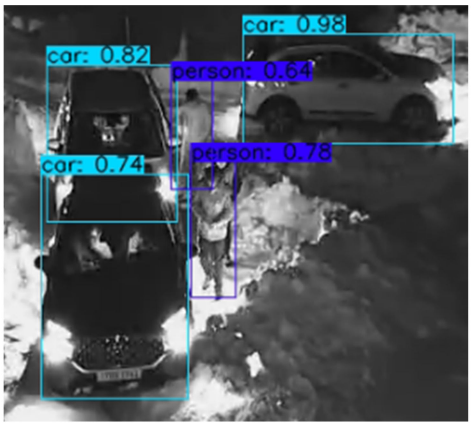


Figure 8. Some of the images captured during the inference performance measurement.

5.2. Evaluation Metrics

The metrics for comparing and analyzing inference performance through four deep learning frameworks are as follows.

1. CPU utilization: this refers to the measurement rate of CPU usage that occurs during inference performance. Nvidia AGX Xavier has eight cores, and the usage of eight cores is calculated as an average. The value is expressed as a percentage and has a value between 0 and 100.
2. GPU utilization: this refers to the measurement rate of GPU usage that occurs during inference performance. The value is expressed as a percentage and has a value between 0 and 100.
3. Accuracy: this refers to the average accuracy of person and car objects that occur during the measurement of inference performance. Even if the number of persons and cars is measured differently for each frame, it is calculated as the overall average of the measured objects. In addition, when objects other than person and car are detected, the accuracy of other objects is discarded in order to reduce errors in false detection and non-detection. The value is expressed as a percentage and has a value between 0 and 100.
4. Latency: This is the execution time measured when performing inference on the image per frame used for inference of the sample video. The value is expressed as milliseconds.
5. Power: This refers to the power consumption of the Nvidia AGX Xavier used for inference. The value is indicated by W.

5.3. Evaluation Result

5.3.1. Comparative Analysis of YOLOv4 Model's Inference Performance in the COCO Dataset

Tables 2 and 3 show the results of measuring inference performance through sample video by transforming YOLOv4-Native.weight and YOLOv4-Tiny.weight according to each deep learning framework in the COCO dataset.

For TensorFlow in the COCO dataset, the CPU utilization of YOLOv4-Native is 20.80%, slightly lower than the 27.99% of YOLOv4-Tiny. However, the GPU utilization rate is 73.50%, which is about 1.4 times higher than the 52.19% of YOLOv4-Tiny. Because it utilizes more GPU computation for object detection, it is confirmed that the GPU utilization rate and the accuracy of human and vehicle object detection are 1.3 times higher. Due to the high utilization of the GPU, the measured power consumption using more power is also about 1.8 times higher. The model of YOLOv4-Native has 3 YOLO heads and 137 pretrained convolutional layers, and YOLOv4-Tiny has 2 YOLO heads and 29 pretrained convolutional layers. Therefore, due to the complexity and size of the model, it has a delay time of 127.82 ms for YOLOv4-Native and is measured to be about 4.5 times higher than the

28.38 ms of YOLOv4-Tiny, so the accuracy of detecting an image object in one frame is high, but the processing speed is slow.

Table 2. Comparative analysis of YOLOv4-Native.weights's inference performance in the COCO Dataset.

Framework (YOLOv4-Native in the COCO Dataset)	Precision	Average CPU Utilization (%)	Average GPU Utilization (%)	Average Accuracy (%)	Average Latency (ms)	Average Power (W)
TensorFlow	Float32	20.80	73.50	Person: 43.53 Car: 71.29	127.82	28.24
TF-Lite	Float32	18.95	0.91	Person: 42.66 Car: 71.12	5228.06	9.67
TF-TRT		24.49	67.41	Person: 42.55 Car: 71.29	61.63	15.87
TRT		29.33	52.04	Person: 43.26 Car: 72.18	27.76	21.91
TF-Lite	Float16	17.99	0.62	Person: 40.13 Car: 71.87	5458.35	9.40
TF-TRT		29.65	47.49	Person: 43.55 Car: 71.27	32.15	11.79
TRT		28.89	54.75	Person: 43.31 Car: 72.47	27.92	22.48

Table 3. Comparative analysis of YOLOv4-Tiny.weights's inference performance in the COCO Dataset.

