NVIDIA Jetson Nano for machine control via Gesture Recognition

Software.
Embedded.

Customer: NA
Project name: JetsonforMCviaGR
Version: 1.0

CONTENT

Page

1 Overview 3

1.1 System architecture 3
1.2 Involved partners 4

2 Implementation of Gesture Recognition 4

2.1 NVIDIA Gesture Recognition reference app 4
2.1.1 Model training 5
2.1.2 Evaluation 7
2.1.3 Deployment 8
2.1.4 Inference 9
2.2 MIT Temporal Shift Module 9
2.2.1 Implementation 9
2.2.2 Deployment 11
2.2.3 Inference 11

3 Real-time performance 11

3.1 Implementation 11
3.2 Real-time tests 12

4 Glossary 17

1 Overview

The NVIDIA Jetson series provides GPU computing capabilities on hardware with a small form factor, which makes the use of AI models in Edge applications more practical. The project described in this document had three goals. The hardware for this project was a Jetson Nano, which is the least powerful platform from the Jetson family:

- Implementing a real-time patch for Linux and the CODESYS software to turn the Jetson Nano into a high-end PLC with EtherCAT and OPC UA communication
- Combining the capability of AI with automation software to develop a system that could be controlled via Gesture Recognition
- run AI workloads whilst also meeting real-time requirements.

1.1 System architecture

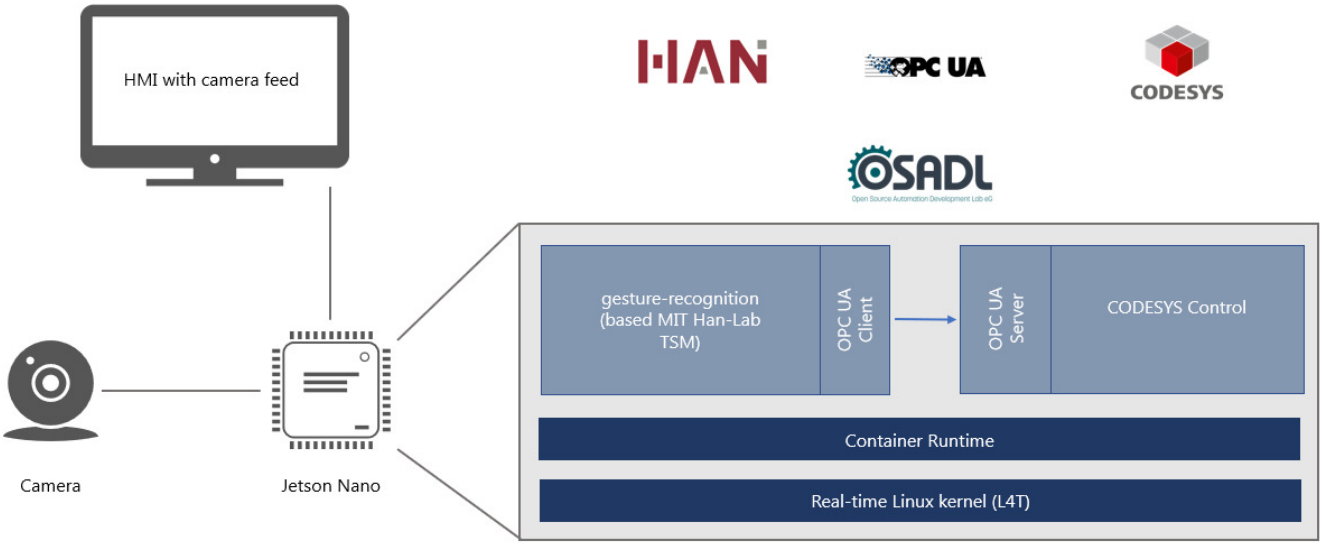

Figure 1: System architecture

Jetson nano is a PLC. It has a camera connected via USB and a display to stream video from the camera and display the detected gesture.

The Jetson nano firmware is based on the real-time Linux kernel with docker runtime. CODESYSControl and gesture-recognition run in 2 separate containers.

The interface to the control application is realized via OPC UA. The control application, in our case, is a CODESYS application with an OPC UA server exposing variables to an OPC UA client. There is no specific adaptation to be done in CODESYS Control to use OPC UA server. Variables to be exposed are simply selected in the symbolic configuration part of the IEC application (CODESYS application).

As the ML program is written in Python, the open-source Python client from FreeOpcUa is used. The reactions by the machine to specific hand gestures are in turn programmed into the control application. To configure the client, a text file is parsed that contains the configurations of the server, such as IP address etc.

1.2 Involved partners

BE.services GmbH

- Model selection
- Model training
- Model implementation
- OPC UA interface to control application
- Real-time patch
- Real-time performance measurement

University of Applied Sciences Kempten

- Provision of machine demonstrator
- Adjustment of control application

2 Implementation of Gesture Recognition

Selecting an appropriate model for the project is crucial considering the constraints in computing power as well as the desired usability of the live gesture recognition. The general workflow for any ML/AI project is described in Figure 1, however, depending on whether pre-labelled data or pre-trained models are available, it is not always required to start at the first step. With pre-trained models the process can be started in the *Model training* phase, or even with *Deployment* depending on how similar the current objective is compared to the one the model was originally trained to solve. Additionally, if the development/training platform differs greatly from the eventual deployment platform, the performance during the inference phase must be assessed, for instance in terms of usability.

