Embedded Selforganizing Systems

Evaluation of Propeller Inspection Using Different Deployment Strategies

Ghita Ikmel
Signals Systems & Components Laboratory (LSSC)
Faculty of Science and Technology
University Sidi Mohamed Ben Abdellah
Fez, Morocco
ghita.ikmel@usmba.ac.ma

Mohamed Salim Harras
Department of Computer Engineering
Chemnitz University of Technology
Chemnitz, Germany
mohamed-salim.harras@informatik.tu-chemnitz.de

Wolfram Hardt
Department of Computer Engineering
Chemnitz University of Technology
Chemnitz, Germany
wolfram.hardt@informatik.tu-chemnitz.de

Najiba El Amrani El Idrissi
Signals Systems & Components Laboratory (LSSC)
Faculty of Science and Technology
University Sidi Mohamed Ben Abdellah
Fez, Morocco
najiba.elamrani@usmba.ac.ma

***Abstract*— In recent years, the use of Unmanned Aerial Vehicles (UAVs) for various applications has increased significantly. Among these applications, the inspection of infrastructures using UAVs has become a prominent area of research. This paper evaluates the efficiency of the YOLOv5 algorithm for propeller inspection. The algorithm's deployment across various platforms such as PC, Google Colab, and Jetson Nano is examined, with a focus on different deployment formats like PyTorch, ONNX, TensorFlow Lite, and others. The study highlights the often-overlooked importance of the deployment phase in the development of AI models and underscores its significance for the practical application of AI in real-world scenarios.**

Keywords*— Computer vision, algorithm deployment, propeller inspection, Deployment strategies, efficiency improvement

I. INTRODUCTION

Artificial Intelligence (AI) has changed how we use information quickly and effectively. AI uses special algorithms and learning methods to handle a lot of data fast, giving us new insights instantly [1]. This is useful in many areas, like industrial quality inspections and autonomous driving, where AI helps make quick and smart decisions by understanding complex situations fast and adapting to changes.

The real-time capabilities of AI extend beyond data analysis, encompassing applications like natural language processing, computer vision, and speech recognition [2]. This makes devices like self-driving cars and smart systems respond immediately to what we say or do, making our experience with them smooth and quick [3]. As AI grows, its fast response will become even more important in different industries. It will help create systems that can quickly adjust to our fast-changing world.

Putting AI models into use is very important for bringing AI into real-life uses, and there are different ways to do this. Using containerization, like with Docker, packages the model so it works the same in different places [4]. This makes it easier to use, allows it to handle more work, and uses resources well. Serverless computing, like with AWS Lambda, removes the need to manage the infrastructure. It adjusts the resources needed based on how much it's used and is cost-effective [5]. RESTful APIs let different applications talk to the AI model easily through standard internet requests. Edge computing puts models closer to where data is collected, like in IoT devices or local servers, which makes response times quicker, crucial for things like IoT and self-driving cars [6]. These methods help AI respond quickly in real scenarios, improving things like chatbots, recommendation systems, and fraud detection by making AI models give fast and efficient predictions.

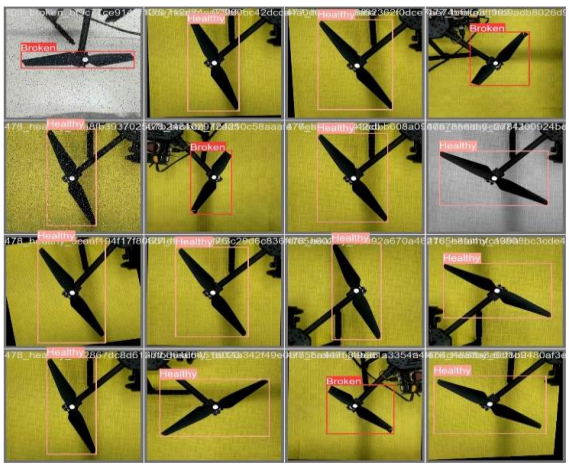

Fig. 1. UAV propeller inspection using YOLOv5 model [7]

Propeller inspection is at the core of our exploration. Our research builds upon this foundation and focuses on the effectiveness of model deployment techniques. By leveraging the pre-trained weights from the research [7], which were trained with an active learning strategy [8], we aim to enhance the real-time detection capabilities of YOLOv5, making the model faster in propeller inspection scenarios (Fig. 1).

