OCR Implementation in Jetson Nano

Pol Medina Arévalo

June 30, 2025

Abstract

This project explores the deployment of optimized deep learning-based OCR models on the NVIDIA Jetson Nano for real-time text extraction in industrial environments. The main objective is to balance accuracy and inference speed on resource-limited edge devices. We adapt and optimize OCR architectures using AI acceleration, then deploy and evaluate them on the Jetson Nano. Results show significant reductions in inference latency with minimal loss in recognition accuracy and inconsistency on some specific model platform executed in Jetson Nano. The project provides a benchmark of OCR performance on Jetson Nano and a reproducible optimization methodology, demonstrating the feasibility, with some caveats, of real-time OCR for industrial automation.

Keywords: Jetson Nano, optical character recognition, OCR, EasyOCR, PaddleOCR, deep learning, edge computing, model optimization, TensorRT, quantization, pruning, real-rime processing, industrial automation, embedded AI, inference acceleration, resource-constrained devices, computer vision, text extraction, benchmarking, deployment.

1 Problem definition

We pursue the implementation of optical character recognition (OCR) models, which involve converting different types of images into machine-readable text, powered by artificial intelligence (AI) on an NVIDIA Jetson Nano microprocessor. The Jetson Nano is designed specifically for AI applications, and due to its powerful GPU and AI-optimized architecture it is well-suited to this task. This leverages the Jetson Nano's capability for efficient parallel processing, making it an ideal platform for computer vision tasks offering a compact solution for edge processing where small form factors and portability are essential.

Within this goal there is a mandatory requirement: optimizing the task execution for speed and efficiency. The challenge is to ensure that these models not only perform correctly but also have extremely low inference times. This means, the project explores AI model acceleration techniques to reduce inference time while maintaining accuracy. The Jetson Nano's GPU combined with model optimization will allow achieving efficient OCR performance suitable for most industrial environments such as this project's use case.

We are focusing in a task useful for manufacturing and warehouse environments where product codes have to be scanned. Codes will consist of alphanumerical printed strings such as the one shown further in Figure 1. The OCR models will be used to digitalize these codes at high speed for further use according to the needs of the processes. In these scenarios, hundreds of products may need to be processed in a very short amount of time, making it necessary to deploy a system that can handle this task swiftly. Again, the Jetson Nano's device and other devices from the Jetson collection are a solid choice.

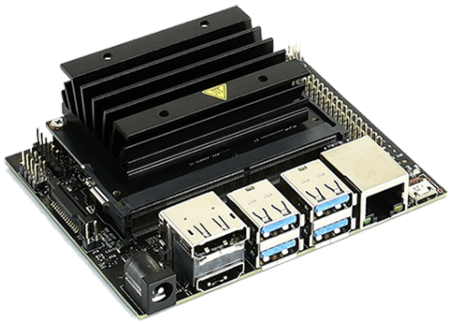


Fig. 1: Jetson Nano device image that showcases its compactness and portability.

Two different platforms will be used during the execution processes: a server with available GPUs and the Jetson Nano. The code will be first developed in standard commercial GPUs and it will be transferred to the Jetson Nano environment in where adaptation and integration will be done. The specifications of the cluster GPU and the Jetson device are specified in Table 1. TFLOPS stands for trillions of floating-point operations per second. LPDDR4 stands for Low Power Double Data Rate and GDDR6 stands for Graphics Double Data Rate. The latter is a high-speed memory while the first is a low power consumption mem-

• Contact E-mail: Pol.Medina@uab.cat
• Supervised by: Vanessa Moreno Font (Arquitectura i Tecnologia de Computadors)
• Academic Year 2024/25

ory. As it is shown, a standard GPU outgrows the Jetson Nano's processor in every aspect, forewarning about an enormous difference in processing time and capabilities.

The NVIDIA Jetson Nano device works with JetPack 4.6.6 [1], which is a software development kit (SDK) for NVIDIA Jetson devices. It provides everything needed to develop most of AI applications. These SDK includes an Ubuntu based OS (v18.04), CUDA support (v10.2) [2], cuDNN support (v8.2.1) [3] and a pre-installed instance of TensorRT (v8.2.1) [4]. These elements ensure an efficient execution of deep learning models leveraging the parallel processing power of the GPU processor and managing its acceleration.

	TFLOPS	CORES	VRAM	POWER
Jetson Nano	0.5	128	4GB LPDDR4	5-10W
RTX 3090	35.6	10,496	24GB GDDR6	350W

TABLE 1: Hardware specifications for available computation resources.

Three main objectives have been defined:

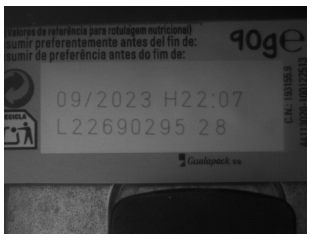
1. **Optimizing AI models in Jetson Nano** in order to achieve an efficient and high-speed inference of an OCR task in use case images.
2. **Benchmarking the efficiency of elected OCR models** in order to assess the acceleration quality and understand address further acceleration approaches. This will be done with baseline and optimized models in order to leverage the accomplished improvements.
3. **Providing a reproducible methodology** including developed source code and detailed and synthesized documentation for further experimentation and optimization with ready-to-run code, all collected in a public GitHub repository.

In summary, these contributions aim not only to advance the deployment of OCR systems in industrial edge environments, but also to provide a reproducible framework and practical insights for future research and real-world applications

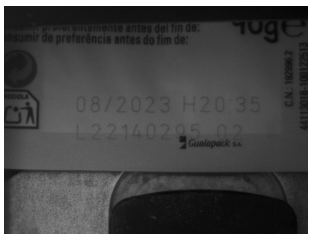
1.1 Use case data

The available dataset contains 1,798 images like the ones represented in Figures 2a and 2b. Along with the images, annotated bounding boxes coordinates in which the codes are contained as well as the ground-truth code are available for each image. The data has three main problems that have to be handled.