Figure 1 ML/AI project workflow

After researching pre-trained models for gesture recognition, the selection was narrowed down to two alternatives, which are explained in the following sections.

2.1 NVIDIA Gesture Recognition reference app

NVIDIA provides a pre-trained model for gesture classification on their NGC platform. This model can distinguish six different gestures and is intended to be used with NVIDIA Deepstream SDK. Along with this model there is sample repository and webinar from NVIDIA that utilizes the Transfer Learning Toolkit (TLT) and Deepstream to develop a two-stage app that detects a hand, crops the respective region, and performs classification with the pre-trained model. As the classification model is already trained, it can be deployed directly. However, the hand detection model must be trained, hence the pipeline is entered at the *Model Training* step:

Figure 2 Entry stage for NVIDIA Gesture Recognition

2.1.1 Model training

The first step is training the hand detection model. For this, NVIDIA's Transfer Learning Toolkit (TLT) was used as described in the published sample app [1]. TLT is a toolkit that enables the training, fine-tuning, pruning, and exporting models by providing a no-code approach to transfer learning for multiple neural network architectures. Training was performed with one NVIDIA RTX 2070 Super desktop GPU.

The dataset used for training was the Egohands dataset that contains 4800 annotated images of hands, whose labels were transformed into the KITTI format with the help of a script by JK Jung [2]. The resulting labels have the following form and include the category and the left, top, right, and bottom location of the bounding box:

```
hand 0 0 0 653 423 825 527 0 0 0 0 0 0 0
```

The dataset is split in a train/validation subset and a test subset, in a ratio of 9:1 containing 4320 and 480 images, respectively. The transfer learning with TLT is performed by first choosing a pre-trained neural network that is available from NVIDIA and then adjusting the configuration file to customize the training process. For instance, training can be launched via this command:

```
tlt detectnet_v2 train -e $SPECS_DIR/egohands_train_resnet18_kitti.txt \
                       -r $USER_EXPERIMENT_DIR/experiment_dir_unpruned_v3 \
                       -k $KEY \
                       -n resnet18_detector \
                       --gpus $NUM_GPUS
```

The configuration file in turn contains the paths for the model, the directory containing the training data etc. In addition, image augmentations, the optimizer and its settings and learning rate scheduling can be set in the configuration file. An excerpt of the config file describing the training specifications looks like this:

```
training_config {
  batch_size_per_gpu: 16
  num_epochs: 120
  learning_rate {
    soft_start_annealing_schedule {
      min_learning_rate: 5e-6
      max_learning_rate: 5e-4
      soft_start: 0.1
      annealing: 0.7
    }
  } …
}
```

Metric	Pre-pruned	Pruned and quantized
mAP [%]	89.5973	90.1111
Size [MB]	45	5

Table 1: Comparison after pruning

As can be seen, also learning rate scheduling can be configured. In total, 120 training epochs were performed. After training, also pruning and post-training quantization were performed. Quantization aware training (QAT) was not performed, as the quantization was not to INT8. The pruning reduced the model size from 45MB to 5MB, and after the process a very slight increase in mAP was noted, as can be seen in Table 1. The last step is to export the model with FP16 precision into an .etlt file, from which a TensorRT engine file can be created that is then deployed and used for inference. Before export, it can be useful to visualize the predictions the hand detector makes. For this, inference was performed on the test set and the predicted bounding boxes are drawn onto the image.

As the sample project provided by NVIDIA is designed for deployment on Jetson Xavier AGX and NX, several adjustments were made to accommodate for the Nano's different hardware.

The used neural network architecture chosen as the backbone was ResNet18 instead of the ResNet34, due to its smaller size but similar performance. Quantization to FP16 was used instead of INT8, as the Nano's Maxwell GPU does not support the INT8 inference. This is currently only supported on Jetson Xavier devices, which possess the NVIDIA DLA (deep learning accelerator). According to the Jetson Nano benchmarks, the Resnet-18 backbone does have lower throughput than the Mobilenet_V2 backbone, which was initially tried. However, in TLT 3.0 using the Mobilenet_V2 backbone for training yields an error that could not be resolved and that is supposed to be fixed in an upcoming release. Similarly, also Resnet-10 was evaluated, with the same training specification. Unfortunately, the mAP for Resnet-10 only reached around 77%, which is significantly worse than the Resnet-18. Consequently, the latter was ultimately chosen as the model for deployment.

Next to the hand detection model, also the gesture classification model must be prepared. This model is available pre-trained by NVIDIA as a .etlt file. For creating TensorRT engine files, the **tlt-converter** tool was used. Usage is also from the command line with the required flags, most importantly the key that was used to export the model to .etlt:

```
./tlt-converter -k nvidia_tlt \
    -t fp16 \
    -p input_1,1x3x160x160,1x3x160x160,2x3x160x160 \
    -e $EXPORT_PATH/model.plan \
    $MODEL_PATH/model.etlt
```

The engine files for both models were created with FP16 arithmetic precision, so no INT8 calibration files are required. TensorRT engine files (.plan or .engine typically) can be understood as standalone binaries that contain the optimized neural network and which can be loaded into an application. The conversion into a TensorRT requires that tensor operations within the network are supported by the TensorRT optimizer, however, all models available for transfer learning with NVIDIA's TLT are fully supported.