In the next sections, we will talk more about how we put the model to use and its effects on quick AI responses. We will evaluate our approach to using YOLOv5 for inspecting propellers, looking at how using pre-developed techniques can make the model better and faster in this specific use. Our goal is to explore different ways to use the model and see how they affect its performance, hoping to improve AI technologies in the important area of propeller inspection. The paper is structured to first discuss the different deployment approaches and their current challenges followed by an in-depth analysis of our proposed methodology and concluding with a comprehensive evaluation of our approach's effectiveness in the use case of propeller inspection.

II. Literature Review

Checking and monitoring propellers is very important for keeping airplanes and drones (Unmanned Aerial Vehicles or UAVs) safe and working well. In this part of our study, we look at two main areas of recent research. First, we discuss different methods used to inspect propellers. There are three types: methods based on data, sound, and vision. Next, we explore deployment strategies, especially in computer vision. This means looking at the latest developments and challenges. By studying these two parts, we want to understand how new technologies like machine learning and practical applications come together. This is important for making these inspections more effective and reliable, leading to safer and more dependable flying.

III. Types Of Propeller Inspection

A. Data-Based Inspection:

Data-based propeller inspection involves the application of data analysis techniques to assess the condition and performance of propellers. Methods such as time series analysis, machine learning, and artificial intelligence are applied to data generated by onboard sensors. This approach enables continuous monitoring and early detection of any anomalies [9][22]. However, this type of inspection requires the usage of sensors on the UAV.

B. Audio-Based Inspection:

Audio-based inspection focuses on analyzing sounds emitted by propellers. Advanced signal processing and machine learning techniques are employed to identify characteristic sound patterns. This can provide insights into potential mechanical issues or variations in performance, contributing to preventive maintenance [10]. These methods are able to detect from light to severe cracks on the propeller but are very sensitive to noise. The sensors used for these methods can either be mounted on the UAV [21] or placed in an external testing environment [23].

C. Vision-Based Inspection:

Vision-based inspection relies on the use of cameras and computer vision techniques to visually assess the condition of propellers. Deep learning models, such as YOLOv5, can be deployed for real-time anomaly detection. This approach offers a detailed evaluation of propeller structure and potential damages [11]. It is also able to detect failures or defects on the propeller only by placing cameras in an external testing environment facing the propeller [8]. Such advancements could lead to a new era of aeronautic maintenance where inspections are more thorough, less intrusive, and significantly more efficient.

IV. Model Deployment

Model deployment, particularly in computer vision, is key for models to work efficiently in real time on different platforms. There are various ways to deploy models, like PyTorch, ONNX, OpenVINO, and TensorRT. As shown in TABLE I, these methods help set up an environment where AI models can run in real time. Some methods are better for cloud computing, while others are good for in-device use, like TensorFlow Lite. Choosing the best method depends on factors like how much computing power is available, how quickly the model needs to respond, and the specific conditions where it will be used.

TABLE I. Summary of Yolov5 Model Deployment Formats

Deployment Format	Description
PyTorch [13]	Flexible deep learning framework with dynamic tensors, GPU support, and autograd.
OpenVINO [14]	Intel-developed toolkit for optimizing and deploying models on diverse hardware.
TorchScript [15]	PyTorch feature converting models into a portable format for cross-platform deployment.
ONNX [16]	Interoperable file format for representing models independently of the framework.
TensorFlow Lite [17]	Lightweight TensorFlow version for mobile and embedded systems with optimized features.
TensorRT [18]	NVIDIA library for accelerating deep learning model deployment on NVIDIA GPUs.

Furthermore, the integration of advanced deployment techniques, such as edge computing, could further reduce latency and increase responsiveness, particularly in environments where quick decision-making is crucial. This advancement is particularly relevant in scenarios where every millisecond counts, such as in autonomous vehicle navigation or real-time monitoring of critical infrastructure.