1. Target code can be overlapping with other text because of printing errors, as seen in Figure 2b.
2. Unneeded text surrounding the target code. Since many general OCR models detect any existing text in an image, text apart from the target can be detected and should be masked or removed.
3. Low contrast between the code's letter color and the background.



(a) Example of case use image with clear code.

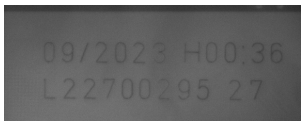


(b) Example of case use image with slightly occluded code.

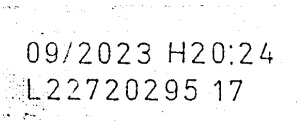
Fig. 2: Two different images displaying the available dataset images.

In order to avoid the mentioned problems, only 100 have been manually preselected from the whole set of images to be used for benchmarking. Given previous research done in code detection and external text omission, it has been assumed that this task will be done by an auxiliary model and has been ignored by cropping the image with the annotated bounding boxes coordinates.

In order to deal with low contrast between code and backgrounds a first approach tried thresholding the image and applying morphological operations to keep characters clear. An example can be seen in Figure 3b. However, preliminary executions showed how this method returned worse results since the models were originally trained with real-world data, so the processing was reverted and the raw grayscale images were used instead, as seen in Figure 3a.



(a) Image example with code cropping.



(b) Image example processed with OTSU thresholding and morphological operations in order to achieve cleaner codes.

Fig. 3: Two different images displaying the attempted preprocessing step done with the cropped image with a code and the image without further processing.

2 Current context on OCR in edge computing

When talking about Optical Character Recognition (OCR), achieving a balance between accuracy and computational efficiency is crucial, particularly when deploying models on resource-constrained devices like the NVIDIA Jetson Nano. Different possibilities in terms of architectures and platforms arise.

Transformer-based architectures can attain high accuracy, but they pose significant challenges for edge deployment due to their high computational and memory requirements. Cloud-based OCR solutions can leverage these models, but they increase costs and require constant server connectivity.

The NVIDIA Jetson Nano has been effectively utilized in various industrial settings for OCR and vision-related tasks

demonstrating its capability in real-world applications:

- **License Plate Recognition:** Implementations have been developed for real-time license plate detection and recognition using the Jetson Nano [5]. This approach resulted in 40 FPS for license plate detection and other 40 FPS for text recognition.
- **Warehouse Automation:** Embedded camera solutions powered by Jetson Nano [6] have been integrated into conveyor systems for OCR applications. The device used is a Jetson AGX Orin and achieves until 15 FPS using four cameras and a 5K resolution images, while being able to work in low-light conditions.
- **Smart Surveillance:** Devices like the SmartCam T1023 [7, 8], which are an embedded camera integrated along with the Jetson Nano or Jetson Orin are deployed for tasks including parking detection and surveillance. However, no measure of performance is delivered regarding this product.

These examples underscore the Jetson Nano's versatility and effectiveness in deploying OCR and vision applications within industrial contexts, but also show the lack of open-source code regarding the implementation of functional and efficient OCR models in the Jetson Nano.

This reinforces the need of releasing open-source and functional approaches to these applications. It is to know that some public GitHub repositories do exist, but their implementations have become deprecated and unusable after the latest version updates.

2.1 Information on Existing OCR Models

Recent evaluations highlight the trade-offs between accuracy and inference time among various OCR models, particularly in terms of efficiency for edge computing. Although several options exist, some of the most popular and supported are the ones shown in Table 2. CRNN stands for Convolutional Recurrent Neural Network and SVTR stands for Short Vision Transformer. Traditional type regards its primarily CPU-bound nature without relying in a Deep Learning architecture. Notice that in terms of latency, an approximation is done but its performance is tightly connected to the computing resources available. This performance is given taking a fully disposable GPU as reference.

Model	Acc. (%)	Lat. (s)	Type
EasyOCR	85	0.042	CRNN
PaddleOCR	90	0.110	SVTR
Tesseract OCR	95	Fast	Traditional
TrOCR	96	Fast	Transformer-based
Google Cloud Vision	95	0.063	Cloud-based

TABLE 2: Comparison of different OCR models. Acc. stands for "accuracy" and Lat. stands for "latency".

2.2 TensorRT acceleration

One of the cores of AI acceleration is TensorRT [4]. This is an SDK designed for high-performance deep learning inference. It is tailored to work with already trained models rather than accelerating inference by parameter tweaking during training. It optimizes a model architecture for the specific NVIDIA hardware being used and it is compatible with TensorFlow and PyTorch. The latter has a relatively new library implementation called torch2trt [9], which applies TensorRT conversion by using more accessible and understandable Python code, allowing a better implementation and traceability in practice.

3 METHODOLOGY

Five main stages have been settled for the execution of the project.

PHASE 1: Cluster GPU environment setup and preparation. This includes the installation of libraries, implementation of CUDA support, dataset parsing and testing of image processing approaches. Four weeks were expected to be spent in this phase, but only two were needed.

PHASE 2: Model selection and baseline development. The benchmarked models were selected at this phase and they were implemented in a "free" resource server. These models are evaluated and metrics are exported here. This phase expected four weeks of development and all four weeks were needed due to added testing between architectures for election.

PHASE 3: Jetson Nano deployment and adaptation. The defined scripts in the previous phase are aimed to be implemented in the Jetson Nano device. This includes the installation of libraries, dealing with unstable version compatibility and ensuring a correct use of the GPU processor.

PHASE 4: Model optimization. The baseline scripts will be modified to include model optimization. This was the most complex phase of the development. It was expected to be completed in six weeks, but in the end seven weeks were needed to fulfill the expectations. Additionally, further exploration and execution was done parallel to phase 5.

PHASE 5: Performance evaluation and documentation encapsulation in GitHub. The results will be gathered, discussed and all documentation needed to easily install dependencies in Jetson Nano for replication will be provided. This phase expected three weeks of workload but four were needed. Different metrics were proposed and tested as well as some unintended benchmarking was done in the end.