2.1.2 Evaluation

Before export, it can be useful to visualize the predictions the hand detector makes. For this, inference was performed on the test set and the predicted detections are drawn onto the image. As can already be seen by the mean average precision of the model of around 90%, detection is not ideal but reasonably good. As a result, in most images the hands are detected well, see Figure 3.

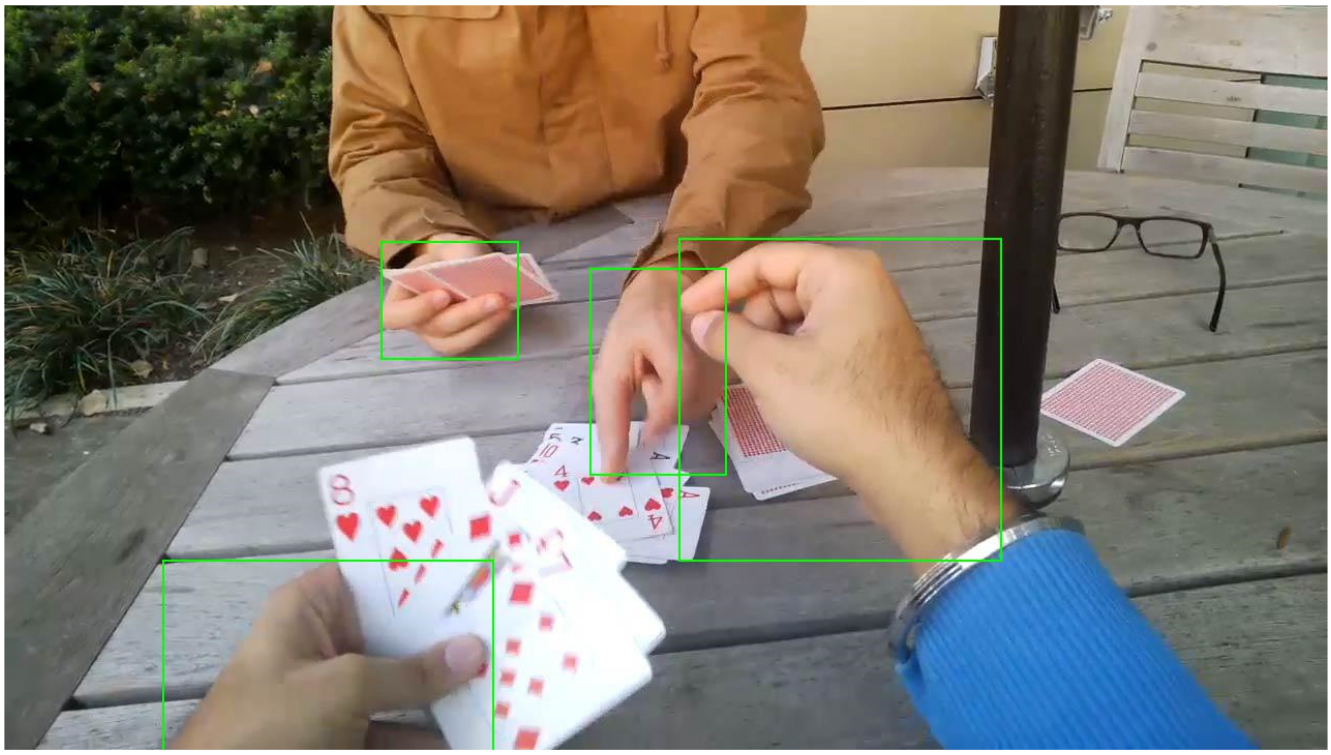

Figure 3: Test set image well detected

However, the model also tends to make false positive predictions for hands where there are none.

By contrast, false negatives were not observed, meaning that if a hand is contained in the image it was detected for the images in the test set. This is not ideal in the context of using it for machine control: falsely detecting hands in the image and performing classification on them can lead to unwanted reactions from the machines. As a countermeasure the confidence threshold for the following classification should be set high. In comparison, the opposite case of not detecting hands in the image is rather insignificant, as this is rare and less impactful on a livestream of a webcam.

2.1.3 Deployment

The deployment to the Nano was done by integrating the two models into a Deeptstream app. Deepstream is a closed-source SDK from NVIDIA for building Intelligent Video Analytics on NVIDIA products. It can be used do build end-to-end vision applications that can include several inference steps, as well as interact with cloud services. These apps are compatible for a wide range of NVIDIA devices, so it is possible to deploy on edge, workstations and cloud making them scalable. The pipeline of the Deepstream app for gesture recognition is shown in Figure 4.