As shown in Fig. 2, the choice of the deployment strategy affects the overall performance of the model. A comparison between four different neural networks shows that OpenVINO outperforms standard formats such as Pytorch in terms of inference latency. The study [19] concludes that the model’s

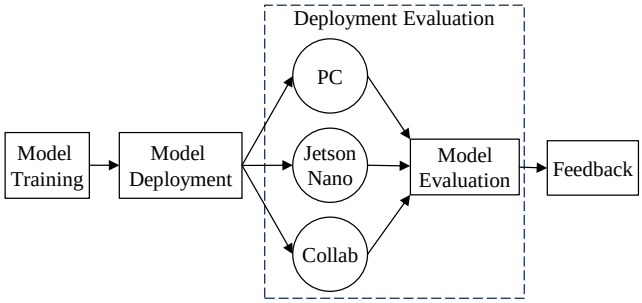

Fig. 3. Evaluation Process Flowchart

speed can be optimized by adopting an OpenVINO-based deployment strategy.

The literature demonstrates growing interest in the deployment phase, emphasizing its impact on the practicality of computer vision algorithms. Special attention is given to enhancing the real-time capabilities of deployed models, contributing to more effective propeller monitoring and proactive failure prevention.

V. Experimental Setup

This section, we will provide a detailed description of the three platforms on which we deployed the YOLOv5 model: PC, Google Colab, and Jetson Nano [12]. Each platform has specific hardware, resources, and computational power characteristics, which influence the performance and results achieved during model deployment TABLE II).

In order to evaluate the efficiency of the AI model, the flowchart in Fig. 3 was adopted to provide a concise and clear depiction of the entire process, from training the model to deploying it the different platforms. The cycle also incorporates a deployment evaluation, with a focus on performance metrics, conducted through Weights & Biases [20].

At the end of this process, we encounter feedback, which is essential for continuously refining and improving our models. It plays a crucial role in our optimization and adaptation efforts to address real-world challenges more effectively. The Pytorch model from [8] and a recorded video, showcasing UAV propellers, are used for the adopted experiment.

TABLE II. Comparison Of Computing Platforms

Platform	Processor	RAM	GPU
Personal Computer	Intel Core i5-7200U	8 GB	Not specified
Google Colab	Intel(R) Xeon(R) Platinum 8259CL	13 GB	NVIDIA T4
Jetson Nano	NVIDIA Tegra X1	4 GB	Not specified

VI. Experimental Methodology

In this section, we will provide a detailed description of the experimental methodology we followed to evaluate the performance of the YOLOv5 model on different platforms.

The evaluation process of deployment strategies, shown in Fig. 4, begins with the 'Selection of Deployment Strategy', where various deployment formats like PyTorch, TensorFlow Lite, and TensorRT are assessed for their suitability in propeller inspection. This crucial phase considers factors such as real-time processing needs and hardware compatibility.

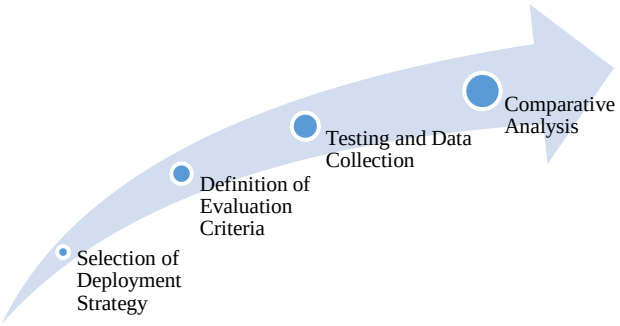

Fig. 4. Workflow of YOLOv5 Model Deployment and Performance Evaluation

Following this, the 'Definition of Evaluation Criteria' stage establishes benchmarks to evaluate each strategy's performance, focusing on aspects like computational efficiency and scalability. The subsequent 'Testing and Data Collection' phase involves rigorous testing of each strategy to gather empirical data on their performance. Finally, in the 'Comparative Analysis' stage, this data is meticulously analysed, comparing the strategies against the established criteria to discern the most effective deployment strategy that balances technical prowess with practical applicability in the propeller inspection use case.

