POLITECNICO DI TORINO

Master degree course in Data Science and Engineering

Master Degree Thesis

Neural Networks hardware-specific optimization using different frameworks

Supervisor
prof. Paolo Garza

Candidate
Riccardo Bosio
291299

Internship Tutor
AllRead's CTO and Co-founder, Marçal Rossinyol

Academic Year 2021-2022

Summary

Artificial Intelligence (AI) is a fast growing sector where investments are increasing, the number of applications is getting bigger and the total amount of publications is doubled from 2010 to 2021.

Among AI fields there is Computer Vision: the goal is to process images or videos and extract relevant information depending on the task we are solving. A huge amount of data is used to train a Deep Learning model to accomplish a specific function.

We can end up having a Neural Network that is able to accomplish a specific task and we can run this model on several hardware, such as CPUs and GPUs. Moreover in several applications it's really important not only to have good quality metrics, but also to process data as fast as possible. The inference time is really important and the trade-off between speed and accuracy is something that should be assessed carefully keeping in mind the context in which we want to deploy our Neural Network.

In this scenario where we can find clients with different hardware and needs, the possibility of optimizing our models to run faster without losing the original accuracy can be a game changer.

In this thesis I tried to reach this goal using different optimization frameworks and testing the improvements of three models. I had the opportunity to do my experiments in a real world scenario, working at AllRead Machine Learning Technologies: it is a Barcelona based startup delivering a Deep Learning based OCR (Optical Character Recognition) software for Supply Chain and Industry 4.0.

The first models that I've optimized in my experiments are the detector and the reader that compose the AllRead's OCR system which is reading license plates. Finally I've applied the different optimization pipelines on the Damage Detection Demo model, a semantic segmentation net that I had previously trained for a potential customer.

At the beginning I implemented a pipeline using Apache TVM, an open

source end-to-end machine learning compiler framework for CPUs and GPUs that enables optimization on any hardware back-end. The results obtained by the TVM optimized company's models were not good on my Intel i5 CPU, therefore we decided to follow another approach: since more than 90% of the company's clients usually have Intel CPUs, we can use OpenVINO to optimize the models.

OpenVINO is an open-source toolkit for optimizing and deploying AI inference specifically for Intel products. The results obtained using this framework are really good: the optimized models are up to 2.6 times faster on Intel core i5 and up to 3.9 times faster on Intel core i7.

On the other hand sometimes clients ask for an edge computing solution and a valid choice in this context is to provide them with an NVIDIA GPU. This is the reason why we decided to optimize the models to run on this specific hardware. To reach this goal I first explored TensorRT.

TensorRT is an SDK for high-performance deep learning inference on NVIDIA products. I tested its performances on an NVIDIA Jetson Xavier available at the office. Using TensorRT I reach the best performances in terms of inference time. Anyway there are still other steps involved in an end to end pipeline, such as pre-processing and post-processing. In order to take advantage of the TensorRT inference time and optimize the entire stream of data I used DeepStream as a last optimization framework.

NVIDIA DeepStream Software Development Kit (SDK) is an accelerated AI framework to build intelligent video analytics pipelines. The results obtained with this optimization are really good since real-time inference is reached. Moreover I can definitely say it is worth for a customer to invest on an NVIDIA edge device: with this optimization tool, the final pipeline is 8x faster than the OpenVINO optimized one running on an Intel i5 CPU.

Acknowledgements

Thank you Franco and Sonia for doing your best everyday in being my parents. I wouldn't be the man I am today without you being my source of inspiration.

Thank you Leonardo for always getting a smile out of me.

Thank you Beatrice for the invaluable help you have given me during these years. Sharing this journey with you has changed me as a person and I can't be more proud of this.

Thanks to my grandparents for the sacrifices they made when I was a child.

Finally, thank you prof. Paolo Garza, you are definitely the most professional professor I have met during these five years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

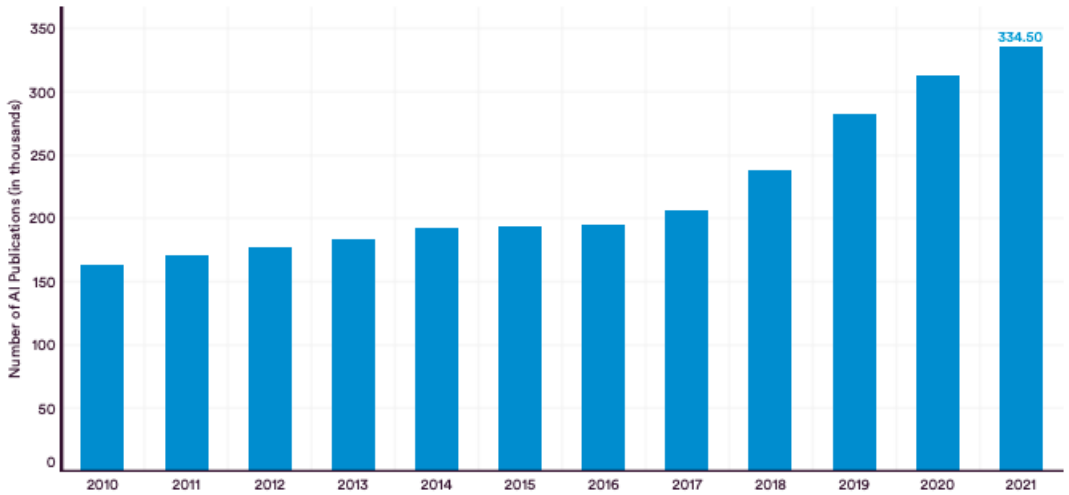
1.1 Artificial intelligence

In 2022, the Artificial Intelligence (AI) market size value is 136.6 billion USD and the forecast growth rate between 2022 and 2030 is 38.1% [1].

Applied AI and industrializing machine learning are two of the most significant technology trends unfolding today. Tech industries are leading in AI adoption, while product development and service operations are the business functions that have seen the most benefits from applied AI.

As you can see in Figure 1.1, from 2010 to 2021, the total number of AI publications doubled, growing from 162,444 in 2010 to 334,497 in 2021 [2].

Figure 1.1. Number of AI publications in the world, 2010–2021.



In 2021, global private investment in AI totaled around 93.5 billion USD, which is more than double the total private investment in 2020.

Both research and investments are active in AI sector, therefore the trend seems quite promising. Hopefully economic issues won't change AI's powerful momentum.

1.2 Computer Vision

Computer vision (CV) is a field of Artificial Intelligence (AI) that focuses on enabling computers and systems to derive meaningful information from digital images, videos and other visual data.

As of 2021, computers can outperform humans in many computer vision tasks. There is a wide range of computer vision tasks, such as image classification, object recognition, semantic segmentation, and face detection.

Computer vision technologies have a variety of important real-world applications, such as autonomous driving, crowd surveillance, sports analytics, and video-game creation.

In this thesis I will work with models solving the following CV tasks: Optical Character Recognition (object detection + reader) and Semantic Segmentation.

1.3 Optical Character Recognition

Optical Character Recognition (OCR) is the process that converts an image of text into a machine-readable text format. The goal is to detect and read the text present in an image.

The global optical character recognition market size was valued at 8.93 billion USD in 2021 and it is expected to expand at a compound annual growth rate (CAGR) of 15.4% from 2022 to 2030 [3].

The growth of the OCR market is primarily attributed to the improvement in productivity and a rise in the penetration of automatic recognition systems: widespread adoption of OCR technology has been observed in healthcare, retail, tourism, logistics, transportation, government, manufacturing, and other sectors.

Digitalization in business organizations has made all the processes faster and more accessible. As companies witness technological advancements, data is becoming a critical element for growth. When data is converted to digital

form, it can be processed by computers and various devices with computing capacity, and this data is easy to share, access, and store.

1.4 Semantic Segmentation

Semantic Segmentation is the Computer Vision task whose goal is to label each pixel of an image with a corresponding class of what is being represented.

It can be considered as a powerful alternative to object detection since it allows the object of interest to cover multiple areas of the image at the pixel level. As a matter of fact Semantic Segmentation techniques cleanly detect irregularly shaped objects, unlike object detection, where objects get surrounded in a bounding rectangle.

Thanks to this accuracy, Semantic Segmentation is useful in applications in multiple fields, such as Autonomous Driving or Robotic Vision.

The main drawback is that most of the relevant methods in Semantic Segmentation rely on a large number of images with pixel-wise segmentation masks. Obviously, manually annotating these masks is quite time-consuming, frustrating and commercially expensive. Therefore, some weakly supervised methods have recently been proposed: they operate the Semantic Segmentation task by using annotated bounding boxes, or even image-level labels.

1.5 AllRead Machine Learning Technologies

AllRead MLT. is a startup founded in 2019 that offers a Deep Learning based OCR software for Supply Chain and Industry 4.0. The goal is to improve the operational efficiency by automatic data capture in images and videos, reducing repetitive manual tasks, eliminating errors and allowing immediate processing of the information.

The main tasks the software can solve are Access Control, Security and Safety, Network Asset Visibility, Inventory, Digitalization, Monitoring. Right now the company is focusing on reading shipping containers, UIC wagons, license plates, utility meters, unit load devices, QR and Barcodes. The software can be installed on premise, on cloud or on device.

1.6 Neural Networks optimization

AllRead has models solving different tasks that can be delivered to any client. Each client has different needs and can choose a specific hardware to run the company's software on: in this scenario it would be a game changer to be able to optimize the model inference on that specific hardware.

This is exactly the goal of my thesis: explore different frameworks and techniques to perform hardware-specific optimization. The idea is to speed up the inference time without losing the original accuracy.