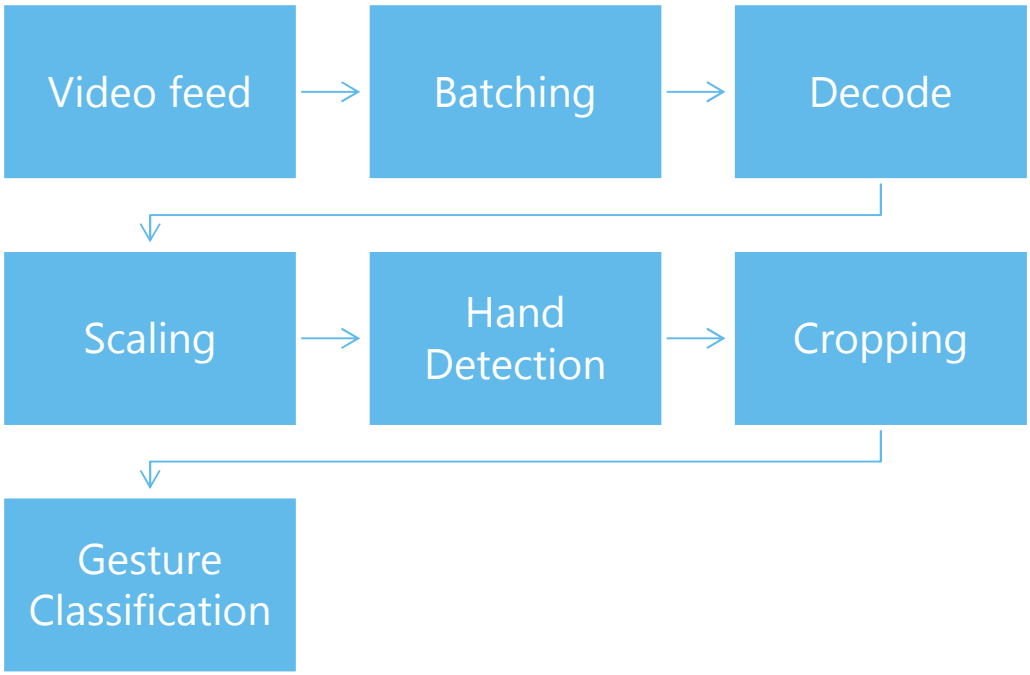

Figure 4: Deepstream pipeline

Since version 5.0, integration of vision models into a Deepstream app can be done by using either TensorRT or Triton Inference Server, the former only supports models compiled to TensorRT engine files, the latter also supports a wider range of model formats.

The provided Deepstream app from the repository was used as the basis, since it contains modifications, such as extending the bounding boxes around the detected hands. This was implemented because the classifier was trained on images with a larger margin around the hands and the unaltered boxes fit the hand too closely. The configuration files were customized to use both models according to their arithmetic precision.

In our case both models were compiled to TensorRT engine files and included into Deepstream via TensorRT. When using Deepstream, the pipeline from source to sink is customized through a configuration file, which includes another configuration file for each inference step that is included.

Deepstream is based on G-streamer and several reference apps are provided by NVIDIA, which demonstrate how to alter parts of the G-streamer pipeline, either to include inferencing steps or for handling output, for instance passing it to some cloud instance. As a result, the implementation of a OPC UA interface would have been included into the G-streamer pipeline as a plug-in. This step was not implemented as the usability of the application was tested before making this effort.

2.1.4 Inference

The working Deepstream app was tested for its usability on the Jetson Nano. The frame rate on the Jetson Nano was around ten fps. Hand detection and tracking with the initial configuration was not always accurate and experiments with different confidence thresholds, batch sizes and clustering settings were not successful in improving performance and usability. The large input size of the frames could be considered a limiting factor, however, reducing the image size cannot be done without changing the size of the training data images.

Consequently, another approach was evaluated: a MobileNetV2 based solution developed by researchers at MIT.

2.2 MIT Temporal Shift Module

Researchers from MIT developed a method of allowing temporal modelling to be performed by regular 2D Convolutional Neural Nets (2D-CNN) by using a fraction of activations from neighbouring frames [3]. This is used to understand motions in videos and can thus be used to recognize gestures. They implemented their model in Pytorch and developed a demo on NVIDIA Jetson Nano with good performance on live video. This project was also NVIDIA Jetson community project of the month in November 2019.

Since the model is already pre-trained for the same purpose as in our application, the ML project pipeline, given in Figure 5 is entered in the *Deployment* phase. Before description of the deployment, the implementation is briefly described.

Figure 5: Entry point for second approach

2.2.1 Implementation

The model itself uses a MobileNetV2 as a backbone and models temporal relationship between frames by replacing a fraction of the current activations with activations from the past frame. The model design is given in Figure 6.

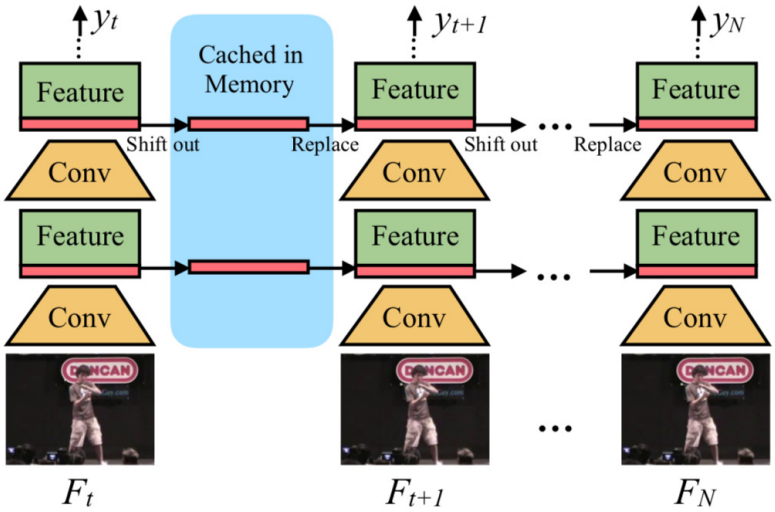

Figure 6: Model design by Lin et al.