3.1 Elected models to implement in Jetson Nano

Since several options exist and a plausible implementation must be ensured the list has to be narrowed down. The options from Table 2 will be the ones taken as reference. This original list included EasyOCR, PaddleOCR, Tesseract, TrOCR and Google Cloud Vision.

Tesseract will be discarded. Its architecture is CPU focused so none of the advantages of Jetson Nano's GPU power will be taken profit of. This makes the use of this

model a non-sense. **Google Cloud Vision** will also be discarded. Its cloud based implementation, while being interesting in some situations is very distant to the approach that we are looking for.

TrOCR is a transformed based OCR. While these architectures are highly effective and achieve great performance, they present significant challenges for constrained hardware platforms. Its high memory use and computational demands resulting in large inference latency make them a hard option. However, since small-size transformers exist, an attempt will be made and a transformer based models will be tested.

EasyOCR, **PaddleOCR** are great options to test in edge devices. Despite their low accuracy with respect to the other approaches, they are based in Deep Learning architectures that Jetson Nano can take profit from in order to accelerate computation. **A custom CRNN** was also implemented to avoid library restrictions and more freedom in optimisation, but most pretrained weights didn't allow recognition in alphanumerical codes that didn't display natural language words. This second approach was discarded.

In summary, three models have been chosen to implement: **EasyOCR** and **PaddleOCR**, which are very promising, and **TrOCR**, which despite its computational requirements will be implemented and tested in order to see if they can actually be executed in Jetson Nano.

3.2 TrOCR preliminar results and exclusion from benchmark

Since the feasibility of implementing a Transformer-based OCR had to be studied, a preliminar execution was done. All raw images from the use case dataset were fed to all three elected models (EasyOCR, PaddleOCR and TrOCR) using an NVIDIA RTX 3090 as execution platform. The average latency per image as well as accuracy metrics were extracted. This led to a representation of each model's behavior, allowing us to know whether it was suitable and logic to implement each in a Jetson Nano given the objectives we were looking for. Figure 4 shows the time per code in milliseconds, the number of correct predictions and the character precision and recall percentages.

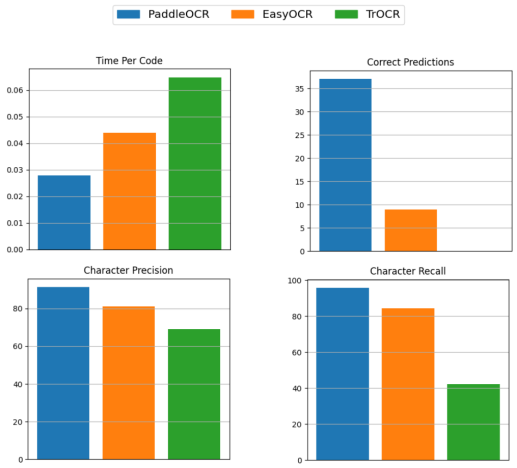


Fig. 4: Comparison after implementing EasyOCR, PaddleOCR and TrOCR in a RTX 3090 GPU.

As Figure 4 shows, EasyOCR and PaddleOCR outperform the transformer OCR with the given dataset formed of alphanumerical codes. Not only inference time is larger, which was already expected given the model complexity, but accuracy metrics are very distant to what would be expected from a Transformer-based model. After some analysis, this can be explained by how the transformer was trained. The transformer model only looks for a single line of text in the image, mixing all predicted characters without differentiating different bounding boxes and generating results that are far from what is needed. TrOCR, given its weak baseline performance without resource constraints is discarded from the benchmark.

4 Development

The development comprehends the system setup, scripting and benchmark generation. Since we are working in an edge device, the setup configuration is a core factor. Some unstable installation or feature may affect the computation efficiency and capabilities of the Jetson Nano.

4.1 System configuration

Once the Jetson Nano has been correctly set up and initialized with JetPack 4.6.6 many dependencies have to be settled. The latest Python version that has available PyTorch with CUDA support is Python 3.6.9. This means that many packages may not work fine with Python or may even not be available. Dealing with this is one of the core tasks when working with a device like the Nano.

In order to accomplish the best results with Jetson devices and specifically with the Jetson Nano it is mandatory to activate the required modes to unleash maximum performance. Different from overclocking, JetPack is able to increase the CPU and GPU clock speeds as well as change the power mode to increase wattage (from 5W to 10W) and therefore performance.

```
# Increase CPU and GPU clock speed
sudo jetson_clocks
# Activate 10W mode in Jetson Nano
sudo nvpmodel -m 0
```

4.2 Mandatory and recommended packages to install

- **OpenCV:** this is the basis for most image operations in practically all computer vision tasks and frameworks that exist in Python. In order to ensure its optimal execution, it is possible to install the package with CUDA support. Although not every method used needs CUDA, having it can increase performance for certain processes. To make this possible, the swap memory has to be increased at least from 2GB to 4GB, since it has only 4GB of RAM memory and installation will require more. Due to the low-speed 4-core CPU the installation will last around 5 hours. Notice that this swap memory increase will be needed for TensorRT engine computation, so reverting it is not recommended unless disk storage is needed. Even increasing

size to 6GB is a good option for allowing maximum security in long processes execution.

- **jtop:** while it can give visually interpretable information about CPU, RAM, SWAP and GPU usage, it also has an information window that let's you check whether dependencies that require GPU usage are active or not as well as relevant OS information. Jtop display can be seen in Figures 5 and 6. Since jtop requires some performance to work and can occupy some resources and even crash, while executing processes it is recommended to use *tegrastats*, which constantly displays all usage information in string format.

- **NANO:** this apk package is essential since it will allow to modify text files and scripts in the Jetson OS.

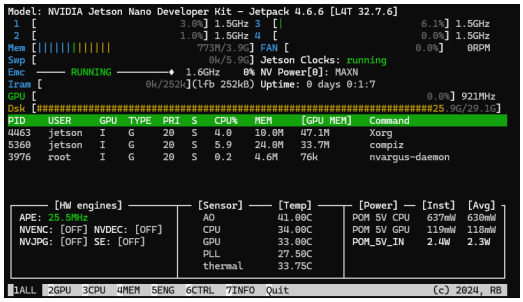


Fig. 5: jtop window displaying resources usage.