Chapter 2

Literature

2.1 Computer Vision tasks

As mentioned in the Introduction, Computer Vision is the branch of AI that deals with images and videos. These inputs are processed in order to extract useful information, depending on the task. In this thesis three models solving different tasks are optimized with various frameworks. The tasks involved are the following:

- Object detection: task of detecting instances of objects of a certain class within an image. The state-of-the-art methods can be categorized into two main types: one-stage methods, that prioritize inference speed (such as YOLO and SSD), and two stage-methods, that prioritize detection accuracy (such as Faster R-CNN and Mask R-CNN).

- Text recognition: task of recognizing written or printed characters such as numbers or letters. Optical Character Recognition systems are often composed of a detector, which detects the text in the image (object detection), and a reader, that reads the detected text (text recognition).

- Semantic segmentation: task of classifying each pixel in an image from a predefined set of classes. It is different from object detection since in this case we are not predicting any bounding box and we are not distinguishing between different instances of the same object.

The first two tasks are achieved in the so called License Plate pipeline, which is an Optical Character Recognition system that reads the license plate of vehicles from frames. Semantic segmentation is instead done by the Damage

Detection model. More information about the models will be presented at the beginning of Chapter 3.

2.2 OCR

Humans read all kinds of textual information daily, since text is present everywhere in man-made environments and it conveys relevant information about the world around us.

Even if OCR engines have been around for several decades, we have seen in the last years a revolution in this field, powered by the latest advancements in deep learning. Nowadays, state of the art methods are able to locate and read text not only in scanned documents but also in natural scenes "in the wild" [4] [5].

In the industry context, text usually contains structured information to be transmitted. Those textual information are not easy to read by human operators. Moreover humans usually read with a word-level processing of the textual context, therefore any structured textual content requiring a character-by-character interpretation is not natural for humans and can lead to mistakes that can have important consequences.

When it comes to structured text, machine-based automated reading software can be the solution, but instead of a generic OCR a different approach is needed, adaptable and specialized to each reading domain.

The solution proposed by AllRead [6] is based on a convolutional neural network that processes images and outputs the desired structured texts in a single shot. The network is composed of a convolutional backbone, that has the goal of extracting visual features, followed by stacking several independent fully connected layers, that specialize in predicting the output probabilities for each symbol, each one with a Softmax activation producing a probability distribution over the n possible classes for each expected symbol of the final reading. The n outputs of the network are then treated as a typical classification output and trained using the Cross-Entropy loss function.

The main advantages of this end-to-end model are that the network can be trained with full images, without any explicit segmentation, directly optimizing the end-to-end reading performance and that the particular design of the network allows for real time reading speeds while achieving high reading accuracy.

2.3 TensorFlow

TensorFlow (TF) [7] is an interface to express and execute machine learning algorithms for both training and inference. It is used for conducting research but also for deploying systems into production, thanks to its ability to run, after small or no changes in the algorithm specification, on a wide and heterogeneous set of systems such as mobile devices and distributed systems. The TensorFlow API has been first released in 2015. In TensorFlow graphs are very important: we can describe a computation as a directed graph, where nodes are operations and typed, multidimensional arrays called tensor flow along the edges.

The main TensorFlow components are the following:

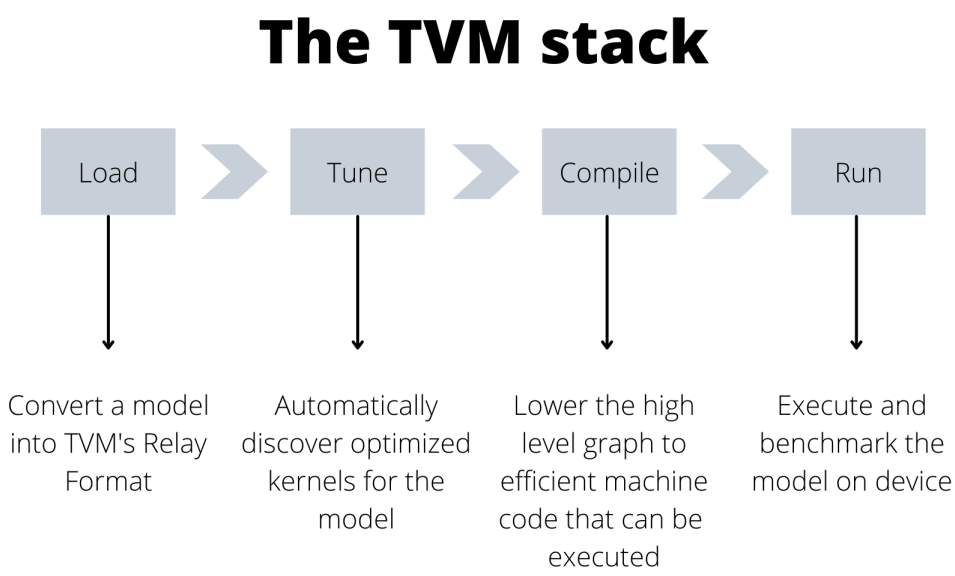
- Operation: it represents an abstract computation and it has a specific name.
- Kernel: the implementation of an operation that can be run on a particular type of device.
- Session: when a client interacts with the TensorFlow system, it creates a session.
- Variable: a special kind of operation that returns an handle to a persistent mutable tensor that survives across executions of a graph.

In TensorFlow, after training, there are two different ways to save the state of a graph in order to freeze its weights. One way is to get the following files: a *.meta* file holding the metadata and the graph structure, a *.index* file, a checkpoint file and a *.data* file with the weights. In this case, it's necessary to have the source code to run inference. The other way is to generate a Protobuf (*.pb*) file, which is a single file containing all the information needed to run inference.

2.4 Apache TVM

Apache TVM [8] is an end-to-end machine learning compiler framework for CPUs, GPUs, and machine learning accelerators. It is open source and its goal is to enable optimization and running computations efficiently on any hardware back-end. This flexibility is a pro of TVM that I would like to underline: we can potentially use it to optimize the models' inference on any hardware created by any company (Intel, AMD, NVIDIA, etc.).

Figure 2.1. The Apache TVM optimization process.



The optimization process can be summarized as in Figure 2.1.

First of all the model is converted into TVM's Relay Format. The front-end component is able to automatically understand the frameworks of the model and to ingest it into an IRModule, which contains a collection of functions that internally represent the model. Relay [9] is a high level graph representation language designed to balance efficient compilation, expressiveness, and portability. It is a statically typed, purely functional, differentiable Intermediate Representation. An intermediate representation (IR) is the data structure or code used internally by a compiler or virtual machine to represent source code. Relay was intended as the top layer of the TVM stack.

In the tuning step, there are the compute expressions, that basically tell us which are the operations and how a certain output is computed, and there are the schedules, that are the ways in which these expressions can be rewritten. This optional step uses machine learning to look at each operation within a model and tries to find a faster way to run it. TVM optimizes across multiple layers in the following way: it looks at the model, it divides it in multiple workloads and then optimize them . For each workload, a set of possible schedules is generated and a final best schedule is selected. A tuning table is produced: low latency and high GFLOPS are better. GFLOP stands

for Giga FLOP: FLOP means floating point operations per second and it is a measure of performance, assessing how fast the computer can perform calculations.

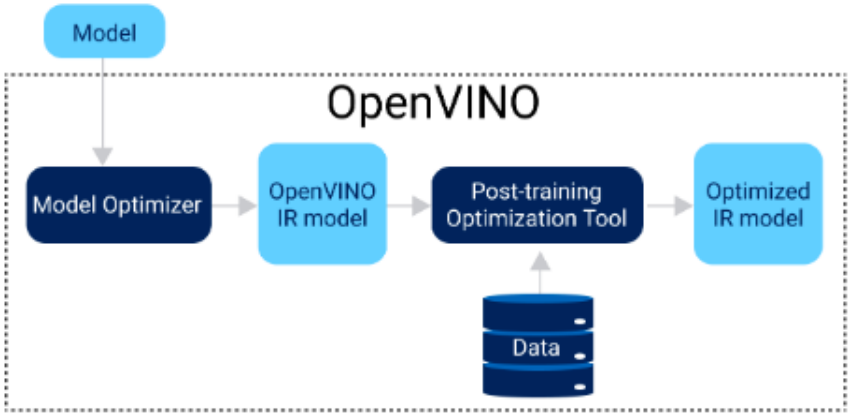
During the compilation step, the model from Relay is translated into a lower-level language that the target machine can understand. The target is the desired hardware we want to run the model on.

Finally we run inference using the TVM runtime and we collect metrics. It's important to remind that the accuracy is not going to change: we are trying to boost inference on a specific hardware.

2.5 OpenVINO

OpenVINO [10] is an open-source toolkit for optimizing and deploying AI inference on Intel products. The general OpenVINO optimization workflow can be seen in Figure 2.2.

Figure 2.2. The OpenVINO workflow.



Optimization happens in two stages:

1. Model Optimizer (MO): it is a command-line tool that converts a model trained with a supported framework into an Intermediate Representation (IR), which can be inferred with the OpenVINO runtime.

2. Post-training Optimization Tool (POT): it provides two main 8-bit quantization methods: Default Quantization and Accuracy-aware Quantization.

The Model Optimizer step is mandatory and it already improves performances. POT is an additional step to get a more optimized model.