By caching a fraction of activations from the previous frames and injecting it when processing the subsequent frame, information about the temporal relationship between frames is passed. Thus, 2D-CNN models, such as MobileNetV2, can be altered to classify videos instead of singular images only. Detailed information is given in the original paper [3].

The model is available pre-trained by the authors on the 20BN-JESTER dataset, which contains about 148k videos of people performing one of 27 gestures. Technically inference could be performed within the Pytorch framework but for acceleration the authors decided to use Apache TVM framework, which is an open-source machine learning compiler framework for CPUs, GPUs, and machine learning accelerators. An overview of the compilation process of TVM is given in Figure 7.

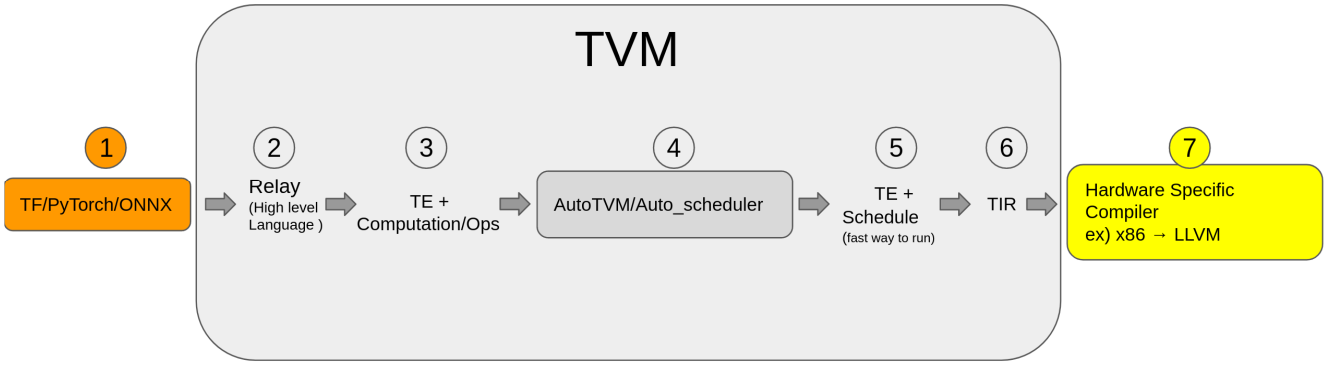

Figure 7: TVM compilation process

Step 4 is of special interest, as it comprises the search for a schedule that is optimized for the target hardware, a process referred to as tuning. The output of this optimisation step is a logfile that contains the schedule for each tensor operation of the neural network. This is required during runtime to compile the hardware specific code. Technically the tuning process is required for the Jetson Nano hardware, however for many operations on CUDA capable hardware, these logfiles are already available, so no further tuning is required. If this tuning is omitted and no suitable logs are available, fallback configurations are used, which lead to significantly worse performance in our experience.

The prerequisites for this model were installed in a Docker container based on the NVIDIA L4T ML image, as it already includes the dependencies for Pytorch and OpenCV. After concluding the installation steps given by the authors of the paper, the python script that performs recognition of the live video was adjusted. Concretely a OPC UA client was added to enable connectivity between AI component and automation component. The detected gesture is written to a OPC UA variable and is used to interact with the automation program of the machinery.

2.2.2 Deployment

The model is deployed in a Docker container to increase portability as well as encapsulate the functionality. To enable the container to access the GPU of the host system, NVIDIA container runtime is required, as shown in Figure 8. Consequently, the correct drivers must be installed on the host system. For Jetson devices these differ from drivers of the discrete GPUs, as the former are integrated directly, whereas the latter are connected via PCIe interface.