To understand how the models perform under real-world conditions, we conducted evaluations using some key metrics. These metrics include a time analysis that involve three aspects: Inference, Pre-processing, Non-Maximum Suppression (NMS). In addition, a thorough analysis of the resource management aspect is also taken into consideration.

- Inference Time: Inference time measures the speed at which the model can detect objects in an image. Measuring this time ensures that the model can operate in real-time or according to the requirements of your application.
- Pre-processing Time: Refers to the steps and techniques used to prepare, clean, and transform raw data into more suitable and actionable data before using it in an analysis or modeling task.
- Non-Maximum Suppression (NMS) Time: This post-processing step is performed after object detections have been generated by a detection algorithm. NMS aims to reduce the number of detections by removing strongly overlapping ones, retaining only the most reliable detections.

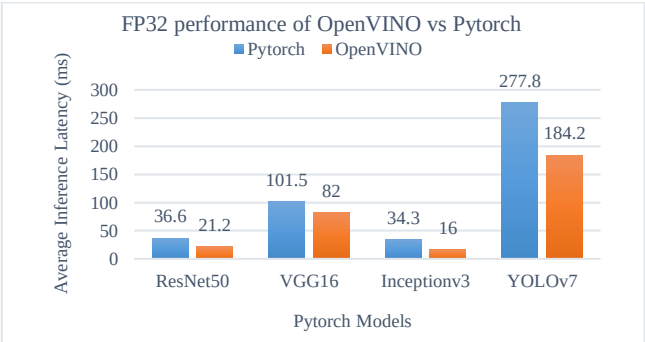

Fig. 2. Average inference latency (in milliseconds) for 100 runs after 15 warm-up iterations on an 11th Gen Intel(R) Core (TM) i7-1185G7 @ 3.00GHz [19]

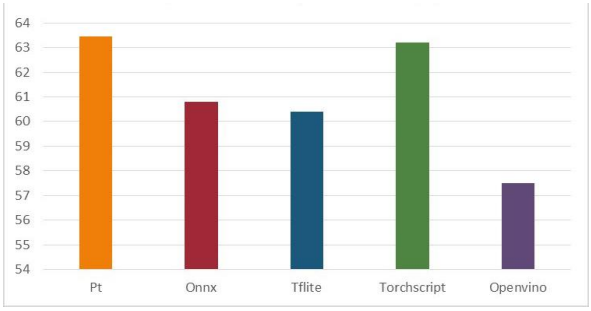

Fig. 5. System memory utilization [%]

- Resource management: Each platform may have its own specific metrics. For example, we evaluated the usage of hardware resources such as memory and the processor.

VII. Results And Discussions

In this part, we talk about the results we got from using the YOLOv5 model on different systems. We focused on how well it worked, how fast it made predictions, and how it managed resources. For our test, we used a 30-second video taken by a drone controlled by our university. We ran the models on this video to see how they performed.

A. Deployment Results on CPU

1) Deployment Results on PC

In TABLE III, we show the results of using YOLOv5 on a personal computer. We looked at how long it takes to prepare the data (pre-processing), make predictions (inference), and do non-maximum suppression (NMS) with different formats. It can be noticed that the inference time consumes most of the time, with the highest time found using Tensorflow Light and the lowest using Openvino.

TABLE III. Deployment Results on PC

Formats	**Pre-processing** [ms]	**Inference** [ms]	**NMS per image** [ms]
.pt	2	228.2	0.6
.onnx	3	260.7	1.1
.tflite	2.6	450.1	0.9
.torchscript	2.6	375	0.8
.openvino	2.4	213	0.6

Fig. 5 gives a detailed view of how much work each part of the computer's processor (CPU) is doing, shown in the 'System CPU Utilization' chart. This chart helps us see how each part of the CPU is contributing and where we might make improvements or find stress points that could affect performance.

CPU utilization differs from one deployment strategy to another. In what concerns the desktop test environment, it can be noticed that among the four cores of the CPU, the model is able to distribute the task in a fair manner. However, the most optimal results is characterized by its ability to balance high performance with efficient resource management. In other words, the optimal model demonstrates a harmonious distribution of workload across all four cores. In this case, we found that the Torchscript and Openvino perfom well at balancing the load between the different cores of the CPU.