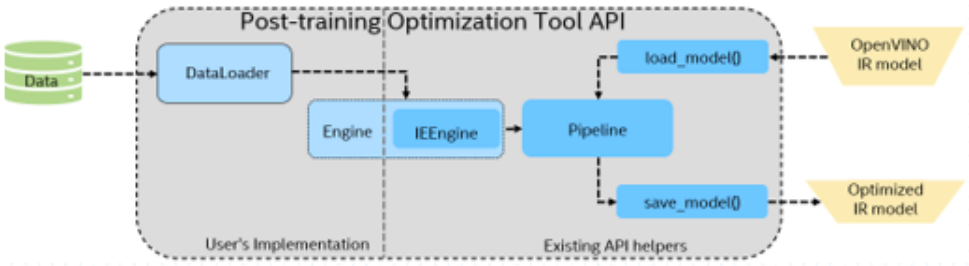
The following are the main MO tasks:

- Convert the format: a model from one of the recognized framework is converted into IR. IR is basically composed of two files: an *.xml* file that describes the layers, the connectivity, the parameters and a binary file that holds all the weights and biases.
- Optimize: it can apply different techniques, such as batch normalization, layers fusion and others hardware agnostic optimizations that can save a lot of computations and memory.
- Convert weights and biases: for example they may change from FP32 to FP16.

2.5.1 Quantization

In Figure 2.3 there is an high level description of the POT workflow.

Figure 2.3. The Post-training Optimization Tool (POT) workflow.



In particular, POT offers two different quantization methods: Default Quantization and Accuracy-aware Quantization. Usually a trained neural-network model is in full-precision format, i.e. FP32. What quantization does is to convert the floating-point number operations to low-bit numbers, such as Int8. This not only reduces the size of the model but also the computational cost to a great extent.

During quantization using POT, an operation, named FakeQuantize, is added into the model graph, based on the target hardware. Basically, the model is optimized according to the computation device on which the POT operation is carried on. During runtime, these FakeQuantize layers convert

the input to the convolution layer into Int8. For example, if the next convolutional layer has Int8 weights, then the input to that layer will also be converted to Int8. Further on, the precision however depends on the next operation. If the next operation requires a full-precision format, then the inputs will be reconverted to full-precision during runtime. Let's take a look at the two algorithms more in detail.

As you can see in Figure 2.4 the Default quantization follows these operations:

1. The input is the full-precision model.

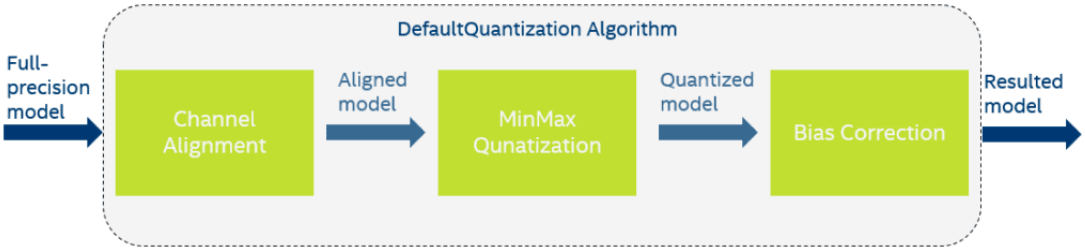
2. Activation Channel Alignment is applied in order to align the activation ranges of the convolutional layers and reduce quantization error. This is done first calculating the mean of the activation values and then aligning them by clipping the activation values within a certain range.

3. Depending on the target hardware chosen, the MinMax Quantization method is performed to insert the FakeQuantize layers into the model graph.

4. Bias Correction is applied to the quantized model in order to make the model output unbiased. Since quantization tends to shift the network's statistics from the learned distribution, bias correction helps to overcome this, by adding a constant to the bias term of each channel, in every layer of the neural network.

Although with Default Quantization you get the fastest quantized model, the main drawback is that you cannot control its accuracy. Most of the time this might not be an issue, but there will be situations where such control becomes crucial.

Figure 2.4. The Default Quantization algorithm.

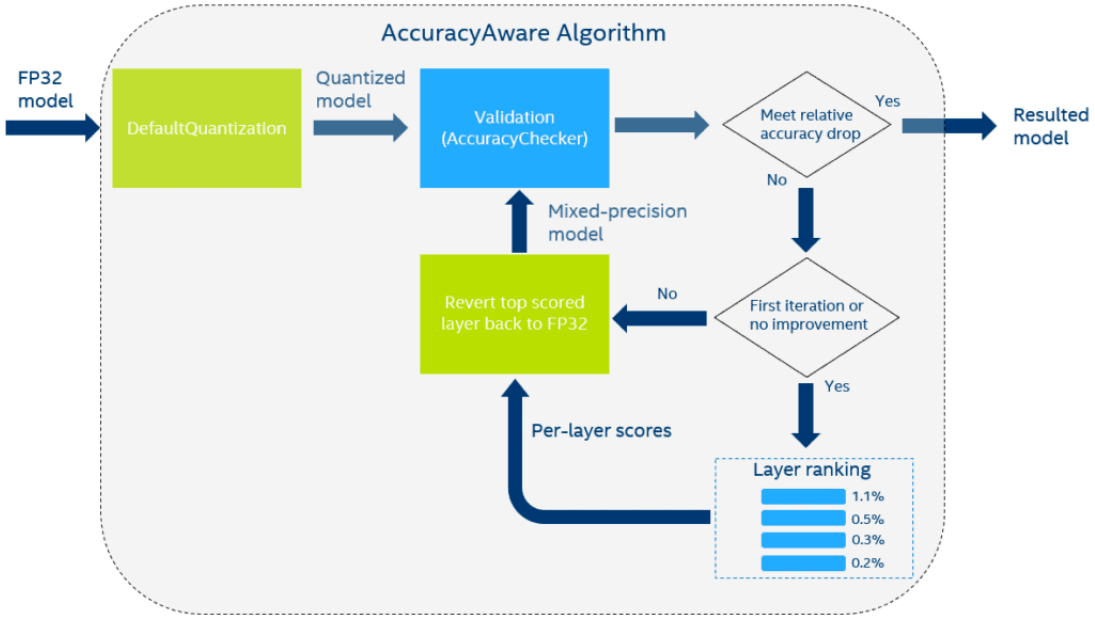


To tackle this issue you can use the Accuracy-aware quantization algorithm, since it does not let the resulting quantized model drop below a predefined accuracy range. This method should be used whenever inference accuracy of the INT8 model is as important as its inference speed.

In Figure 2.5, you can see the Accuracy-aware Quantization workflow:

1. The input is the full-precision model, which passes through the Default Quantization algorithm to output a fast, 8-bit quantized model.
2. The quantized model infers on a sample-validation set, which provides an accuracy score. If the accuracy level is satisfactory and matches the required threshold, the resulting model is given as output. Otherwise, since the accuracy score does not match the criteria, a number of extra steps is needed before reaching the output stage.
3. If it is the first iteration, a layer-wise ranking is done to check which layer affects the accuracy the most.
4. The quantized layer contributing most to the accuracy drop is completely reverted back to the original full-precision format.
5. The process restarts from point 2.

Figure 2.5. The Accuracy-aware Quantization algorithm.

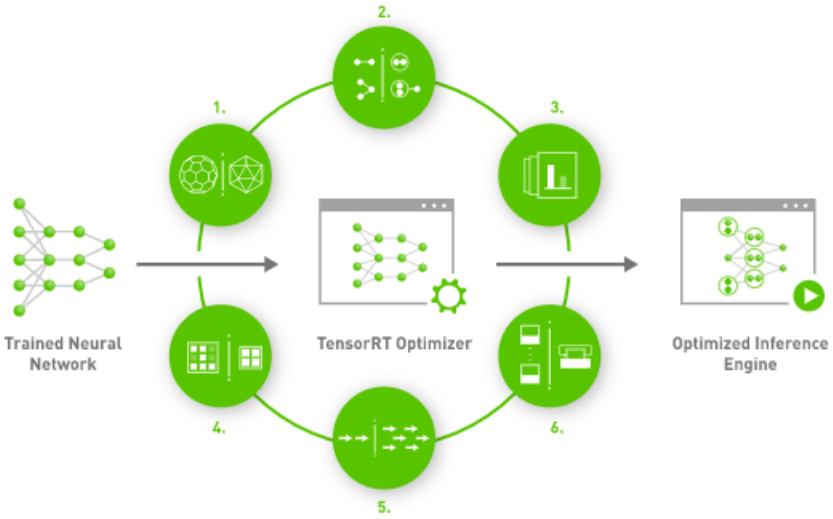


2.6 TensorRT

NVIDIA TensorRT is an SDK designed to deliver high-performance deep learning inference: it includes a deep learning inference optimizer and runtime.

The TensorRT workflow can be seen in Figure 2.6.

Figure 2.6. The TensorRT workflow.



The tasks that the TensorRT optimizer can accomplish are the following:

1. Weight Activation Precision Calibration: throughput maximization by quantizing models to INT8 while preserving accuracy.
2. Layer Tensor Fusion: nodes fusion in a kernel in order to optimize the use of GPU memory and bandwidth.
3. Kernel Auto-Tuning: best data layers and algorithms selection based on target GPU platform.
4. Dynamic Tensor Memory: memory footprint minimization and re-using memory for tensors efficiently.
5. Multi-Stream Execution: scalable design to process multiple input streams in parallel.
6. Time Fusion: recurrent neural networks optimization over time steps with dynamically generated kernels.

2.7 DeepStream

DeepStream [11] is an NVIDIA's SDK that delivers a complete streaming analytics toolkit to rapidly develop Vision AI applications and services that can be deployed on-premises, on the edge and in the cloud. Using DeepStream, developers have great flexibility since they can develop in C/C++, Python, or use a low-code graphical programming with Graph Composer. Moreover DeepStream offers an extensive AI model support for popular object detection and segmentation models (such as YOLO, SSD, and others) and there is also the possibility of integrating custom functions and libraries in order to use customized models. It uses GStreamer, which is a library for constructing graphs of media-handling components that range from simple audio/video streaming to complex audio mixing and video processing.