Framework (YOLOv4-Tiny in the COCO Dataset)	Precision	Average CPU Utilization (%)	Average GPU Utilization (%)	Average Accuracy (%)	Average Latency (ms)	Average Power (W)
TensorFlow	Float32	27.99	52.19	Person: 33.33 Car: 60.73	28.38	14.38
TF-Lite	Float32	20.32	0.13	Person: 33.96 Car: 60.38	600.67	9.71
TF-TRT		28.13	51.08	Person: 33.95 Car: 60.46	30.25	11.01
TRT		31.18	39.76	Person: 34.70 Car: 60.31	18.02	12.06
TF-Lite	Float16	20.57	0.13	Person: 33.76 Car: 60.10	604.10	9.48
TF-TRT		31.81	38.62	Person: 33.76 Car: 60.93	24.19	10.10
TRT		31.07	42.20	Person: 34.02 Car: 61.01	17.85	12.40

TF-Lite, TF-TRT, and TRT in the COCO dataset, the model was converted by dividing the mixing precision by float32 and float16. In the case of TF-Lite, the accuracy of object detection for both float32 and float16 in YOLOv4-Native and YOLOv4-Tiny is similar to the previously measured results, but it can be seen that the latency is very high. This is because TF-Lite is a lightweight deep learning framework of TensorFlow, but has different purposes. TF-Lite is a deep learning framework specialized for mobile devices such as Android and IOS; it does not apply to the Jetson AGX Xavier that is currently used, so the GPU usage rate does not work below 1% [41]. Therefore, the GPU usage rate is low, and the power consumption is also the lowest among all deep learning frameworks because the GPU does not operate.

For TF-TRT in the COCO dataset, both YOLOv4-Native and YOLOv4-Tiny use the GPU efficiently with more than 50% of GPU usage. In YOLOv4-Native, the power consumption of TF-TRT is measured as 15.87 and 11.79 W, which are slightly higher than the 9.67 and 9.40 W of TF-Lite. In the case of YOLOv4-Tiny, the values are slightly different, but the results are similar. Moreover, when compared to TensorFlow, the accuracy of objects is almost unchanged, and both result in lower latency and lower power consumption.

For TRT in the COCO dataset, it has the highest CPU utilization among the four deep learning frameworks, and GPU utilization is appropriately used from at least 42.20 to 54.75%. In the case of TRT (float32, float16) of YOLOv4-Tiny, the average power consumption is low at about 12 W, but in the case of TRT (float32, float16) of YOLOv4-Native, the power consumption is measured at about 22 W. When compared with the basic TensorFlow of 28.24 W, as a low number is measured, the object accuracy of people and cars is almost unchanged, and it shows low latency.

5.3.2. Comparative Analysis of YOLOv4 Model's Inference Performance in the PASCAL VOC Dataset

Tables 4 and 5 show the results of measuring inference performance through sample video by transforming YOLOv4-Native.weight and YOLOv4-Tiny.weight according to each deep learning framework in the PASCAL VOC dataset.

In the case of all deep learning frameworks in the PASCAL VOC dataset, when compared to the COCO dataset, the measured values are different, but show somewhat similar results.

For TensorFlow in the PASCAL VOC dataset, the CPU utilization of YOLOv4-Native is slightly lower than that of YOLOv4-Tiny, and the GPU utilization is also higher than that of YOLOv4-Tiny. The latency of YOLOv4-Native is higher than that of YOLOv4-Tiny, so the image object detection accuracy of one frame is high, but the processing speed is slow. In particular, in the accuracy of the object, the person was measured 3–4% lower, and in the case of cars, they were measured 9–11% lower. For TF-Lite in the PASCAL VOC dataset, the delay time is very high. The reason is omitted because it is described in Section 5.3.1. For TF-TRT in the PASCAL VOC dataset, both YOLOv4-Native and YOLOv4-Tiny use the GPU efficiently with a GPU utilization rate of about 50%. For TRT in the PASCAL VOC dataset, CPU and GPU utilization rates are used appropriately among the four deep learning frameworks. Also, a low power consumption value was measured, and a result in which the accuracy of the object is maintained to some extent was measured.

Table 4. Comparative analysis of YOLOv4-Native.weights's inference performance in the PASCAL VOC Dataset.

Framework (YOLOv4-Native in the VOC Dataset)	Precision	Average CPU Utilization (%)	Average GPU Utilization (%)	Average Accuracy (%)	Average Latency (ms)	Average Power (W)
TensorFlow	Float32	22.43	70.35	Person: 38.25 Car: 69.81	115.39	26.10
TF-Lite	Float32	20.44	1.87	Person: 38.74 Car: 70.03	4983.97	10.73
TF-TRT		23.62	68.92	Person: 39.01 Car: 70.35	58.14	16.87
TRT		27.98	55.30	Person: 39.26 Car: 69.95	26.93	23.72
TF-Lite	Float16	18.98	1.24	Person: 37.13 Car: 68.34	4810.34	10.29
TF-TRT		30.46	49.50	Person: 38.13 Car: 68.97	29.48	12.45
TRT		29.71	56.32	Person: 38.31 Car: 70.22	26.70	23.72

Table 5. Comparative analysis of YOLOv4-Tiny.weights's inference performance in the PASCAL VOC Dataset.