The Fig. 8 presents an overview of System Memory Utilization (%), showcasing how the computer's system memory is utilized over time.

In analyzing the results of TABLE IV, we see that the .onnx format takes the longest to prepare data (3 ms), followed closely by .tflite and .torchscript (both 2.6 ms). Other formats have pre-processing times between 2 and 2.4 ms. For making predictions, .tflite takes the longest (450.1 ms), followed by .torchscript (375 ms), .onnx (260.7 ms), .openvino (213 ms), and finally .pt (228.2 ms). The NMS time for most formats is about the same, around 0.8 to 1.1 ms, except for .pt and .openvino, which are faster at 0.6 ms.

Looking at the CPU and memory usage charts, we see how YOLOv5 performs on a PC. It uses different CPU cores effectively. Each framework in the chart is shown with four values, for the four CPU cores. For example, Torchscript uses about 50% of the CPU cores, more than other frameworks. TFLite uses about 30% of the CPU, but it doesn't spread the workload evenly across the cores.

2) Deployment Results on Jeston Nano

TABLE IV displays deployment results on Jetson Nano, including pre-processing, inference, and NMS times. Examining the data, it's notable that the .pt format demonstrates relatively better performance in terms of inference time and NMS per image. On the other hand, the .tflite format exhibits the longest time for inference, while the .torchscript format also records higher inference times. The .openvino format stands out as one of the fastest for inference.

TABLE IV. Deployment Results on NVIDIA Jetson Nano

Formats	**Pre-processing** [ms]	**Inference** [ms]	**NMS per image** [ms]
.pt	3.9	596.2	2.1
.onnx	7.3	767.0	3.7
.tflite	5.3	2093.9	2.4
.torchscript	5.9	1015.0	3.6
.openvino	4.7	887.6	2.3

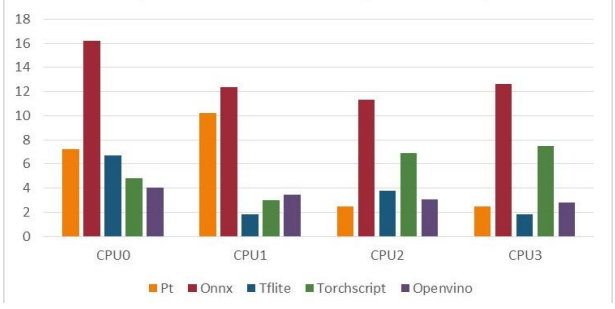

Fig. 6. System CPU utilization (per core) [%]

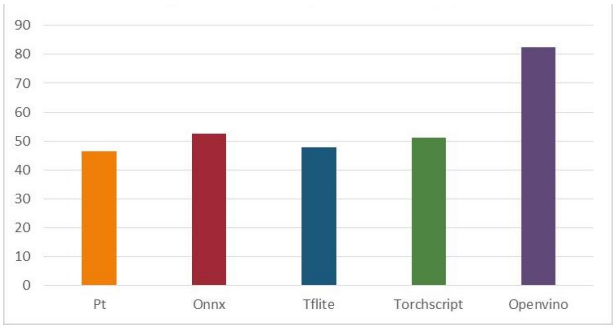

Fig. 7. System memory utilization [%]

Fig. 6 and Fig. 7 show how much of the CPU and memory the models use during the tests. Different models use the CPU differently. For example, the ONNX model uses about 20% of the CPU cores, while TFLite uses about 10%. However, TFLite doesn't use the CPU cores as efficiently, which could be improved. The .Openvino model uses the most system memory, followed by the .Onnx model. This might be because the .Openvino model (27.3MB) is larger than other formats like .pt (13.7MB), so it needs more memory.

When we compare how each model uses the CPU and memory, torchscript seems to be the most efficient. It uses CPU resources well, which helps it perform better overall. This fits with the idea of AI models that are efficient in using resources, specifically in the context of edge devices.

B. Deployment Results on GPU

TABLE V presents deployment results of how we used YOLOv5 on Google Colab. It includes times for getting the data ready (pre-processing), making predictions (inference), and non-maximum suppression (NMS).