The main advantage is that you can use TensorRT for inference, which is the best optimization framework for NVIDIA GPUs, and meanwhile you optimize also the entire stream of data. In this way you improve not only the inference time, but also the other steps involved in an end-to-end pipeline, such as the pre-processing, post-processing and so on.

NVIDIA introduced Python bindings to help you build high-performance AI applications using Python, a very easy to use language widely adopted by data scientists when creating AI models. The DeepStream Python application uses the Gst-Python API action to construct the pipeline and use probe functions to access data at various points in the pipeline. The data types are all in native C and require a shim layer through PyBindings or NumPy to access them from the Python app. Tensor data is the raw tensor output that comes out after inference. If you are trying to detect an object, this tensor data needs to be post-processed by a parsing and clustering algorithm to create bounding boxes around the detected object.

2.8 Hardwares and development environment

As already mentioned in the introduction, the hardware I used in the experiments are three: two CPUs, Intel Core i5 dual-core and Intel Core i7 - 8650U, and one GPU, NVIDIA Jetson Xavier. Running tests on CPUs was pretty straight-forward since I had the machine in house. Using the Jetson has been more complicated: it has been installed in headless mode in the office and I had to connect via Secure Shell (ssh) to access it.

Secure Shell is a network communication protocol that enables two computers to communicate and share data in a secure way, using encryption. In particular I have used ssh to "login" on the Jetson remotely and then be able to perform operations.

In order to transfer data, I have used the Secure copy protocol (SCP), which is a means of securely transferring files between a local and a remote host. SCP is based on the Secure Shell protocol.

Concerning the DeepStream prototype, since I didn't have any NVIDIA GPU on my laptop, I mainly developed the code locally using Pycharm and then I moved the code on the Jetson to test it. Same for the C++ shared library (.so) that I generated: more details in chapter 3.

Chapter 3

Problem Statement and Solutions

3.1 The goal

AllRead is delivering different products to different clients. Clients have different hardware on which they can run AllRead's models. The main goal of my thesis was to create and test different optimization pipelines in order to deliver optimized models depending on the client's hardware. For example if a specific client is interested in having model X running on hardware Y, we want to provide him with X optimized to run on Y. In order to achieve this goal, I developed and tested different optimization pipeline involving different frameworks. The models I used to test these optimizations were three: a detector, a reader and a semantic segmentation model. In this way I covered all the main Computer Vision tasks. In sections 3.2 and 3.3 there is an explanation of the models used, while in the other sections of the chapter the different optimization pipelines are described.

After applying each of the following pipelines, I benchmarked the optimized models to assess the possible loss of accuracy/mIoU.

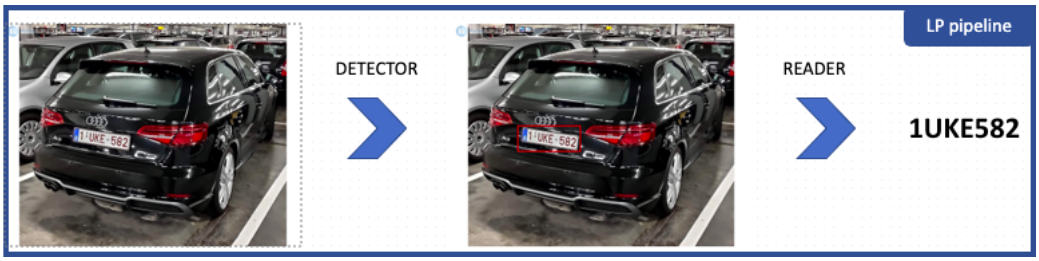
3.2 The License Plate pipeline

The License Plate pipeline is one of the OCRs that AllRead is deploying to the customers and it is mainly used for access control in ports. As you can see in Figure 3.1, it is composed of a detector and a reader. Later on I will refer to these models as the License Plate detector and the License Plate

reader. The two models perform these actions inside the LP pipeline:

1. LP detector: it takes the full image as input and it produces a bounding box around the license plate in the frame.
2. LP reader: the input is the original image cropped on the bounding box area generated by the detector. The outputs are the reading and the country of the cropped license plate.

Figure 3.1. The LP pipeline.



I will not provide any additional information about these models because of Intellectual Property.

In order to perform my experiments I had to know how to pre-process the input image and how to post-process the outputs of the models: with this knowledge and the Protobuf files of both the detector and the reader, I was able to apply all the optimization pipelines described in the following sections.

3.3 The Damage Detection demo

The idea behind this model was to develop a demo to show potential clients what we can achieve so far and then to propose a partnership in order to improve the model with more data from them. In Figure 3.3 you can see how the Damage Detection model works: it gets an image as input and it generates a segmentation map of the damages in that image. In the following subsections I present the model's architecture first and then I describe how I trained it from scratch.

3.3.1 The U-Net inspired architecture

The model architecture takes inspiration from the U-Net paper [13]. As the paper explains, it consists of:

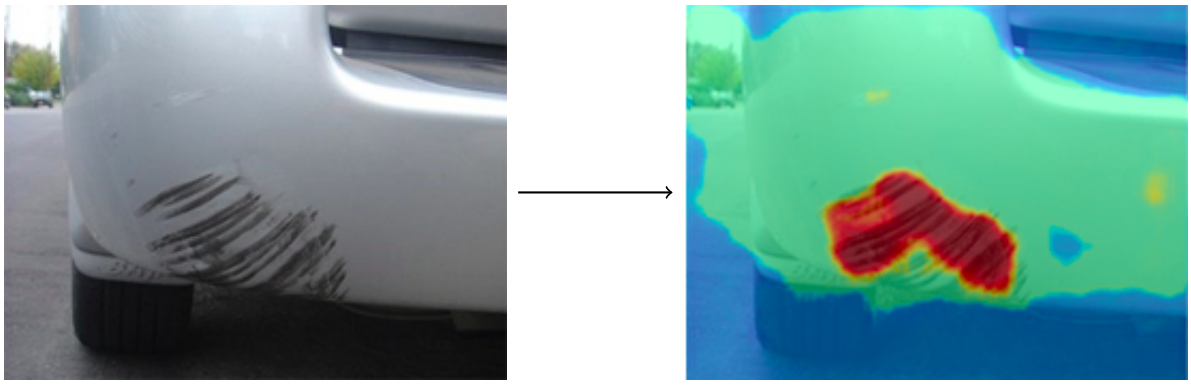


Figure 3.2. The Damage detector input and output.

- a contracting path: it has the typical architecture of a convolutional network. It is composed of repeated applications of two 3x3 convolutions, each one followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation.
- an expansive path: every step consists of an upsampling of the feature map, a 2x2 convolution, a concatenation with the corresponding cropped feature map from the contracting path and two 3x3 convolutions, each one followed by a ReLU. The final layer is a 1x1 convolution.

I have used MobileNet [12] pre-trained on ImageNet as backbone.

3.3.2 The training

In order to build a training dataset I explored around 3000 images the company already had. I didn't annotate all of them: there were many different damages, some of them were very small and some others were big, there were very old cars and pretty new ones, there were also broken windows and missing pieces. The goal was to be coherent while building the dataset. Therefore at the end I annotated 833 images for training and 50 for validation. The test set was composed of 50 images. The tool I used to annotate images is `labelme` [14].

I have trained the model for 201 epochs, using a batch size of 16 and a learning rate of 0.001 with Adam as optimizer. The losses I tested are the weighted cross entropy and the penalty-reduced pixel-wise logistic regression with focal loss. The first one performed better in this scenario.

The augmentations are taken from the `albumentations` package: in particular I used `ShiftScaleRotate()`, `LongestMaxSize()`, `PadIfNeeded()`, `RandomCrop()`, `RandomGamma()`, `ColorJitter()`, `ToGray()`, `MotionBlur()`, `GaussNoise()` and `JpegCompression()`. The hyperparameters finetuning involved the batch size, the learning rate, the loss and the parameters of some augmentations (in particular the rotation angle of `ShiftScaleRotate()`). The metric used to evaluate the model is the Intersection over Union.

The final performances of the model were not bad: the IoU is actually not very significant since identifying the real shape of the damages is really subjective. The best way to understand how the model performs is to look at the segmentation maps of the test set. In the test set there were images taken from different distances and also the size of the damages varied a lot. It emerges that the model struggles to detect small damages in images taken from afar, while when pictures are taken from a closer distance the model performs better.

3.4 The Apache TVM optimization pipeline

Apache TVM is the first framework I tried out: I built two different optimization pipelines, one using the python API (AutoTVM), and one using a command line application (TVMC Python). I first tested the pipelines on some standard architectures from Keras and then I applied the optimization to the models described at the beginning of this section.

3.4.1 AutoTVM

The TVM optimization framework has APIs available for different languages: in particular I have used the Python API (AutoTVM). In order to build an optimization pipeline using AutoTVM, I need to define the following:

- Target: a `tvm.target.Target` object that contains the target description. The target is the hardware we want to optimize our model for. In my optimization pipeline it is `target = tvm.target.Target("llvm", host="llvm")`.

- Frontend: a `tvm.relay.frontend` to import models from different frameworks. In the pipeline I built I have used the method `from_tensorflow()` to load the TensorFlow graph from the Protobuf file into relay.

- The model and its input: it is necessary to consider the model that is being optimized and to define a valid input for this model. In this case, I created a variable containing the Protobuf of my model and I pre-processed a real image to obtain the corresponding input vector.