Framework (YOLOv4-Tiny in the VOC Dataset)	Precision	Average CPU Utilization (%)	Average GPU Utilization (%)	Average Accuracy (%)	Average Latency (ms)	Average Power (W)
TensorFlow	Float32	29.45	50.28	Person: 32.43 Car: 59.24	27.17	13.42
TF-Lite	Float32	22.14	0.57	Person: 31.86 Car: 58.38	573.16	10.14
TF-TRT		30.68	52.48	Person: 32.04 Car: 59.42	29.83	11.56
TRT		29.27	38.03	Person: 32.67 Car: 59.91	17.90	10.39
TF-Lite	Float16	21.48	0.41	Person: 30.73 Car: 57.42	580.53	10.06
TF-TRT		30.52	40.38	Person: 31.97 Car: 58.42	25.49	11.24
TRT		28.35	45.92	Person: 31.16 Car: 59.70	17.30	13.76

5.4. Discussion

Considering the resource usage (CPU and GPU utilization), object accuracy, latency, and power consumption measured through the evaluation metrics, the opinion of using deep learning frameworks for the Jetson AGX Xavier 16 GB platform is as follows. If object detection services and research are conducted based on TensorFlow, the use of TF-TRT seems to have the best performance, and if service and research are conducted on SoC-type mobile devices (Android, IOS), it is most efficient to use TF-Lite. Finally, if you do not have enough knowledge of TensorFlow, it seems that you can efficiently conduct object detection service and research using pure TRT.

TRT is a software development kit (SDK) that enables fast inference for GPUs developed by Nvidia. When the model is converted to TF-TRT and TRT, the object accuracy is slightly lower than that of the existing model, but it can provide efficiency in terms of latency and power consumption that is relatively low when compared to the desktop environment in the low specification of the Nvidia Jetson platform. However, TRT is a technology that can be used not only by the Nvidia Jetson platform but also by GPUs with Tensor Cores. In other words, if Tensor Cores are included in a high-performance GPU (e.g., RTX 3090) that does not use the Jetson Nvidia platform, you can directly build and use the TRT [42]. In addition, if you use a high-performance GPU, it is an environment using a desktop, and the power supply responsible for power is likely to be stable, it is not necessary to detect objects through TRT through relatively low object accuracy and reduced power consumption. Therefore, if deep learning-based object detection is performed on the Jetson Nvidia platform or in a general desktop environment, it is thought that the measurement results presented in this paper can efficiently conduct service and research on object detection.

6. Conclusions

Deep learning-based object detection technology using YOLO in embedded systems needs to be optimized for low latency and high accuracy detection rates and low power consumption. This paper applied TensorFlow and TRT, which are deep learning frameworks generally used in Nvidia Jetson AGX Xavier embedded systems. In addition, the deep learning framework was compared and analyzed by developing technology that can measure CPU utilization, GPU utilization, object detection accuracy, latency, and power consumption by fusion monitoring tools. As a result, TensorFlow used on the desktop can be applied to Nvidia Jetson AGX Xavier, but has the highest power consumption and high latency. In the case of TF-Lite, since it operates efficiently only on mobile devices, there is no GPU utilization in AGX Xavier, and it has a high latency to the extent that it cannot be used. In other words, if you are conducting deep learning services and research on mobile devices, it seems efficient to use TF-Lite. Additionally, if it is an embedded system with built-in Tensor Cores, it is most efficient to use TF-TRT and pure TRT. In the case of pure TRT, it does not utilize the TensorFlow library, and supports Python programming. In other words, if you have insufficient implementation knowledge of the TensorFlow deep learning framework, it seems most efficient to use pure TRT. Therefore, we will gradually study guidelines for the efficient use of deep learning frameworks with large datasets such as Open Images and Image Net, starting with comparative analysis with TensorFlow by applying PyTorch libraries to Nvidia Jetson AGX Xavier.