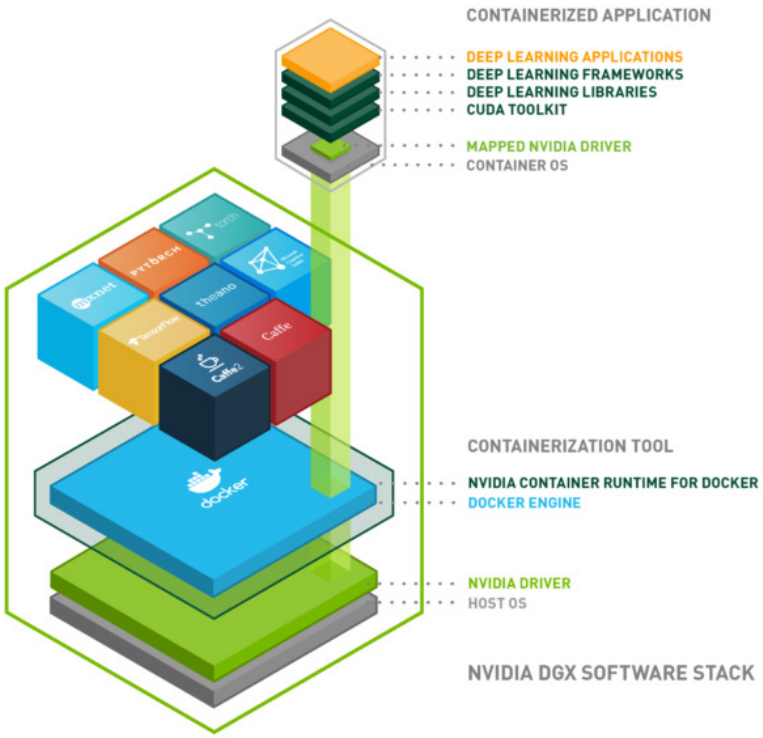

Figure 8: NVIDIA container toolkit

The NVIDIA JetPack SDK readily includes the drivers, deep-learning libraries, and higher-level SDKs, such as Deepstream mentioned earlier. Additionally, since v4.2.1 JetPack also includes the container runtime that allows using the Jetson's GPU within containerized applications. NVIDIA also offers a range of container images for the Jetson platform, which contain key machine learning and computer vision frameworks that are required in this project, such as OpenCV and Pytorch. These container images are a good starting point, since only TVM must be built. The finished container is then started together with the control application as a microservice using the Docker compose tool. The application opens a window on the display, therefore the window manager on the host system must allow clients to open windows and the webcam must also be passed as a device when running the microservice. Additionally, the NVIDIA runtime must be specified in case it is not set as default for the Docker daemon.

2.2.3 Inference

Analogous to the first approach, the inference performance was evaluated by running the model on the Jetson Nano in combination with a Logitech C920 USB camera with power supplied by the barrel jack. One advantage is that resolution of the video feed can be altered within the program. This approach to gesture recognition allowed stable frame rates around 30 fps. From the usability perspective, gestures were recognized with sufficient reliability, however, it was noticed that entering and exiting the frame with the hand is sometimes classified as a gesture. This can be avoided once one has a little experience with the gestures. As there are 27 classes in total (25 when excluding the no action classes), technically the same number of machine responses can be implemented. Regarding the interface to the control application, once the server and client were configured, communication via OPC UA worked error-free.

3 Real-time performance

3.1 Implementation

Standard Linux kernel (L4T), which is used in Jetson Nano is not real-time capable, but for PLCs, which are used to control a machinery, real-time performance is a mandatory requirement.

I/Os, which are used in the industrial automation system, are connected to the PLC (Jetson Nano) via EtherCAT (Ethernet based) and therefore real-time communication is also critical for the device.

To provide real-time for Linux kernel, PREEMPT_RT Patch is used. It is applied to Linux kernel source code, using standard script "kernel-4.9/scripts/rt-patch.sh" available in L4T.

CODESYSControl runs in the docker container and having real-time Linux kernel is not sufficient to execute the application within the container with real-time characteristics. The docker container must be started with the following parameters to be able to execute real-time applications, using docker compose file:

```
ulimits:
 rtprio: 99
cap_add:
  - sys_nice
```

The Linux image based on the resulting real-time Linux kernel 4.9.140-rt93 is prepared and used to run real-time performance tests and control machinery.

3.2 Real-time tests

Real-time tests are executed, using BE.services real-Time Test Framework.

BE.services Real-Time Test Framework (RTTF) is a product allowing to test real-time performance of an Industrial Control System and its Ethernet based communication. The architecture of RTTF is presented in Figure 9.