After defining these variables we can first generate a Relay function to run on TVM graph executor, using `tvm.relay.build()`, and then we obtain the final wrapper of the TVM module, using `tvm.contrib.graph_executor.GraphModule()`. Before these two operations, we can add a tuning step to optimize our model: without this step, no hardware specific optimization is applied and the model is only compiled to work on the TVM runtime. It's important to add this tuning step because if we run inference using the compiled model we might not obtain the expected performance. Therefore we need to define additional elements:

- Runner: set up basic parameters for the TVM runner and then create it using `tvm.autotvm.LocalRunner()`. The main parameters include `number`, the number of different configurations that we will test, `repeat`, how many measurements for each configuration, `timeout`, maximum time on how long to run training code for each configuration. In my case, `number = 10`, `repeat = 1` and `timeout = 10`.

- Tuner: create a `tvm.autotvm.tuner` and specify its basic parameters, such as `early_stopping`, which is the minimum number of trials to run before a stopping condition can be applied, and `trials`, which is the number of trials. In my pipeline I used the `XGBTuner()`, `early_stopping = 100` and `trials = 1500`.

We can now add the tuning process: TVM will try running many different operator implementation variants to see which perform best. To do this it is going to use Machine Learning and in particular the algorithm we specified as `tvm.autotvm.tuner`.

3.4.2 TVMC

I built the optimization pipeline using TVMC Python, an high-level API for TVM. In particular, TVMC is a tool that exposes TVM features such as auto-tuning, compiling, profiling and execution of models through a command line interface.

TVMC simplifies a lot the optimization flow: there is one method for each required stage of the TVM workflow reported in Figure 2.1.

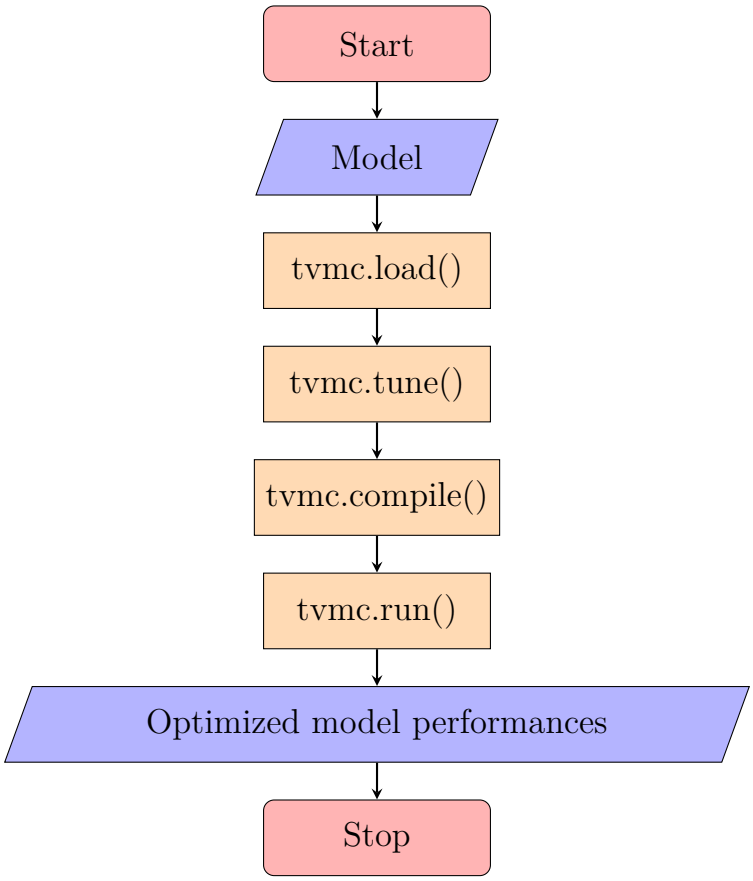


Figure 3.3. The TVMC Python pipeline code's flow diagram.

In Figure 3.3 you can see the code flow diagram of the TVMC Python optimization pipeline.

The result of importing a model into TVM is a `TVMCModel` object, which contains the pre-compiled graph definition and parameters that define what the model does.

Compiling a `TVMCModel` produces a `TVMCPackage`, which contains the generated artifacts that allow the model to be run on the target hardware. This is the optimized model: if you want to use it in production, you need to export this `TVMCPackage`.

Finally running a `TVMCPackage` produces a `TVMCResult` object, which contains the outputs of the model and the measured runtime. In this step you can pass the hyperparameter `inputs`, which is a dictionary that maps input names to Numpy values. If not provided, inputs will be generated using the `fill_mode` argument, whose valid options are `[zeros, ones, random]` and

the default is `random`.

3.5 The OpenVINO optimization pipeline

Concerning OpenVINO, I built two different optimization pipelines: the first one is very simple and exploits the standard functionalities of the framework, while the second one is an extension of the previous one.

3.5.1 Model Optimizer

The Model Optimizer is a command-line tool that converts a model trained with a supported framework (TensorFlow in this case) into an Intermediate Representation (IR), which can be inferred with the OpenVINO runtime. The IR is composed of the following files:

- *.xml*: a file describing the network topology.
- *.bin*: a file which contains the weights and biases binary data.

There are different parameters that can be passed when calling the Model Optimizer. These are the most relevant:

- `input_shape`: to set up static shapes in case the model has dynamic input.
- `data_type`: convert all floating-point weights to another data type.
- `layout`: to specify the layout or change it (it can be *nhwc* for example).
- `mean_values`, `scale_values`, `scale`: to embed the corresponding preprocessing block for mean-value normalization of the input data and optimize it.
- `reverse_input_channels`: to embed color channel format reversing.
- `input`: to cut the model from the beginning and start at a specific layer.
- `output`: to cut the model at the end and finish with a specific layer.

In the Results section I will refer to this pipeline as the Baseline.

3.5.2 Post-training Optimization Tool

The Post-training Optimization Tool (POT) allowed me to do an additional optimization using its two quantization methods: the Default Quantization and the Accuracy-aware Quantization. In order to use the POT, I need an IR model, a representative calibration dataset representing a use case scenario and, in case of accuracy constraints, a validation dataset and accuracy metrics. I implemented two different pipelines, one for each quantization method. Their workflow can be described in four steps (the second one is for Accuracy-aware Quantization only):

1. Prepare the dataset interface using `openvino.tools.pot.DataLoader` base class. In particular it is necessary to override the `__len__()` method, which is responsible of returning the dataset size, and the `__getitem__()` method, which provides access by index, incorporates model-specific pre-processing and returns `(data, annotation)`. `data` is the input that is passed to the model at inference as `numpy.array` and `annotation` must be `None` for Default Quantization, while for Accuracy-aware Quantization it must have the format expected by the Metric class we will create.

2. Define the accuracy metric using `openvino.tools.pot.Metric` abstract class (Accuracy-aware only). Override the properties `value`, which returns the accuracy metric value for the last model output with format `Dict[str, numpy.array]`, `avg_value`, the average metric over collected results as `Dict[str, np.array]`, and `higher_better`, which should be `True` if higher metric means better performance or `False` if not. Override the methods `update(output, annotation)`, which computes and updates the metric value using the last model output and annotation, `reset()`, which collects the accuracy metric, and `get_attributes()`, which returns the dictionary `{metric_name: {attribute_name: value}}` and the required attributes are `direction` and `type`.

3. Select quantization parameters. The most important ones are `target_device`, which indicates the hardware to quantize for, `stat_subset_size`, which tells the size of data subset to calculate activations statistics used for quantization (no less than 300), and `maximal_drop` (Accuracy-aware only), which defines the maximum accuracy drop that has to be achieved after the quantization.

4. Define and run the quantization process. The steps are the following:

- load the model;
- initialize the engine for metric calculation and statistics collection;
- create a pipeline of compression algorithms and run it;
- (OPTIONAL) compress the model weights to quantized precision to reduce the size of the final *.bin* file;
- save the model to the desired path.

In order to use the optimized models you need to save the IR format and then run it using the OpenVINO runtime. The optional compression step can be very useful in context where we need to operate on low resources devices and we need to save memory space. In my experiments this functionality allowed me to obtain files whose size was the 30% of the original one.

3.6 The TensorRT engine generation process

The goal is now to optimize our models to run inference on NVIDIA GPUs. In particular, as explained in chapter 2, the target hardware is the Jetson Xavier. In order to convert a model from Protobuf to TensorRT, these are the steps I followed:

1. Convert the model to *.onnx*;
2. Generate a TensorRT engine using the `trtexec` command available in the Jetson.

Let's see these two steps in detail. Open Neural Network Exchange (ONNX) [15] is an open ecosystem that empowers AI developers providing an open source format for AI models. It defines an extensible computation graph model, as well as definitions of built-in operators and standard data types. In order to convert a model from Protobuf (*.pb*) to ONNX (*.onnx*) I have used the `tf2onnx` [16] library.

The conversion can be done using the `t2onnx.convert` and passing the following parameters: `output`, the name of the output file, `graphdef`, which is the Protobuf of the model you want to convert, `inputs` and `outputs`, that specify the TensorFlow model's input/output names, `opset`, the set of operators to use when mapping operations from TensorFlow to ONNX, `inputs-as-nchw`. This last parameter is really important: in order to generate the TensorRT engine, you need to pass an ONNX model with `nchw` format. Therefore if the original TensorFlow model is in the `nhwc` format,

it is necessary to use the `inputs-as-nchw` parameter to adapt it. Also, the `opset` parameter should be chosen carefully otherwise a loss of accuracy could be induced. Concerning the second step, `trtexec` is a command-line wrapper tool to use TensorRT without having to develop your own application. The `trtexec` tool has three main purposes:

- Benchmarking networks on random or user-provided input data;
- Generating serialized engines from models;
- Generating a serialized timing cache from the builder.

I exploited the first functionality with the goal of understanding which is the new inference time of the model and also I have used the generated engine both to integrate it in the original inference script and to use it in the DeepStream pipeline described in the next section.