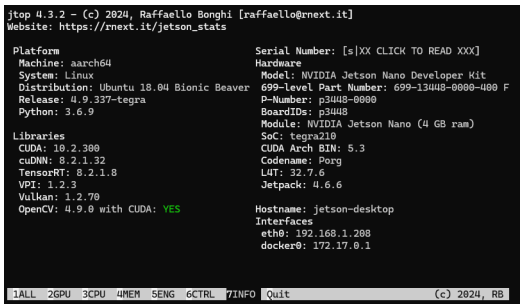


Fig. 6: jtop information window displaying dependencies versions and OS data.

Docker containerization is a possibility and its implementation was attempted. However, some issues raised with the NVIDIA runtime flag that allows the container to use GPU acceleration. Moreover, some important tools like TensorRT were already pre-installed by JetPack and were in the host OS. Containerization forced a copy of the files duplicating disk space. While it was possible, this approach was discarded and different environments were created to work with the required dependencies without incompatibility issues. Moreover, by removing the Docker daemon layer from the execution process less computational charge will be demanded by the OS and more available resources the OCR tasks will have.

4.3 EasyOCR in Jetson Nano

EasyOCR is built with PyTorch, so it requires the library to import it and use it. Up-to-date versions of EasyOCR are only available from Python 3.8 forward, but PyTorch doesn't offer CUDA support for Python $>$ 3.6.9. This is a compatibility problem. While we want a functional EasyOCR version we also need CUDA to be available. If not, the execution in Jetson Nano is senseless. There is a way to install a functional and complete version of EasyOCR with CUDA support in Python 3.6.9.

If EasyOCR is simply pip installed the installation will raise several pip version unstability warnings and it will be killed. A dependency has to be manually installed and a specific version of EasyOCR must be selected. Any other version will kill the installation process and the package will not be installed.

```
pip install "opencv-python-headless<4.6"
pip install "easyocr==1.6.2"
```

Once the installation is completed, the import of EasyOCR will raise errors in Python. A last step is needed. A script from the source code from the installed EasyOCR package has to be modified, since a package is calling a function that doesn't exist in the installed dependencies because of version mismatching. Therefore, the import `from bidi import get_display` will have to be substituted by `from bidi.algorithm import get_display`. Now EasyOCR is completely functional and can be executed with Jetson Nano's GPU processor.

4.3.1 EasyOCR model acceleration

Generally OCR architectures are formed by two different models: the detector and the recognizer. The first detects different blocks of text in an image and the latter predicts each bounding box characters into actual text. The TensorRT acceleration for each module is done differently for each case.

- **The detector** is optimized with torch2trt, which is a module that computes TensorRT acceleration in PyTorch models without having to export the model to ONNX format. This implementation is simpler and more traceable since all the pipeline is defined in a script.

- **The recognizer** is optimized with TensorRT. The module is exported to ONNX format and accelerated with the standard TensorRT installation. This module raised several issues since the forward process was different for inference than for training and different inputs were expected. However this hybrid approach was finally done and in the end both modules were accelerated.

4.4 PaddleOCR in Jetson Nano

PaddleOCR is built over the PaddlePaddle framework [10], which stands for PArallel Distributed Deep LEarning. It is developed by Baidu, which is a Chinese equivalent to Google. It's installation is simpler than EasyOCR's, but there is an important caveat: it is only available for Python $>=$ 3.7.5. This is not a problem, since PaddlePaddle is different from PyTorch and does have CUDA support for using the GPU.

The PaddlePaddle wheels have to be downloaded from their official site and a simple pip install installs the framework with GPU support. Then the *paddleocr* library has to be installed and the PaddleOCR model is fully functional. Notice that since we are working with different Python versions, working with isolated environments will give better consistency among executions and will avoid many problems.

When calling the PaddleOCR object a default model is downloaded, which uses a PP-OCRv3 architecture. They have several models available for downloading in their official site which are meant to be more efficient while keeping the same architecture. Quantization, distillation and other acceleration methods have been applied during their training for optimizing their inference. However, changing the default model by any of these online alternatives always resulted in a drastical drop of performance and latency, so the default model was finally used.

4.4.1 PaddleOCR model acceleration

PaddleOCR works different than EasyOCR. PaddlePaddle has its own implementation of TensorRT embedded to the package, which allows optimisation by simply activating a "True" flag named "use_trt". Nevertheless, there exist different tweakable parameters that can affect this acceleration like quantization values. Exporting the models into ONNX format is rather complex since it doesn't have direct support from PaddlePaddle and documentation is written in Chinese, difficulting the understanding of the framework.

5 RESULTS

In order to ensure optimal execution, a random image from the whole dataset (outside the 100 set) has been inferred 30 consecutive times before starting the inference as warm up. Different metrics will be used to benchmark the results. A set of metrics will evaluate the prediction performance of the model and another set of metrics, which are simpler, will evaluate the efficiency in computation.

5.1 Accuracy metrics

Since we are working with alphanumerical codes, different types of accuracies should be used, allowing flexibility and a better representation and understanding on how the model is actually performing over the used dataset.

5.1.1 Strict Accuracy

As the name states, it is the most restrictive metric. It gives a binary outcome for each prediction, being 1 if the whole prediction string exactly matches the ground truth and 0 if it doesn't.

$$StrictAccuracy = \frac{1}{M}\sum_{i=1}^{M} 1(y_i = \hat{y}_i)$$

$M =$ *number of samples*; $y_i =$ *ground truth for sample* i;

$\hat{y}_i =$ *prediction for sample* i; $1 =$ *indicator: 1 if true, 0 else*

5.1.2 Character Error Rate (CER)

This metric measures how different the predicted string is from the ground truth in terms of the minimum number of character insertions, deletions and substitutions that are needed to transform one into the other. It is the Levenshtein distance divided by the length of the ground truth and, as it, the bigger the value the more different the prediction is.

$$CER = \frac{S + D + I}{N}$$

$S =$ *number of substitutions*; $D =$ *number of deletions*;

$I =$ *number of insertions*;

$N =$ *number of characters in ground truth*

5.1.3 Character-Level

These are computed as generic precision and recall metrics but treating each character as an item in a set. Precision computes how many of predicted characters were correct and recall computes how many of ground truth characters were correctly predicted.