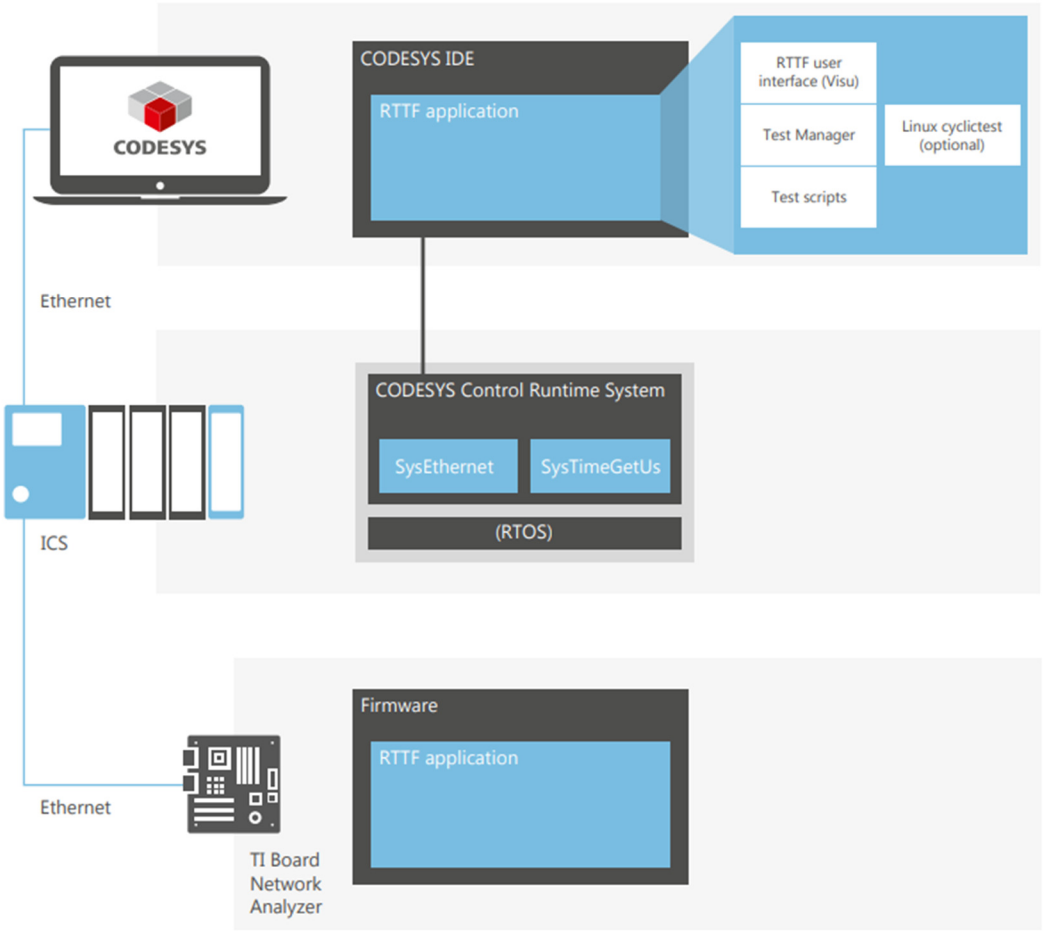

Figure 9: Real-time test framework

It consists of several parts:

1. CODESYS Real-time tests application. It is executed in the PLC. The user configures the test parameters in the CODESYS Visualization masks, which are displayed directly in CODESYS IDE or in web-browser. CODESYS IDE is connected via USB to Jetson Nano.

2. If necessary, the CODESYS Test Manager can be used to automate execution of different test scenarios. Such tests scenarios are described in the test scripts and can include for example execution of real-time tests during intensive usage of the file system or the GPU.
3. Optionally, Linux specific tests like cyclictest can be executed by the real-time test framework.
4. The network analyser is based on TI Industrial Communications Engine and connected to the ethernet adapter of Jetson Nano to measure the transmission jitter of Ethernet frames.

There are 3 types of measurements performed as shown in Figure 10.