3.7 The DeepStream pipeline

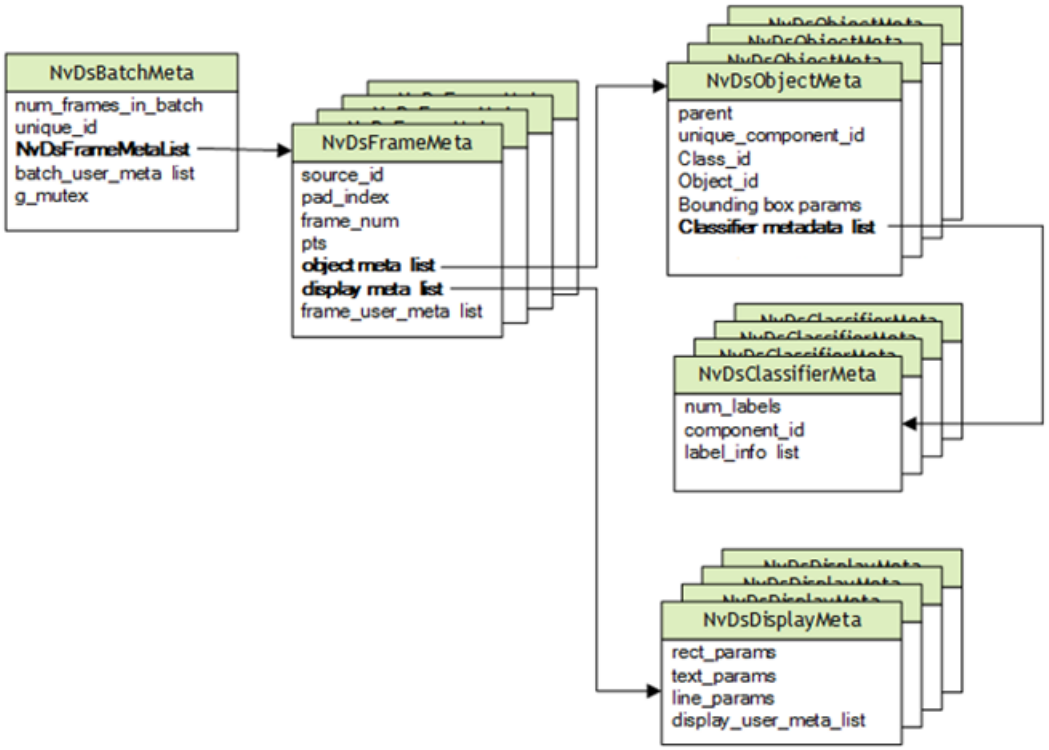
The development of the DeepStream pipeline has been the most challenging part of my thesis. The idea of AllRead was to create a prototype in order to demonstrate we can reach next level speed in terms of processing time. To reach this goal, I have decided to take the License Plate pipeline and try to replicate it using DeepStream. The main difference between the original LP pipeline and the DeepStream prototype is that the first one takes as input one frame at a time, while the second one can process a stream directly.

3.7.1 The metadata

In order to understand how DeepStream works, I have to first explain the metadata. Since DeepStream is built on GStreamer, the basic unit of data transfer is the Gst Buffer. Each Gst Buffer has associated metadata and the DeepStream SDK attaches the DeepStream metadata object, called `NvDsBatchMeta`.

More in detail, DeepStream uses an extensible standard structure for metadata: the basic metadata structure `NvDsBatchMeta` starts with batch level metadata, while subsidiary metadata structures hold frame, object, classifier, and label data. As you can see in Figure 3.4, the `NvDsBatchMeta` can contain different `NvDsFrameMeta` structures that keep the information of all the different frames in the batch. Again the `NvDsFrameMeta` can include different `NvDsObjectMeta` structures, responsible of transporting information about

Figure 3.4. DeepStream metadata overview.



detected objects in that frame (for example, the bounding boxes generated by the LP detector). After detection, if we want to perform also classification (for example applying the LP reader to the detected objects), the `NvDsObjectMeta` can be linked to an `NvDsClassifierMeta` structure that handles the result of classification on that specific detected object. Finally, the `NvDsFrameMeta` has a list of `NvDsDisplayMeta` structures, containing information about what to display.

3.7.2 The pipeline structure

Since DeepStream SDK is based on GStreamer framework, in order to build a pipeline we can use both DeepStream GStreamer plugins and standard GStreamer plugins. The most important object in GStreamer is the `GstElement` object. An element is the basic building block for a media pipeline and it needs to be encapsulated in a plugin in order to be used by GStreamer. However, a single plugin may contain the implementation of several elements, or just a single one. Plugins are only loaded when their provided elements are

requested. Elements can have one or two pads: the pads are the element's interface to the outside world. Data streams from one element's source pad to another element's sink pad. Finally, a `GstElement` can have several properties which are implemented using standard `GObject` properties. Most plugins provide additional properties to give more information about their configuration or to configure the element. There are three types of elements:

- Source elements: they don't accept data, they only generate data to be used by a pipeline.
- Filter elements: they have both input and outputs pads. They operate on data that they receive on their input (sink) pads, and they provide data on their output (source) pads.
- Sink elements: they accept data but do not produce anything. They are end points in a media pipeline.

Keeping these concepts in mind, we can now analyze how I implemented the DeepStream License Plate pipeline. The prototype code I developed is structured in this way:

1. Initialize GStreamer and create Pipeline element.
2. Create the different elements needed.
3. Define the properties of each element.
4. Add the elements to the pipeline and link them together.
5. Create an event loop and feed GStreamer bus messages to it.

Concerning this last step, a bus is a simple system that takes care of forwarding messages from the streaming threads to an application in its own thread context. Every pipeline contains a bus by default, so applications do not need to create a bus or anything. The only thing applications should do is to set a message handler on a bus, which is similar to a signal handler to an object. When the mainloop is running, the bus will periodically be checked for new messages, and the callback will be called when any message is available.

In Figure 3.5, you can see the architecture of the DeepStream License Plate pipeline. It takes as input a .H264 stream, it pre-processes the data, it applies the LP detector first and then it applies the LP reader on the output of the detector. It has the option of displaying the bounding boxes with

LP predictions, but I didn't use it since the AllRead's Jetson Xavier was in headless mode. More details about the different plugins are available in the next subsection.

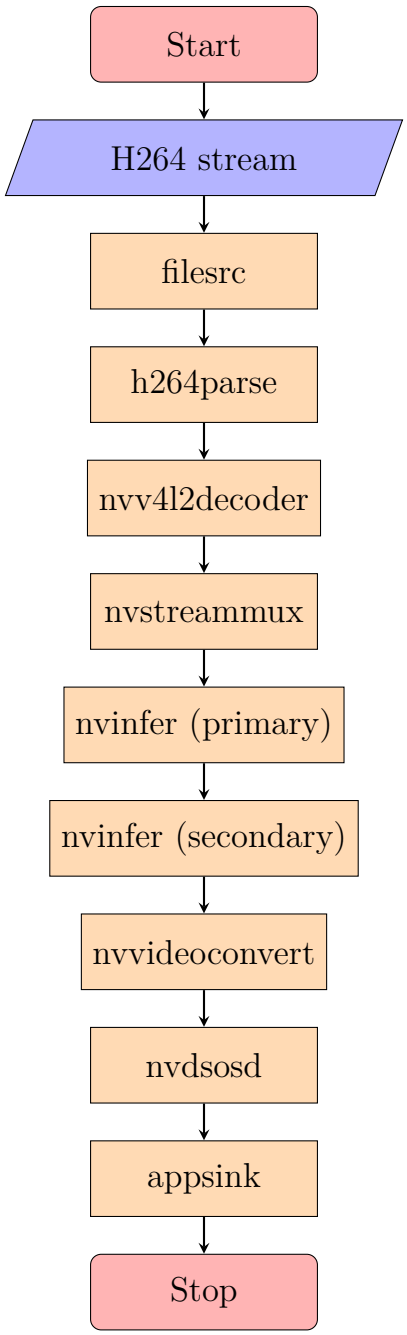


Figure 3.5. The DeepStream pipeline architecture.

3.7.3 Plugins description

These are the plugins I have used to implement the DeepStream License Plate pipeline:

- `filesrc`: source element responsible of reading data from a file in the local file system. You need to specify that file setting the property `location`.

- `h264parse`: it is used to parse H.264 streams.

- `nvv4l2decoder`: it leverages the hardware decoding engines on Jetson by interfacing with `libv4l2` plugins on that platform. The input is an encoded bitstream and the plugin uses the NVDEC hardware engine to decode that bitstream. Supported formats are H.264, H.265, JPEG and MJPEG. The output is a Gst Buffer with decoded output in NV12 format.

- `nvstreammux`: it forms a batched buffer of `batch-size` frames from multiple input sources and it pre-processes the input image.

  Even if I didn't use its first functionality since I had a single source, it is interesting to quickly explain what it can do. The muxer uses a round-robin algorithm to collect frames from the sources: it tries to collect an average of (batch-size/num-source) frames per batch from each source (if all sources are live and their frame rates are all the same). The number varies for each source, though, depending on the sources' frame rates. Then it pushes the batch downstream when the batch is filled, or the batch formation timeout `batched-pushed-timeout` is reached. The timeout starts running when the first buffer for a new batch is collected.

  Concerning the input pre-processing instead, the muxer outputs a single resolution (i.e. all frames in the batch have the same resolution). This resolution can be specified using the `width` and `height` properties. The muxer scales all input frames to this resolution. The `enable-padding` property can be set to true to preserve the input aspect ratio while scaling by padding with black bands.