The Fig. 9 provides a visual overview of the graphics processing unit (GPU) temperature over time. On the other hand, the Fig. 10 provides a visual representation of how much memory the GPU uses as it works on different tasks. This graph, put together by Weights and Biases, helps us see how the GPU's memory use changes, which tells us about how efficiently the memory is being used. The Fig. 11 represents the workload that the graphics processor (GPU) supports in relation to its maximum capacity.

In our study of how YOLOv5 works on Google Colab, we noticed some interesting differences between the formats used. When we look at the time it takes to get data ready (pre-processing), we see that it's pretty similar across most formats, between 0.8 to 1 ms.

TABLE V. DEPLOYMENT RESULTS ON GOOGLE COLLAB

Formats	**Pre-processing** [ms]	**Inference** [ms]	**NMS per image** [ms]
.pt	0.8	11.6	2.1
.onnx	1.0	17.3	1.8
.tflite	0.9	343.8	1.5
.torchscript	1.0	12.4	1.6
.openvino	0.8	185	1.5
.engine	1.0	10.6	2.1

If we look at the time for making predictions (inference), the .tflite format takes the longest, about 343.8 ms. This is followed by .openvino (185 ms), .onnx (17.3 ms), .Torchscript (12.4 ms), .pt (11.6 ms), and .engine (10.6 ms). An interesting point is that the ".pt" and "*.engine" formats take the longest time for NMS per image, about 2.1 ms, which is slightly more than the other formats.

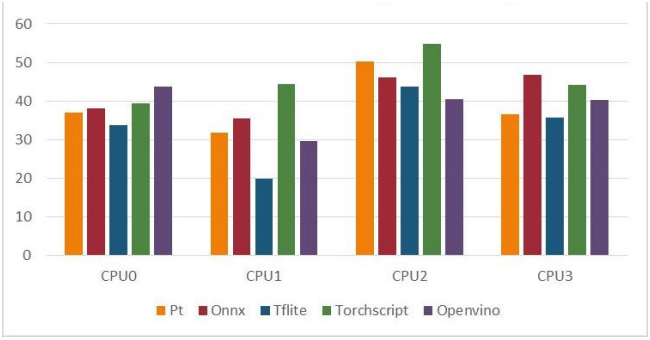

Fig. 8. System CPU utilization (per core) [%]

We also looked at how the GPU temperature changes with different models. We found that the longer a model takes to make predictions, the hotter the GPU gets. For example, the "Engine" model, which has a shorter prediction time, only makes the GPU reach a maximum of 37 degrees Celsius. On the other hand, the "Tflite" model, with the longest prediction time, causes the GPU temperature to go from 42 to 65 degrees Celsius.

Our study of GPU memory usage and GPU efficiency shows an interesting pattern. We noticed that when a model uses more memory, it usually takes less time to make predictions. For instance, the 'Engine' model uses the most memory, followed by the 'ONNX' and 'PyTorch' models. However, the 'Tflite' model, even though it takes the longest to make predictions, uses a steady amount of memory.

When we look at how efficiently the GPU is used, there's a big difference between models. The 'Engine' model, even though it uses a lot of memory, is quite good at using the GPU. It's followed closely by the 'Torch' and 'PyTorch' models in terms of efficiency. In contrast, the 'Tflite' and 'OpenVINO' models don't use the GPU much, as shown by their low utilization percentages, like 0.4%. This helps explain the performance results we see in the TABLE V.

C. Summary of Findings

To conclude, TABLE VI presents how fast YOLOv5 makes predictions (inference times in milliseconds) on different systems, like a local PC, Google Colab, and Jetson Nano.

The results obtained from our deployment of YOLOv5 on various platforms reflect significant improvements in how well and quickly the model works on these platforms. For example, the original model (.pt) used to take 228.2 ms to make predictions on a local PC, but on Google Colab, it's much faster at 11.6ms. This shows that the deployment strategy highly reflects on the model prediction speed.