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

5.1.4 Latency metrics

Regarding computation efficiency, two main metrics will be given: the average time needed for the OCR model to predict a code in an image and the frames per second (FPS) the model can achieve.

5.2 Baseline results

Both models have been executed in all platforms and in several conditions. Results are given with inference in CPU and GPU for the baseline execution. This way, the evolution in performance and latency can be better followed. In case of the Jetson Nano, the 10 watt mode was always activated along with the Jetson Clocks flag set as "running".

5.2.1 CPU results

On CPU, PaddleOCR consistently outperforms EasyOCR in both accuracy (lower CER, higher precision/recall) and speed, especially on the server. The Jetson Nano shows a significant drop in throughput for both models, but PaddleOCR remains more efficient and accurate. See results in Table 3.

Metric	EasyOCR		PaddleOCR	
	Server	Nano	Server	Nano
Strict Acc.	0.00%	0.00%	0.00%	0.00%
CER	20.54%	48.06%	18.19%	18.19%
Precision	84.32%	67.80%	85.24%	85.79%
Recall	94.55%	90.18%	94.24%	94.07%
Preprocess	1 ms	20 ms	4 ms	32 ms
OCR	238ms	10155 ms	106 ms	3510 ms
Total	239ms	10246 ms	111 ms	3542 ms
Throughput	4.2 FPS	0.01 FPS	9.01 FPS	0.28 FPS

TABLE 3: CPU Performance: EasyOCR vs PaddleOCR

5.2.2 GPU results

GPU acceleration greatly improves inference speed for both models, especially on the server. PaddleOCR achieves the highest throughput and lowest CER on the server, but its performance drops on the Nano GPU, particularly in recall and CER. See results in Table 4.

Metric	EasyOCR		PaddleOCR	
	Server	Nano	Server	Nano
Strict Acc.	0.00%	0.00%	0.00%	0.00%
CER	20.70%	20.73%	18.19%	39.83%
Precision	84.26%	84.31%	85.24%	76.47%
Recall	94.55%	94.58%	94.24%	71.90%
Preprocess	1 ms	17 ms	4 ms	29 ms
OCR	45 ms	749 ms	33 ms	4799 ms
Total	46 ms	766 ms	37 ms	4828 ms
Throughput	21.74 FPS	1.31 FPS	27.03 FPS	0.21 FPS

TABLE 4: GPU Performance: EasyOCR vs PaddleOCR

5.3 Accelerated results and comparison

The main aspect that allows noticeable difference in latency is quantization. This method transforms the model's parameters to lower-precision representations from 32-bit floatinpoint to 16 or even 8-bit, reducing computational load and enabling faster inference.

Both models have been executed under the same conditions but giving the different TensorRT implementations as described in Section 4. Server's GPU inference has been computed in FP16 as reference. In case of the Jetson Nano, the 10 watt mode was always activated along with the Jetson Clocks flag set as "running". All results are visible in Tables 5 and 6. INT8 quantization was attempted, but the TensorRT engine couldn't be generated because of resource constraints. The process was always killed due to lack of enough swap memory.

5.3.1 EasyOCR

TensorRT acceleration significantly reduces inference time for EasyOCR, especially in FP16 mode on the Nano, increasing throughput more than threefold compared to FP32, with only a minimal impact on accuracy metrics. While without any optimization the model latency was of 488ms, the accelerated version reduced time inference to 173ms, resulting in a speedup of 64.5%. See results in Table 5.

5.3.2 PaddleOCR

PaddleOCR benefits greatly from TensorRT acceleration on the server, achieving the highest throughput overall. On the Nano, FP32 mode yields the best accuracy and lowest CER, while FP16 mode causes a dramatic drop in prediction quality, despite improved speed. See results in Table 6.

5.4 Results discussion

The results obtained show that in the best configuration (PaddleOCR with TensorRT optimization in FP32 on the Jetson Nano), a character-level precision of around 90% and recall of 96% were achieved, with a Character Error Rate (CER) of approximately 13.8%. However, strict accuracy remained consistently at 0% across all experiments and configurations. This means that, while the models are able to recognize most individual characters, they fail to achieve exact string-level transcription in any of the 100 benchmark images.

An example is the following code:
`PRO:08/2022EXP:08/2023L2215029520H02:05`,
which was predicted as
`R00872022FXP:08/20232213029520H02'05`

While the strings share some similarity, they are very different and the prediction does not reflect the real code. Output handling cannot help either to fix some misprediction errors that could happen betweeen an L and number 1, for example. Errors come from very different characters.

Regarding computational efficiency and an acceptable accuracy, the best end-to-end latency comes from EasyOCR with FP16 quantization, achieving an inference average of 173 ms, which translates to a throughput of 7.45 FPS. For an industrial environment where a high volume of products must be processed (e.g., 10–30 items per second), this speed is far below the real-time threshold (ideally 30 FPS or more). It has been seen that care must be taken with low-precision (FP16) modes.

5.4.1 Analysis of Limitations and Causes

The unsatisfactory results may not be solely due to the hardware limitations of the Jetson Nano, but rather to a combination of interrelated factors:

- **Dataset limitations and domain mismatch:** The benchmark dataset consists of only 100 images of alphanumeric codes, which are short, dense, and sometimes present low contrast or minor occlusions. EasyOCR and PaddleOCR, while robust, were predominantly trained on natural language text (longer words, high contrast). This domain difference means the models are not optimized for the specific typology of product codes. The lack of fine-tuning or domain adaptation causes the recognition head to confuse visually similar characters (e.g., 'O' vs '0', '1' vs 'I', 'B' vs '8', 'S' vs '5'), resulting in a relatively high CER and, crucially, zero strict accuracy.