- `nvinfer`: it infers on input data using NVIDIA TensorRT. Also, it handles part of the pre-processing: the pre-processing function is `y = net_scale_factor * (x - mean)`. The `Gst-nvinfer` plugin performs transforms (format conversion and scaling), on the input frame based

on network requirements, and passes the transformed data to the low-level library. The low-level library pre-processes the transformed frames (performs normalization and mean subtraction) and produces final float planar data which is passed to the TensorRT engine for inferencing.

The `Gst-nvinfer` plugin can work in three modes: in primary mode it operates on full frames, in secondary mode it operates on objects added in the meta by upstream components and in pre-processed tensor input mode it operates on tensors attached by upstream components. When operating in pre-processed tensor input mode, the pre-processing inside `Gst-nvinfer` is completely skipped. For the LP pipeline implementation we use nvinfer two times, one in primary mode to infer using the LP detector and one in secondary mode to infer with the LP detector.

Downstream components receive a Gst Buffer with unmodified contents plus the metadata created from the inference output of the `Gst-nvinfer` plugin. To create the metadata from the inference output, you should set the following properties: `custom-lib-path`, `output-tensor-meta=1`, `parse-classifier-func-name` for the reader model and `parse-bbox-func-name` for the detector.

In order to configure the plugin, you can set the property `config-file-path`, passing the path to a configuration file where you define the other properties of the plugin. The main generic properties to define are the following: `process-mode`, that is the mode (primary or secondary) in which the element is to operate on, `model-engine-file`, which is the pathname of the serialized model engine file, `batch-size`, it is the number of frames or objects to be inferred together in a batch, `gie-unique-id`, that is the unique ID to be assigned to the GIE (GPU Inference Engine) to enable the application and other elements to identify detected bounding boxes and labels.

For secondary inference you need to define `operate-on-gie-id`, which is the unique ID of the GIE on whose metadata (bounding boxes) this GIE is to operate on, `operate-on-class-ids`, that indicates the class IDs of the parent GIE on which this GIE is to operate on, `input-object-min-width`, `input-object-min-height`, `input-object-max-width` and `input-object-max-height`, used to define the minimum and maximum dimensions of the bounding boxes the secondary GIE has to operate on.

To apply the pre-processing function, these properties must be set up: `net-scale-factor`, that is the pixel normalization factor, `offsets`, that

represents the array of mean values of color components to be subtracted from each pixel. Consider that the array length must equal the number of color components in the frame and that the plugin multiplies mean values by `net-scale-factor`.

Finally to create metadata from the inference output, it is needed to define the properties I have previously described.

- `nvvideoconvert`: it performs video color format conversion. It accepts NVMM memory as well as RAW (memory allocated using `calloc()` or `malloc()`), and provides NVMM or RAW memory at the output.

- `nvdsosd`: it draws bounding boxes, text, and region of interest (RoI) polygons. The plugin takes as input an RGBA buffer with attached metadata from the upstream component. It draws bounding boxes, which may be shaded depending on the configuration (e.g. width, color, and opacity) of a given bounding box. It also draws text and RoI polygons at specified locations in the frame: the parameters are configurable through metadata.

- `appsink`: sink plugin that supports many different methods for making the application get a handle on the GStreamer data in a pipeline. Unlike most GStreamer elements, `Appsink` provides external API functions. Connecting the `new_sample` signal to a particular function, we can apply custom actions every time a new sample is available.

3.7.4 The C++ parsing functions

The goal of a parsing function is to post-process the output of a model and to attach the obtained objects to the metadata.

In the `nvinfer` plugin description we have seen that it is needed to define the `custom-lib-path` property: it is the absolute pathname of a library containing custom method implementations for custom models.

Inside this library I have implemented:

- `extern "C" bool NvDsInferParseCustomDetector( std::vector<NvDsInfer LayerInfo> const &outputLayersInfo, NvDsInferNetworkInfo const &networkInfo, NvDsInferParseDetectionParams const &detectionParams, std::vector<NvDsInferObjectDetectionInfo> $objectList)`: the detector parsing function. This function accesses the detector's output layer through the `outputLayersInfo` parameter and updates the `objectList`

with the detected bounding boxes. The `NvDsInferObjectDetectionInfo` structure is composed of ID of the class to which the object belongs, the vertical and horizontal offsets of the bounding box shape for the object, the width and height of the bounding box shape for the object and the object detection confidence.

- `extern "C" bool NvDsInferParseCustomReader( std::vector<NvDsInfer LayerInfo> const &outputLayersInfo, NvDsInferNetworkInfo const &networkInfo, float classifierThreshold, std::vector<NvDsInferAttribute> &attrList, std::string &descString)`: the reader parsing function. Also in this case it is possible to access the reader's output layer through the `outputLayersInfo` parameter. The goal is to populate the `attrList` parameter with the LP readings of the different objects. The `NvDsInferAttribute` structure holds information about one classified attribute and in particular it is composed of the index of the label, the output for the label, the confidence level for the classified attribute and the string label for the attribute. The `classifierThreshold` parameter can be used to filter out predictions with confidence under a certain threshold.

After implementing this library, I need to set the `parse-classifier-func-name` property for the reader model and the `parse-bbox-func-name` property for the detector. These properties contain the name of the custom bounding box/classifier output parsing function.

Chapter 4

Results

In this chapter I present the results obtained testing the different optimization pipelines on the three models running on different hardware. More importance has been given to the License Plate pipeline testing, since it is a model currently used in production by the company. The Damage detector optimization is only an additional experiment I did for this thesis.

The first section is about the metrics I have used to evaluate the optimized models. Then there is one section for each optimization framework.

4.1 Metrics

In the experiments we need two different kind of metrics:

- Quality metrics: the goal is to first assess that the optimized model is not reaching lower performances. In particular, the model after optimization should achieve the same quality metrics results it was obtaining before optimization. Depending on the different Computer Vision tasks, it is necessary to use different metrics for each model: Intersection over Union will be computed for the License Plate detector and for the Damage detector, while Accuracy is the one to evaluate the License Plate reader.

- Speed metrics: these are needed to assess the impact of the different optimizations. For each model I compute some statistics of the inference time, such as the mean and the standard deviation (usually in milliseconds ms). Obviously, if the optimization is working properly, the model should be faster in terms of inference time. When computing speed metrics for different models from different frameworks it's important to be coherent and try to start the clocks at the same time: including or not

including a specific part of the process in the inference time computation can change the result and mislead the final evaluation.

4.1.1 Intersection over Union

Intersection over Union (IoU) is a metric used in object detection to describe the extent of overlap of two boxes. In particular, given two bounding boxes, one with area A and the other one with area B,

$$IoU = \frac{A \cap B}{A \cup B}.$$

We compute this considering the bounding box associated with the Ground Truth (GT) and the one predicted by the model: the greater the region of overlap, the greater the IoU. I have used this IoU in order to evaluate the performance of the License Plate detector.

Concerning the Damage detector instead, I computed the IoU at pixel level: it's more or less the same, but we are not considering two bounding boxes, but two more general sets of points. I have done this because both the GT and the prediction are not boxes anymore, they are instead sets of points. Another solution could have been to obtain, from both the GT and the prediction, the enclosing bounding boxes and to use them to compute the IoU. The enclosing bounding box is the smallest box containing all the points of a set of points. Visualizing some of the enclosing bounding boxes, I realized it was too much inaccurate for some damage shapes. Therefore I decided to compute IoU using the sets of points directly.

Note that this metric is not as significant as accuracy for example: a small change in the IoU can be also introduced by a small error in the annotation process. During the annotation process there is a margin of error that we should consider. Of course the higher the quality of the training set (in terms of precision of the bounding boxes), the more reliable the IoU is.

4.1.2 Accuracy

Accuracy is the number of correctly predicted samples divided by the total number of samples. More formally, it is defined as the number of true positives and true negatives divided by the number of true positives, true negatives, false positives, and false negatives. I have used Accuracy to evaluate the performance of the License Plate reader: when one single digit or letter in the license plate is wrong, we will consider that reading as wrong. Therefore we will consider as good readings only the entirely right predictions.

4.1.3 Inference time statistics

In order to compute the inference time of a model I mainly use two methods. The first one is to run inference several times on the same image and calculate some statistics of the inference time: mean, standard deviation, minimum and maximum. To do this I have used the `timeit` module, in particular the `Timer` class, which times execution speed of small code snippets. The second way is applying the model on a set of samples: using the `time` library, I create a clock just before running inference on a pre-processed image and another one right after having generated the prediction. Then I keep summing the differences between the two clocks (they correspond to the different inference times) for all the samples in the set. At the end I divide by the number of samples and I obtain the mean time per iteration. It is important to use the same set of examples when comparing performances between two models.

When the goal is to compute the performance of the end-to-end License Plate pipeline, which consists of applying the LP detector first and then the LP reader, I use again the `time` module to create three different clocks. Two clocks to time the inference time of the two models and one clock to compute the total execution time: the last one involves not only the inference but also all the other steps of the workflow, such as image pre-processing and post-processing. As you will see later on in this chapter, DeepStream is the only optimization that affects the entire end-to-end pipeline rather than inference only.

4.2 Apache TVM

My first experiment was testing the TVM optimization pipeline on different standard architectures. As you see in Table 4.1, the results seem promising using the TVM runtime. This is a key point: on the TVM runtime, the optimized model is always faster than the unoptimized one. When I optimize a model that I currently have in production, I should compare its current execution time with the one of the optimized model. When I optimized both the LP detector and reader, I realized that the models running on the TensorFlow runtime were at the end faster than the optimized ones running on the TVM runtime. Therefore this optimization was not suitable for our models: at least it didn't lead to any improvement on my laptop Intel i5 CPU. In Table 4.2 you can see the performance of the License Plate detector on the different runtimes.