This substantial improvement in inference time, especially on Google Colab, signifies the adaptability of our optimization strategies to cloud-based platforms. The transition from a local PC to Google Colab, with its GPU infrastructure, resulted in a noticeable enhancement in performance. Our model's ability to effectively leverage available hardware resources translated into a smoother and faster inference process.

TABLE VI. DEPLOYMENT RESULTS

Formats	**PC** (CPU)	**Google Colab** (GPU)	**Jetson Nano** (CPU)
.pt	228.2	11.6	596.2
.onnx	260.7	17.3	767.0
.tflite	450.1	343.8	2093.9
.torchscript	375	12.4	1015.0
.openvino	213	185	887.6
.engine	-	10.6	-

Furthermore, comparing deployment formats on different hardware, such as the transition from a CPU device to a GPU device, highlights the versatility of YOLOv5. For example, the TensorRT model on a GPU device achieved an impressive inference time of around 10,5 ms, demonstrating the algorithm's efficiency in more powerful hardware environments.

Apart from being faster, our results also show how well the model uses CPU cores. When we looked at how the CPU was used on a local PC, we saw different ways each framework used the CPU cores. For example, ONNX used 20% of the CPU cores effectively, while TFLite used about 10% but not as evenly. This suggests we could improve how resources are allocated, especially for use on edge devices.

By making our original model more efficient, we've made it more relevant. The faster response time helps with real-time monitoring. Also, using energy more efficiently makes the system more sustainable, which is important when integrating AI solutions.

Lastly, using CPU cores efficiently is key for real-time performance. Our results show how important it is to choose the right format for the hardware we're using. This is especially true for environmental monitoring, as using resources efficiently helps the AI model work better in real-world situations.

VIII. CONCLUSION

In conclusion, this study delves into the deployment and performance analysis of the YOLOv5 real-time object

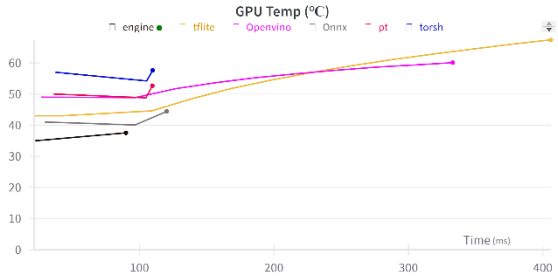

Fig. 9. GPU temperature [C^0]

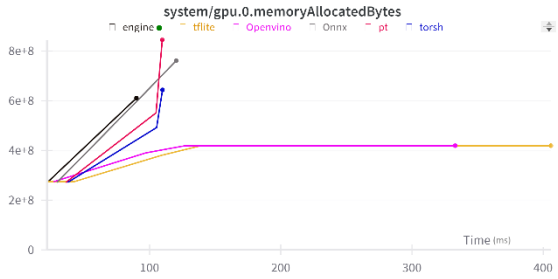

Fig. 10. GPU memory Allocated Bytes

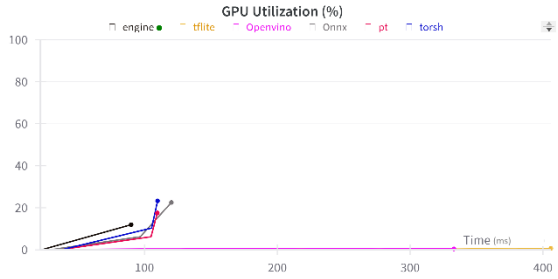

Fig. 11. GPU utilization [%]

detection model on diverse platforms, namely a Personal Computer (PC), Google Colab, and Jetson Nano. The exploration encompasses a comprehensive evaluation of the model's efficiency, execution time, and resource utilization across these platforms, shedding light on the implications for sustainable environmental monitoring.

The results showcase the adaptability and versatility of the YOLOv5 model across different hardware environments. Notably, the optimization efforts have led to a substantial reduction in inference time, demonstrating the model's efficiency in responding to environmental events, a crucial aspect for real-time monitoring applications. The transition from a local unit to the cloud, leveraging GPU infrastructure, significantly enhances performance, emphasizing the importance of hardware resources in model deployment.