Author Contributions: Conceptualization, D.-J.S. and J.-J.K.; Methodology, D.-J.S. and J.-J.K.; Software, D.-J.S.; Validation, J.-J.K.; Formal analysis, D.-J.S. and J.-J.K.; Investigation, D.-J.S.; Resources, J.-J.K.; Data curation, D.-J.S.; Writing—original draft preparation, D.-J.S.; Writing—review and editing, J.-J.K.; Visualization, D.-J.S.; Supervision, J.-J.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Lu, H.; Li, Y.; Chen, M.; Kim, H.; Serikawa, S. Brain intelligence: Go beyond artificial intelligence. *Mob. Netw. Appl.* **2018**, *23*, 368–375. [CrossRef]
2. Nasteski, V. An overview of the supervised machine learning methods. *Horiz. B* **2017**, *4*, 51–62. [CrossRef]
3. Hwang, J.J.; Jung, Y.H.; Cho, B.H.; Heo, M.S. An overview of deep learning in the field of dentistry. *Imaging Sci. Dent.* **2019**, *49*, 1–7. [CrossRef] [PubMed]
4. Zou, Z.; Shi, Z.; Guo, Y.; Ye, J. Object detection in 20 years: A survey. *arXiv* **2019**, arXiv:1905.05055. [CrossRef]
5. Buber, E.; Banu, D.I.R.I. Performance analysis and CPU vs GPU comparison for deep learning. In Proceedings of the 2018 6th International Conference on Control Engineering & Information Technology (CEIT), Istanbul, Turkey, 25–27 October 2018; pp. 1–6. [CrossRef]
6. Chetlur, S.; Woolley, C.; Vandermersch, P.; Cohen, J.; Tran, J.; Catanzaro, B.; Shelhamer, E. cudnn: Efficient primitives for deep learning. *arXiv* **2014**, arXiv:1410.0759. [CrossRef]
7. Zhao, Z.Q.; Zheng, P.; Xu, S.T.; Wu, X. Object detection with deep learning: A review. *IEEE Trans. Neural Netw. Learn. Syst.* **2019**, *30*, 3212–3232. [CrossRef] [PubMed]
8. Pham, M.T.; Courtrai, L.; Friguet, C.; Lefèvre, S.; Baussard, A. YOLO-Fine: One-stage detector of small objects under various backgrounds in remote sensing images. *Remote Sens.* **2020**, *12*, 2501. [CrossRef]
9. Li, Z.; Peng, C.; Yu, G.; Zhang, X.; Deng, Y.; Sun, J. Light-head r-cnn: In defense of two-stage object detector. *arXiv* **2017**, arXiv:1711.07264. [CrossRef]
10. Bochkovskiy, A.; Wang, C.Y.; Liao, H.Y.M. Yolov4: Optimal speed and accuracy of object detection. *arXiv* **2020**, arXiv:2004.10934. [CrossRef]
11. Embedded Systems with Jetson. Available online: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/ (accessed on 5 February 2022).
12. Shuai, X.; Shen, Y.; Tang, Y.; Shi, S.; Ji, L.; Xing, G. Millieye: A lightweight mmwave radar and camera fusion system for robust object detection. In Proceedings of the International Conference on Internet-of-Things Design and Implementation, Charlottesville, VA, USA, 18–21 May 2021; pp. 145–157. [CrossRef]
13. Xun, D.T.W.; Lim, Y.L.; Srigrarom, S. Drone detection using YOLOv3 with transfer learning on NVIDIA Jetson TX2. In Proceedings of the 2021 Second International Symposium on Instrumentation, Control, Artificial Intelligence, and Robotics (ICA-SYMP), Bangkok, Thailand, 20–22 January 2021; pp. 1–6. [CrossRef]
14. Jeon, J.; Jung, S.; Lee, E.; Choi, D.; Myung, H. Run your visual-inertial odometry on NVIDIA Jetson: Benchmark tests on a micro aerial vehicle. *IEEE Robot. Autom. Lett.* **2021**, *6*, 5332–5339. [CrossRef]
15. Kortli, Y.; Gabsi, S.; Voon, L.F.L.Y.; Jridi, M.; Merzougui, M.; Atri, M. Deep embedded hybrid CNN-LSTM network for lane detection on NVIDIA Jetson Xavier NX. *Knowl. Based Syst.* **2022**, *240*, 107941. [CrossRef]