- **Impact of preprocessing:** Although some preprocessing techniques (such as OTSU thresholding) were explored, preliminary results showed that using grayscale images without further processing was slightly superior. The variability in image quality (contrast, lighting, noise) requires more adaptive methods (like CLAHE or edge enhancement filters), which were not implemented, leaving many problematic images unoptimized for model input.

- **Model design and optimization:** The exclusion of TrOCR was justified by its poor initial performance and inability to differentiate multiple bounding boxes, highlighting the importance of model architecture for the use case. For PaddleOCR, it was observed that enabling FP16 mode on the Jetson Nano caused a

		Baseline models (PyTorch)				Accelerated models (TensorRT)		
		Server CPU	Jetson Nano CPU	Server GPU	Jetson Nano GPU	Server GPU	Jetson Nano GPU (FP32)	Jetson Nano GPU (FP16)
EasyOCR	**Strict accuracy**	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**
	Character Error Rate (CER)	**20.54%**	**48.06%**	**20.70%**	**20.73%**	**21.75%**	**20.25%**	**20.29%**
	Character-Level Precision	**84.32%**	**67.80%**	**84.26%**	**84.31%**	**83.63%**	**82.67%**	**82.61%**
	Character-Level Recall	**94.55%**	**90.18%**	**94.55%**	**94.58%**	**94.48%**	**86.52%**	**86.39%**
	Time per preprocess (ms)	**1ms**	**20ms**	**1ms**	**17ms**	**1ms**	**17ms**	**18ms**
	Time per ocr (ms)	**238ms**	**10155ms**	**45ms**	**749ms**	**39ms**	**476ms**	**155ms**
	Time per sample (ms)	**238ms**	**10246ms**	**46ms**	**766ms**	**40ms**	**488ms**	**173ms**
	Throughput	**4,2 FPS**	**0,01 FPS**	**21,74 FPS**	**1,31 FPS**	**25 FPS**	**2,03 FPS**	**7,45 FPS**

TABLE 5: Table benchmarking EasyOCR performance metrics.

		Baseline models (PyTorch)				Accelerated models (TensorRT)		
		Server CPU	Jetson Nano CPU	Server GPU	Jetson Nano GPU	Server GPU	Jetson Nano GPU (FP32)	Jetson Nano GPU (FP16)
PaddleOCR	**Strict accuracy**	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**0.00%**	**7.00%**	**0.00%**
	Character Error Rate (CER)	**18.19%**	**18.19%**	**18.19%**	**39.83%**	**18.16%**	**13.80%**	**81.54%**
	Character-Level Precision	**85.24%**	**85.24%**	**85.24%**	**76.47%**	**85.26%**	**90.30%**	**30.80%**
	Character-Level Recall	**94.24%**	**94.24%**	**94.24%**	**71.90%**	**94.27%**	**95.74**	**20.91%**
	Time per preprocess (ms)	**4ms**	**32ms**	**4ms**	**29ms**	**5ms**	**30ms**	**29ms**
	Time per ocr (ms)	**106ms**	**3510ms**	**33ms**	**4799ms**	**13ms**	**485ms**	**113ms**
	Time per sample (ms)	**111ms**	**3542ms**	**37ms**	**4828ms**	**17ms**	**515ms**	**142ms**
	Throughput	**9,01 FPS**	**0,28 FPS**	**27,03 FPS**	**0,21 FPS**	**58,82 FPS**	**1,94 FPS**	**7,03 FPS**

TABLE 6: Table benchmarking PaddleOCR performance metrics.

drastic degradation in prediction quality, despite improved speed. This indicates that low-precision quantization is not trivial and may require specific training or quantization-aware fine-tuning to maintain accuracy. The TensorRT implementation for EasyOCR, while reducing inference time, did not achieve the expected improvements compared to benchmarks on more powerful devices like the Jetson Xavier NX, which are presented further in this section. This suggests that maybe the TensorRT version in JetPack 4.6.6 does not optimize certain layers as efficiently.

- **Jetson Nano hardware limitations:** While the Jetson Nano is a powerful platform for edge computing, its 128 CUDA cores, 4GB RAM, and 5–10W power consumption impose inherent limits. Swap memory pressure during TensorRT engine building and inference is a limiting factor.

In summary, although a significant reduction in inference times has been achieved compared to CPU execution, the current models are not ready for applications that demand perfect code transcription. While inference time optimization is important, improving full-string recognition accuracy is the most critical factor and the main bottleneck for the real-world applicability of the system.

5.4.2 Implications for the Industrial Use Case

The combination of zero strict accuracy and low throughput makes the current system unviable for industrial applications that require:

- **Real-time processing:** The current latency would immediately create a bottleneck in a production line.
- **High reliability:** 0% strict accuracy means every code read would require manual verification, defeating the purpose of automation.
- **Robustness:** The degradation of accuracy in low-precision modes (FP16) and inconsistency in reading certain characters demonstrate a lack of robustness for variable industrial environments.

We have seen that based on the accuracy metrics and the latency of the models, results are rather disappointing. While a big reduction in inference time has been accomplished, it is not enough given the purpose of the application. The code prediction results aren't good either, making this model not usable for real-world application in case close to perfect precision is needed. It must be studied whether these results are justified by the nature of the Jetson Nano, which cannot achieve a better performance, or if it is a matter of a wrong implementation or wrong dependencies installation.

5.4.3 Contextualisation on Jetson Nano performance

A single demonstration regarding PaddleOCR performance can be found online in a blog called *Setting Up PaddleOCR with CUDA on NVIDIA Jetson from Scratch* [11]. This demonstration explains how to install PaddleOCR in Jetson Nano and how to generate predictions. While it doesn't show any accuracy metrics at all, it gives a speed comparison between CPU, GPU and GPU + TensorRT execution. The results can be seen in Table 7.