Furthermore, the study highlights the significance of effective CPU core utilization, especially in the context of sustainable environmental monitoring. Different deployment formats exhibit distinct behaviors in distributing the workload among CPU cores, emphasizing the importance of choosing deployment strategies that align with hardware characteristics for optimal and sustainable AI deployments.

The findings underscore the broader implications for AI-based solutions in sustainable environmental monitoring.

Beyond inference times, considerations of resource distribution and utilization are crucial for achieving optimal performance and adaptability in real-world applications. As the world grapples with environmental challenges, the integration of efficient and sustainable AI solutions becomes imperative, and this study contributes valuable insights to this evolving field.

ACKNOWLEDGMENTS

I would like to express our heartfelt gratitude to the inviting faculty for their support and encouragement in conducting this research. Their unwavering commitment to fostering academic and research initiatives has been instrumental in the success of this project.

We would also like to extend our appreciation to SAXEED program for their generous financial support. Their investment in our research has allowed us to explore innovative solutions and contribute to the field of environmental monitoring. Thank you for believing in the importance of our work and for making this research endeavor possible.

REFERENCES

[1] S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach," Pearson, 2010.

[2] D. Jurafsky and J. H. Martin, "Speech and Language Processing," Draft of 3rd edition, 2020.

[3] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," in Advances in Neural Information Processing Systems.

[4] Docker. [Online]. Available: https://www.docker.com/

[5] AWS Lambda. [Online]. Available: https://aws.amazon.com/lambda/

[6] W. Shi et al., "Edge Computing: Vision and Challenges," IEEE Internet of Things Journal, 2016.

[7] S. Saleh, B. Battseren, M. S. Harras, A. Chaudhry, and W. Hardt, "Toward Accurate and Efficient Burn Marks Inspection for MAV Using Active Learning."

[8] M. S. Harras, S. Saleh, B. Battseren, W. Hardt, "Vision-based Propeller Damage Inspection Using Machine Learning."

[9] J. Smith et al., "Data-based Propeller Inspection Techniques: A Review," Journal of Propulsion Technology, vol. 25, no. 3, pp. 123-140, 2021.

[10] A. Johnson et al., "Audio-based Inspection for Propeller Anomaly Detection: A Comprehensive Survey," Acoustics in Transportation, vol. 15, no. 2, pp. 87-104, 2022.

[11] C. Brown et al., "Vision-based Propeller Inspection Using Deep Learning Models," IEEE Transactions on Industrial Electronics, vol. 48, no. 5, pp. 2311-2325, 2020.

[12] K. Miller et al., "Optimizing Deployment for Real-time Propeller Inspection: A Comparative Study of PyTorch, ONNX, and TensorRT," Proceedings of the IEEE International Conference on Robotics and Automation, 2019.

[13] PyTorch. [Online]. Available: https://pytorch.org/

[14] Intel. OpenVINO Toolkit [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/openvino-toolkit.html

[15] PyTorch. TorchScript [Online]. Available: https://pytorch.org/docs/stable/jit.html

[16] ONNX [Online]. Available: https://onnx.ai/

[17] TensorFlow. TensorFlow Lite [Online]. Available: https://www.tensorflow.org/lite

[18] NVIDIA. TensorRT [Online]. Available: https://developer.nvidia.com/tensorrt

[19] S. J. Kershaw Vishnudas Thaniel S. Devang Aggarwal, Natalie, "Faster inference for PyTorch models with OpenVINO Integration with Torch-ORT," Microsoft Open Source Blog, Dec. 01, 2022.

[20] L. Biewald, "Experiment tracking with weights and biases", 2020, software available from wandb.com. [Online]. Available: https://www.wandb.com

[21] G. Iannace, G. Ciaburro, and A. Trematerra, "Fault diagnosis for UAV blades using artificial neural network," Robotics, vol. 8, no. 3, 2019.

[22] A. Joshuva and V. Sugumaran, "Wind Turbine Blade Fault Diagnosis Using Vibration Signals through Decision Tree Algorithm," Indian J. Sci. Technol., vol. 9, no. 48, 2016.

[23] A. Altinors, F. Yol, and O. Yaman, "A sound based method for fault detection with statistical feature extraction in UAV motors," Appl. Acoust., vol. 183, 2021.