16. Valladares, S.; Toscano, M.; Tufiño, R.; Morillo, P.; Vallejo-Huanga, D. Performance evaluation of the Nvidia Jetson Nano through a real-time machine learning application. In Proceedings of the International Conference on Intelligent Human Systems Integration, Palermo, Italy, 22–24 February 2021; pp. 343–349. [CrossRef]
17. Mittal, S. A Survey on optimized implementation of deep learning models on the NVIDIA Jetson platform. *J. Syst. Arch.* **2019**, *97*, 428–442. [CrossRef]
18. Bokovoy, A.; Muravyev, K.; Yakovlev, K. Real-time vision-based depth reconstruction with nvidia jetson. In Proceedings of the 2019 European Conference on Mobile Robots (ECMR), Prague, Czech Republic, 4–6 September 2019; pp. 1–6. [CrossRef]
19. Nara, M.; Mukesh, B.R.; Padala, P.; Kinnal, B. Performance evaluation of deep learning frameworks on computer vision problems. In Proceedings of the 2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI), Tirunelveli, India, 23–25 April 2019; pp. 670–674. [CrossRef]
20. Shams, S.; Platania, R.; Lee, K.; Park, S.J. Evaluation of deep learning frameworks over different HPC architectures. In Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), Atlanta, GA, USA, 5–8 June 2017; pp. 1389–1396. [CrossRef]
21. Süzen, A.A.; Duman, B.; Şen, B. Benchmark analysis of jetson tx2, jetson nano and raspberry pi using deep-cnn. In Proceedings of the 2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA), Ankara, Turkey, 26–28 June 2020; pp. 1–5. [CrossRef]
22. Baller, S.P.; Jindal, A.; Chadha, M.; Gerndt, M. DeepEdgeBench: Benchmarking Deep Neural Networks on Edge Devices. In Proceedings of the 2021 IEEE International Conference on Cloud Engineering (IC2E), San Francisco, CA, USA, 4–8 October 2021; pp. 20–30. [CrossRef]
23. Verma, G.; Gupta, Y.; Malik, A.M.; Chapman, B. Performance evaluation of deep learning compilers for edge inference. In Proceedings of the 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Portland, OR, USA, 17–21 June 2021; pp. 858–865. [CrossRef]
24. Lin, T.Y.; Maire, M.; Belongie, S.; Hays, J.; Perona, P.; Ramanan, D.; Dollár, P.; Zitnick, C.L. Microsoft coco: Common objects in context. In Proceedings of the European Conference on Computer Vision, Zurich, Switzerland, 6–12 September 2014; pp. 740–755. [CrossRef]
25. Pascal VOC Dataset Mirror. Available online: https://pjreddie.com/projects/pascal-voc-dataset-mirror (accessed on 15 March 2022).
26. YOLOv4 Darknet. Available online: https://github.com/AlexeyAB/darknet (accessed on 15 March 2022).
27. Recent trends in artificial intelligence projects. Available online: https://www.itfind.or.kr/WZIN/jugidong/1899/file2522744291233853764-189902.pdf (accessed on 1 February 2022).
28. TensorFlow-Lite Guide. Available online: https://www.tensorflow.org/lite/guide?hl=en (accessed on 17 February 2022).
29. TensorFlow-Lite Converter. Available online: https://www.tensorflow.org/lite/convert/index (accessed on 17 February 2022).
30. Nvidia TensorRT Introduction. Available online: https://developer.nvidia.com/tensorrt (accessed on 17 February 2022).
31. TensorFlow-TensorRT Documentation. Available online: https://docs.nvidia.com/deeplearning/frameworks/tf-trt-user-guide/index.html (accessed on 24 February 2022).
32. Yolov4 TensorFlow Source Code. Available online: https://github.com/hunglc007/tensorflow-yolov4-tflite (accessed on 15 February 2022).
33. Yolov4 TensorRT Source Code. Available online: https://github.com/jkjung-avt/tensorrt_demos (accessed on 15 February 2022).
34. Jetson Platform Monitoring Tool. Available online: https://github.com/rbonghi/jetson_stats (accessed on 12 February 2022).
35. Jetpack SDK. Available online: https://developer.nvidia.com/embedded/jetpack (accessed on 10 January 2022).
36. NVPModel Clock Configuration. Available online: https://info.nvidia.com/rs/156-OFN-742/images/Jetson_AGX_Xavier_New_Era_Autonomous_Machines.pdf (accessed on 10 January 2022).
37. Jetson Installing TensorFlow. Available online: https://docs.nvidia.com/deeplearning/frameworks/install-tf-jetson-platform/index.html (accessed on 24 February 2022).
38. Yolov4-Native Weights Files. Available online: https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v3_optimal/yolov4.weights (accessed on 19 January 2022).
39. Yolov4-Tiny Weights Files. Available online: https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v4_pre/yolov4-tiny.weights (accessed on 15 March 2022).
40. Convert2Yolo Source Code. Available online: https://github.com/ssaru/convert2Yolo (accessed on 15 March 2022).
41. TensorFlow-Lite GPU Delegate. Available online: https://www.tensorflow.org/lite/performance/gpu (accessed on 7 February 2022).
42. Nvidia TensorRT Install Guide. Available online: https://docs.nvidia.com/deeplearning/tensorrt/install-guide/index.html (accessed on 17 February 2022).