Configuration	Time
CPU	$\sim$0.5s
GPU	$\sim$0.65s
GPU + TensorRT	$\sim$0.03s

TABLE 7: Inference Time by online blog.

These results are drastically better than ours. While, at best, we accomplish a speed of $\sim$500 milliseconds with relatively good performance, the user that published the blog without further fine-tuning accomplished 30 milliseconds. Since the results are not supported by any detail rather than the execution time, some benchmark comparison has to be done in order to situate if our Jetson Nano is working as it should or if it is underperforming due to some wrong dependency installation.

In order to set an overall baseline performance on our Jetson Nano, we will compute existing benchmarks on the same device and compare the results. This way, we will find out about the expected computing capabilities of the

computing platform and understand if the results can be explained by the Jetson Nano processing capacity. We can see the difference between a reference Jetson Nano and the one used for this project in Figure 7. As reference, the benchmark that is being taken into account is from the official GitHub NVIDIA-AI-IOT/torch2trt repository [9].

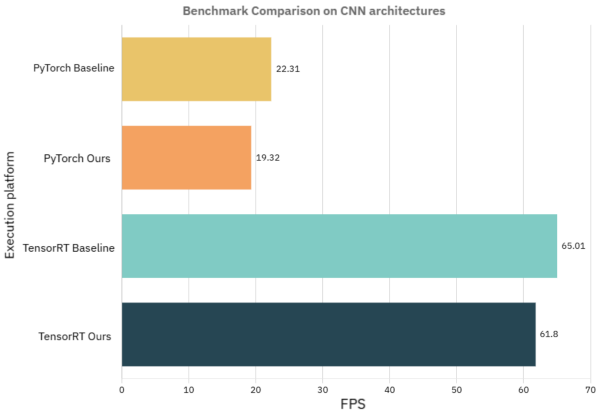


Fig. 7: Average throughput for several models (alexnet, squeezenets, resnets, densenets) executed both in PyTorch and TensorRT in a reference Jetson Nano and Jetson Nano used for this paper.

The difference in performance from the reference Jetson Nano is of 3 FPS, being our Jetson Nano slightly slower. This difference shows that the Jetson Nano is performing as it should, replicating very closely other devices' performance.

5.4.4 Comparison with Jetson Xavier NX

While practically no reference to OCR tasks in the standard Jetson Nano can be found online, you can find several references to the use of the Jetson Xavier, which is a much more capable Jetson device that drastically outperforms our device. It is relevant to provide this information since it is an alternative to the latter. However, the price of the processor is 6 times higher than the Jetson Nano price, which is remarkable. The difference in both devices specifications can be seen in Table 8. The difference in performance benchmark on CNN architecures can be seen in Figure 8, proving that, in average, the Jetson Xavier is 19 times faster than the Jetson Nano. As reference, the Jetson Xavier benchmark data has been obtained from the NVIDIA-AI-IOT/jetson_benchmarks GitHub repository [12].

Specification	Nano	Xavier NX
TFLOPS	∼0.5	∼6
CUDA Cores	128	384
VRAM	4GB	8GB
Power	5–10W	10–20W

TABLE 8: Jetson Xavier NX vs Jetson Nano specifications.

By knowing the approximate difference between both models we can try to understand if any issue can be happening in terms of latency. We will take another existing benchmark in the Jetson Xavier, the NVIDIA-AI-IOT/scene-text-recognition GitHub repository [13], to check how different each EasyOCR module behaves.

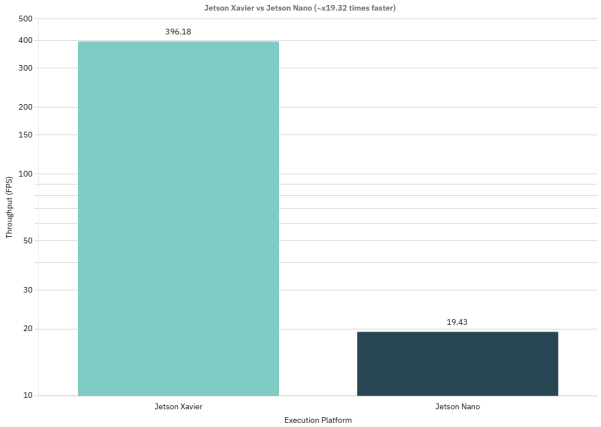


Fig. 8: Logarithmic bar plot representing the average throughput difference between the Jetson Xavier and the Jetson Nano (using inception, vgg19, u-net with segmentation, yolov3, resnet50 and mobilenet)

Model	Throughput (FPS)	Latency (ms)
Detection	12,386	84.900
Detection TRT	24,737	48.990
Recognition	174,518	5.900
Recognition TRT	7,118,642	0.160

TABLE 9: EasyOCR modules performance metrics on Jetson Xavier NX.

The results demonstrate how extremely fast the inference is in the Jetson Xavier with TensorRT acceleration. The whole pipeline lasts ∼50 milliseconds in total.

Model	Throughput (fps)	Latency (ms)
Detection	4,370	217.540
Detection TRT	10,520	95.020
Recognition	8,270	107.330
Recognition TRT	12,880	77.650

TABLE 10: EasyOCR modules performance metrics on Jetson Nano.

If we compute the difference for each step we will be able to see how different the two devices behave. We will take as reference the x19 inference time obtained by the data in Figure 7. Detection is 2,56 times faster in Xavier and TensorRT detection is 1,94 times faster in Xavier. The variance is very small. However, in recognition the differences vary. Recognition without TensorRT acceleration is 18,19 times faster in Xavier, which is very close to what was extracted from the previous benchmark. However, recognition with TensorRT acceleration is 485,31 times faster in Xavier. This is an exaggerated value that may indicate that some issue may be affecting TensorRT performance in the recognition module inference.

6 Further approaches

Based on the analysis of the results and the identified limitations, several actions are proposed to address the main issues encountered during the project and to realign the objectives towards a more robust and effective OCR solution on edge devices.