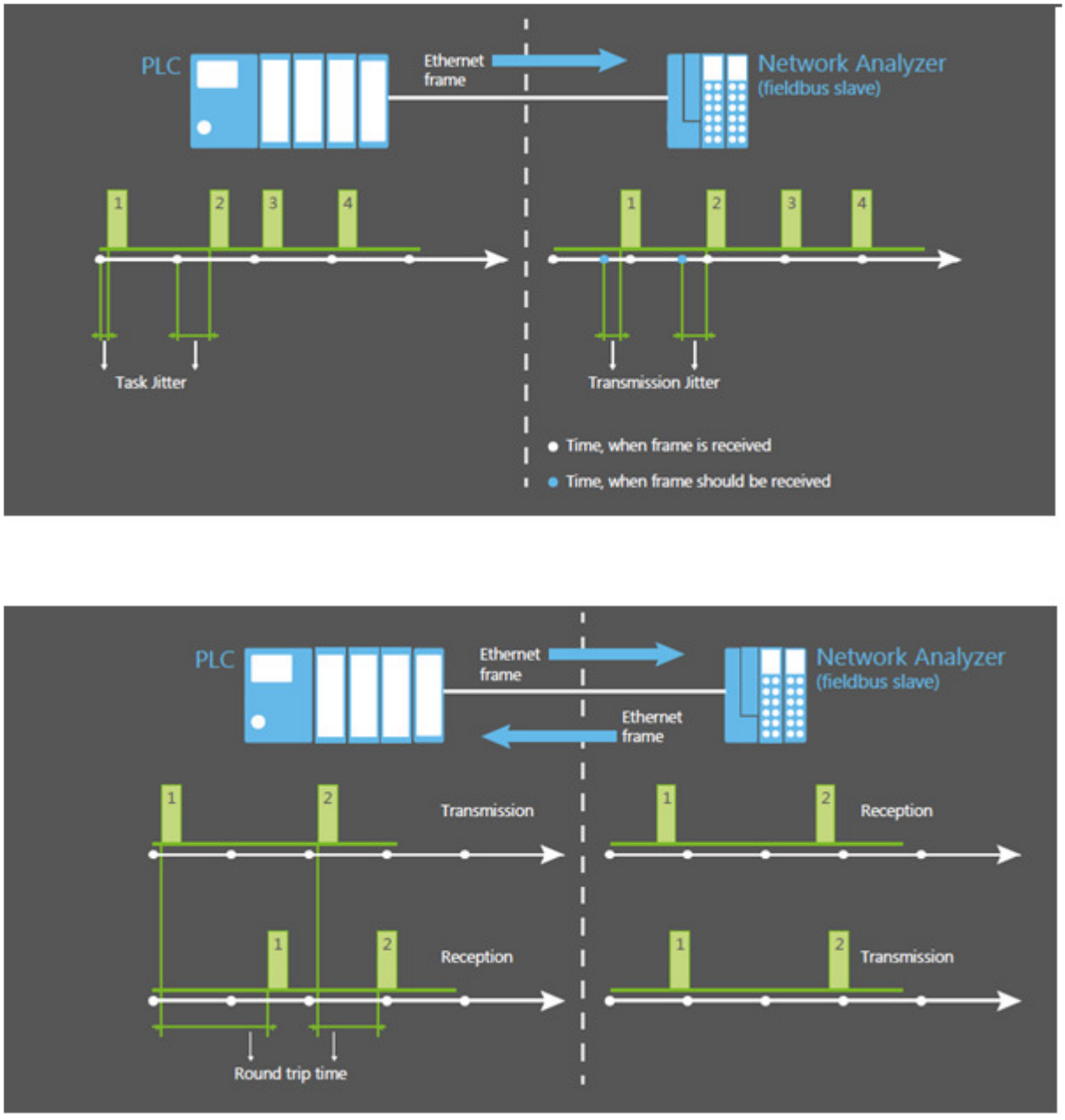

Figure 10: Real-time measurements

1. **Task jitter**. It is measured directly in the PLC by making high-resolution timestamps in every control cycle.
2. **Transmission jitter**. It is measured in the Network Analyser by receiving frames.
3. **Roundtrip time**. The network Analyser receives frames and send them back immediately. The test application measures the time between sending and receiving of frames.

The real-time performance was tested without any load and during running of ML application (gesture recognition). Tests were performed on real-time and standard Linux kernels.

Results with the standard Linux kernels without and with ML workload are presented in Figure 11 and Figure 12, respectively.

Figure 11: Results with standard Linux kernel without ML

Figure 12: Results with standard Linux kernel with ML

Results with the real-time Linux kernels without and with ML workload are presented in Figure 13 and Figure 14, respectively.

Figure 13: Results with RT Linux kernel without ML

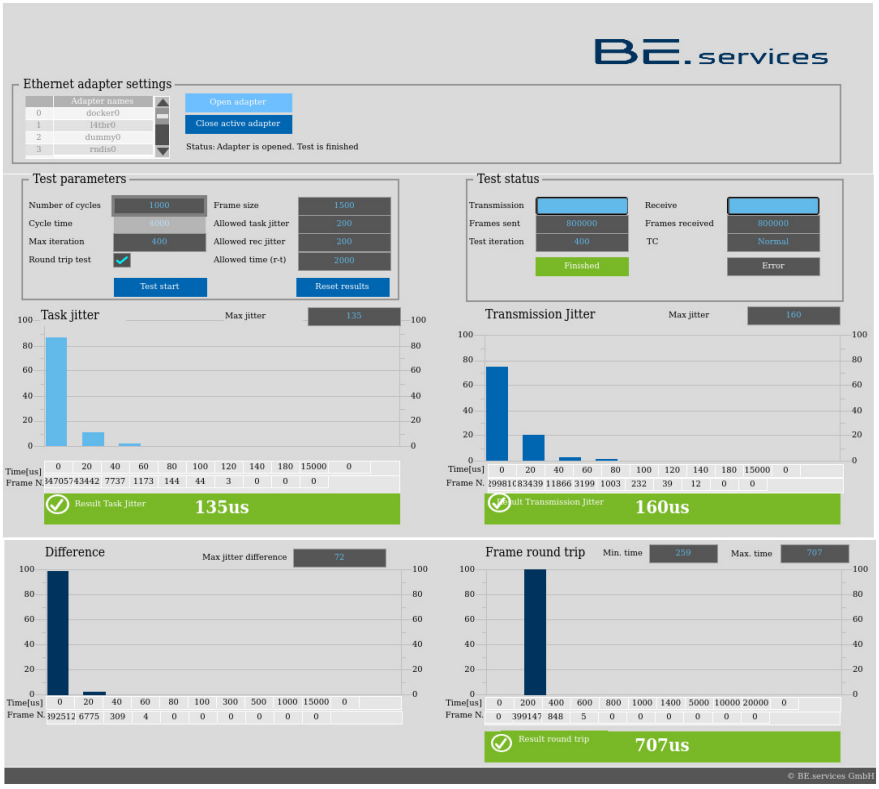

Figure 14: Results with RT Linux kernel with ML

As we can see, the real-time performance of standard Linux kernel is rather poor, especially if ML is executed. Obviously, it is not acceptable for most control applications.

That said, real-time Linux kernel (with Preempt_RT Patch) shows very positive results even on high CPU and GPU load during execution of ML.

4 Glossary

CNN	Convolutional Neural Network
GPU	Graphics Processing Unit
L4T	Linux 4 Tegra, board support package from NVIDIA, contains Linux kernel, NVIDIA drivers
ML	Machine learning
PLC	Programmable logic controller
RT	Real-time
RTTF	Real-time test framework
SDK	Software development kit
TLT	Transfer Learning Toolkit, software toolkit from NVIDIA
TVM	Apache TVM machine learning compiler and runtime framework

Bibliography

[1] K. Sirazitdinova and A. Sardana, "Github," [Online]. Available: https://github.com/NVIDIA-AI-IOT/gesture_recognition_tlt_deepstream. [Accessed 10 May 2021].

[2] J. Jung, "Github," [Online]. Available: https://github.com/jkjung-avt/hand-detection-tutorial/blob/master/prepare_egohands.py. [Accessed 10 May 2021].

[3] J. Lin, C. Gan and S. Han, "TSM: Temporal Shift Module for Efficient Video Understanding," *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV),* 2019.