- **Dataset expansion and domain adaptation:** One of the most impactful improvements would be to expand the labeled dataset, leveraging the full set of available images (over 1,700) and, if possible, generating synthetic data that mimics the real codes' appearance, fonts, and noise patterns. Fine-tuning the OCR models (especially the recognizer component) on this domain-specific data is expected to significantly reduce the Character Error Rate and improve strict accuracy, as the models would better learn the structure and variability of the target codes.

- **Advanced preprocessing techniques:** Implementing adaptive preprocessing methods, such as CLAHE (Contrast Limited Adaptive Histogram Equalization), local thresholding, or edge-preserving filters, could help normalize image quality and enhance code visibility, especially in low-contrast or noisy samples. A preprocessing pipeline that dynamically selects the best enhancement method per image or region could further improve recognition robustness.

- **Model optimization and lightweight architectures:** Exploring lighter OCR architectures, such as PP-OCRv3-tiny or custom pruned/quantized models, may help reduce inference time and memory usage without sacrificing accuracy. Additionally, quantization-aware training or post-training quantization (including INT8) could be tested, although the Jetson Nano's hardware limitations must be considered.

- **Error correction and post-processing:** Integrating post-processing steps, such as regular expression validation can help correct common recognition errors and increase the effective strict accuracy.

- **Pipeline and parallelization improvements:** Optimizing the data pipeline by introducing multi-threading or asynchronous processing (e.g., separating image capture, preprocessing, and inference stages) could help hide preprocessing latency and increase overall throughput. Batching multiple regions of interest per inference pass may also improve GPU utilization.

- **Hardware considerations:** If real-time performance and high reliability are strict requirements, migrating to a more powerful edge device such as the Jetson Xavier NX or Orin should be considered. These platforms offer significantly higher computational resources and better support for advanced model optimizations, making them more suitable for demanding industrial applications.

- **Reproducibility and portability:** To facilitate future experimentation and deployment, creating Dockerfiles or containerized environments for the entire OCR pipeline would improve reproducibility and ease of installation, especially when dealing with complex dependencies and version mismatches.

In summary, addressing the current limitations requires a combination of data-centric improvements, model adaptation, smarter preprocessing, and, if necessary, hardware upgrades. Prioritizing domain fine-tuning and solid post-processing is likely to lead to the most immediate gains in accuracy and reliability.

7 Conclusions

This project explored the use of deep learning OCR on edge devices like the Jetson Nano for industrial applications. While techniques such as TensorRT improved inference speed, the system still does not meet the speed or accuracy required for real-time industrial production.

PaddleOCR proved to be the most balanced option, but even in its optimized form, it did not meet expectations, and its accuracy was too low for tasks that require perfect code reading. Quantization and pruning were also not practical due to memory and compatibility limitations on the Jetson Nano.

Despite these challenges, this work provides a solid starting point for future OCR optimization on edge devices. The Jetson Nano is suitable for prototypes or less demanding applications, but not for large-scale industrial deployments with current models.

For future improvements, we recommend expanding and adapting the dataset, fine-tuning the models, and using advanced preprocessing and post-processing techniques. For applications that require high real-time performance and reliability, more powerful hardware such as the Jetson Xavier NX or Orin should be considered.

In summary, this project highlights the current challenges and opportunities of embedded AI in industry, offering valuable insights for both researchers and the industrial sector.

References

[1] NVIDIA Corporation, "Nvidia jetpack sdk," https://developer.nvidia.com/embedded/jetpack, 2025, version 6.2.1, accessed June 30, 2025.

[2] F. Oh, "What is cuda?" https://blogs.nvidia.com/blog/what-is-cuda-2/, 2012, accessed June 30, 2025.

[3] NVIDIA Corporation, "Nvidia cudnn: Cuda deep neural network library," https://developer.nvidia.com/cudnn, 2025, accessed June 30, 2025.

[4] ——, "Nvidia tensorrt documentation," https://docs.nvidia.com/tensorrt/index.html, 2025, accessed June 30, 2025.

[5] winter2897, "Real-time auto license plate recognition with jetson nano," https://github.com/winter2897/Real-time-Auto-License-Plate-Recognition-with-Jetson-Nano, 2025, gitHub repository, accessed June 30, 2025.

[6] G. Sankar, "Why ar2020-based cameras are perfect for ocr enablement in warehouse automation," https://www.e-consystems.com/blog/camera/applications/why-ar2020-based-cameras-are-perfect-for-ocr-enablement-in-warehouse-automation/, 2024, accessed June 30, 2025.

[7] Advanced Integration, “Smartcow – smartcam t1023,” https://www.advanced-integration.ae/smartcow-smartcam-t1023/, 2025, accessed June 30, 2025.

[8] SmartCow AI Technologies Ltd., “Smartcam t1023 datasheet, version 0.1,” https://assets-global.website-files.com/62fcf08c522a551f5456cb42/65fab6608d6edc84284d8fd0_SmartCam-T1023_Datasheet_V0.1_240219.pdf, 2024, accessed June 30, 2025.

[9] NVIDIA AI IOT, “torch2trt: An easy to use pytorch to tensorrt converter,” https://github.com/NVIDIA-AI-IOT/torch2trt, 2025, gitHub repository, accessed June 30, 2025.

[10] Roboflow, “What is paddlepaddle?” https://blog.roboflow.com/what-is-paddlepaddle/, 2023, accessed June 30, 2025.

[11] Zichun, “Setting up paddleocr with cuda on nvidia jetson from scratch,” https://linzichun.com/posts/setup-paddleocr-cuda-tensorrt-jetson-nano/, 2023, accessed June 30, 2025.

[12] NVIDIA AI IOT, “jetson_benchmarks: Benchmarks targeted for jetson (using gpu+2dla),” https://github.com/NVIDIA-AI-IOT/jetson_benchmarks, 2025, gitHub repository, accessed June 30, 2025.

[13] ——, “scene-text-recognition,” https://github.com/NVIDIA-AI-IOT/scene-text-recognition, 2025, gitHub repository, accessed June 30, 2025.