Model	Mean (ms)	Median (ms)	Max (ms)	Min (ms)	Std (ms)
Alexnet	112.45	106.54	260.65	92.38	19.04
Alexnet-OPT	57.16	55.73	64.90	53.25	3.95
Inception v2	158.72	157.24	172.35	156.39	4.57
Inception v2-OPT	118.02	117.50	120.50	117.27	1.12
Resnet 18	154.45	150.98	189.70	135.38	17.93
Resnet 18-OPT	98.44	98.39	99.03	98.31	0.20
Resnet 50	325.95	324.57	335.16	323.23	3.33
Resnet 50-OPT	227.69	227.54	230.20	226.96	0.88
Resnet 101	655.36	648.70	692.52	639.17	17.73
Resnet 101-OPT	467.00	464.49	488.38	459.98	8.49
Densenet	243.50	242.98	246.35	242.63	1.18
Densenet-OPT	170.37	170.33	170.63	170.23	0.12
Googlenet	181.84	174.87	250.09	138.03	33.47
Googlenet-OPT	97.06	96.67	100.69	96.43	1.21

Table 4.1. TVM optimization results on Intel i5 CPU. The model without OPT is just loaded to the TVM runtime. When there is OPT in the name, it means the model has been optimized and then tested on the TVM runtime.

Model	Mean (ms)	Std (ms)
TensorFlow	59.08	1.09
TVM - unoptimized	363.40	3.17
TVM - optimized	79.44	0.55

Table 4.2. LP detector inference performances on different runtimes. Both the TVM models are run on the TVM runtime.

4.3 OpenVINO

OpenVINO (OV) was the object of several experiments. I have tested four different pipelines:

- Baseline: it consists of applying the Model Optimizer without changing the data type.

- FP16: it exploits the Model Optimizer functionality of changing data type, coverting all the weights to Floating Point 16.

- Quantization: Post-training Optimization Tool (POT)'s Default Quantization.
- Accuracy-aware Quantization: POT's Accuracy-aware Quantization.

Model	Mean (ms)	Std (ms)	IoU
TensorFlow	59.08	1.09	0.707561
Baseline	27.14	0.57	0.707561
FP16	26.98	0.56	0.707559
Quantization	22.07	0.54	0.690587
Accuracy-aware Quantization	22.47	1.06	0.701180

Table 4.3. LP detector inference performances on Intel Core i5 dual-core using different optimization methods. Baseline is the model just converted to the IR format and run on the OpenVINO runtime. FP16 is the model converted to FP16 precision (instead of FP32 which is the original one). Quantization is the model quantized without any constrain on Intersection over Union loss. Finally, Accuracy-aware quantization is the quantized model with a constrain on the loss of Intersection over Union.

Model	Mean (ms)	Std (ms)	IoU
TensorFlow	42.47	1.91	0.707561
Baseline	13.56	0.81	0.707561
FP16	11.78	0.30	0.707559
Quantization	8.38	0.23	0.688902
Accuracy-aware quantization	10.83	0.42	0.718029

Table 4.4. LP detector inference performances on Intel Core i7 - 8650U using different optimization methods. Baseline is the model just converted to the IR format and run on the OpenVINO runtime. FP16 is the model converted to FP16 precision (instead of FP32 which is the original one). Quantization is the model quantized without any constrain on Intersection over Union loss. Finally, Accuracy-aware quantization is the quantized model with a constrain on the loss of Intersection over Union.

I tested these different pipelines on the License Plate detector running on Intel Core i5 dual-core in Table 4.3 and on Intel Core i7 - 8650U in Table 4.4. As you can see, the FP16 optimization doesn't improve a lot the performances with respect to the Baseline. Quantization pipelines instead speed

up the model on both machines: in particular with Default Quantization we reach the best performance in terms of speed metrics, but we lose some points in Intersection over Union (IoU). Probably with Accuracy-aware Quantization we obtain the best trade-off between inference time and IoU: few IoU points can also be not so significant, but at the end the Default Quantization is not so faster than the Accuracy-aware one. Therefore Accuracy-aware Quantization produces an optimized LP detector which is 2.62x faster on i5 CPU and 3.92x faster on i7 CPU.

Model	**Mean (ms)**	**Std (ms)**	**LP ACC**	**Country ACC**
TensorFlow	79.30	2.90	0.9589	0.9866
Baseline	52.93	0.63	0.9589	0.9866
FP16	51.39	0.68	0.9589	0.9866
Quantization	48.33	0.64	0.8952	0.9733
Accuracy-aware quantization	47.95	0.89	0.9527	0.9866

Table 4.5. LP reader inference performances on Intel Core i5 dual-core using different optimization methods. Baseline is the model just converted to the IR format and run on the OpenVINO runtime. FP16 is the model converted to FP16 precision (instead of FP32 which is the original one). Quantization is the model quantized without any constrain on LP accuracy loss. Finally, Accuracy-aware quantization is the quantized model with a constrain on the loss of Intersection over Union.

Model	**Mean (ms)**	**Std (ms)**	**LP ACC**	**Country ACC**
TensorFlow	47.36	2.18	0.9589	0.9866
Baseline	24.73	0.55	0.9589	0.9866
FP16	23.95	0.36	0.9589	0.9866
Quantization	18.22	0.75	0.9044	0.9774
Accuracy-aware quantization	18.39	1.49	0.9527	0.9866

Table 4.6. LP reader inference performances on Intel Core i7 - 8650U using different optimization methods. Baseline is the model just converted to the IR format and run on the OpenVINO runtime. FP16 is the model converted to FP16 precision (instead of FP32 which is the original one). Quantization is the model quantized without any constrain on LP accuracy loss. Finally, Accuracy-aware quantization is the quantized model with a constrain on the loss of Intersection over Union.

Then I optimized the License Plate reader on both Intel Core i5 dual-core

in Table 4.5 and Intel Core i7 - 8650U in Table 4.6. The LP reader's behaviour is similar to the detector's one: FP16 doesn't improve so much with respect to the Baseline and Default Quantization leads to the best performance in terms of inference time on i7 CPU. On i5 CPU the best optimization pipeline is for sure the Accuracy-aware Quantization, since it is the fastest and it is limiting the loss of accuracy to a minimum quantity. Also on i7 CPU it is pretty clear that the Accuracy-aware Quantization is the best optimization pipeline: it is around 0.17 milliseconds slower but it has almost 5 accuracy points more than the Default one. In this case we have seen the utility of the constraint on accuracy loss.

Models	**Total time (s)**	**Det. time (ms/it)**	**Read. time (ms/it)**
TensorFlow	73	63.2	79.1
OV Baseline	53	27	50.8
OV Acc.-aware Quant.	50	21.4	46.3

Table 4.7. LP pipeline end-to-end performances on Intel Core i5 in terms of total time (seconds) to process 298 images, detection time (milliseconds per iteration) and reading time (milliseconds per iteration). TensorFlow means both the detector and reader are running on TensorFlow runtime, OV Baseline means both models are just converted to the OpenVINO format and no further optimization is performed, while OV Acc.-aware Quant. means the models are converted to OpenVINO format and Accuracy-aware quantization is performed.

Finally I tested the different optimization pipeline applying them on the end-to-end License Plate pipeline running on Intel Core i5 CPU. The results are available in Table 4.7. The Accuracy-aware Quantization pipeline is 2.10x faster than the TensorFlow one in terms of total inference time. On the other hand, it is only 1.46x faster in terms of total processing time: this is due to the fact that inference is only one of the steps of an end-to-end pipeline.

In order to conduct a more exhaustive evaluation of the different pipelines and test also a model belonging to another CV task, I performed the same experiment with the Damage detector model running on Intel i5. The results can be seen in Table 4.8: the behaviour is more or less the same we experience with the LP pipeline models.

The conclusion is that with OpenVINO we improve significantly the inference time of our models. The main limitation is that this framework is only designed to operate on Intel products.

Model	Mean (ms)	Std (ms)	IoU
TensorFlow	67.13	1.51	0.525833
Baseline	40.44	0.63	0.525833
FP16	40.27	0.61	0.525812
Quantization	35.89	0.59	0.511129
Accuracy-aware Quantization	36.21	0.73	0.525176

Table 4.8. Damage detector inference performances on Intel Core i5 dual–core using different optimization methods. Baseline is the model just converted to the IR format and run on the OpenVINO runtime. FP16 is the model converted to FP16 precision (instead of FP32 which is the original one). Quantization is the model quantized without any constrain on Intersection over Union loss. Finally, Accuracy-aware quantization is the quantized model with a constrain on the loss of Intersection over Union.

4.4 TensorRT

In this section you can see the results of the comparison between the OpenVINO optimization on CPUs and the TensorRT one on GPU. The CPUs are once again the Intel Core i5 and the Intel Core i7, while the GPU is the Jetson Xavier.

Model - Machine	Mean execution time (ms)
TensorFlow - i5	59.08
OpenVINO - i5	22.47
TensorFlow - i7	42.47
OpenVINO - i7	10.83
TensorRT - Jetson	5.16

Table 4.9. LP detector performances comparison between TensorRT optimization on NVIDIA Jetson Xavier and the previous ones.

The License Plate detector performances are available in Table 4.9 and the LP reader ones can be seen in Table 4.10.

Also in this case, I have tested the Quality metrics of the optimized models and they correspond to the ones obtained by the original models. The TensorRT detector IoU is the same as the TF model and the execution time on Jetson is more than 11x faster than on i5. Also the reader improves runtime speed without losing accuracy: the optimized model infers on Jetson more

Model - Machine	Mean execution time (ms)
TensorFlow - i5	79.3
OpenVINO - i5	47.95
TensorFlow - i7	47.36
OpenVINO - i7	18.39
TensorRT - Jetson	8.45

Table 4.10. LP reader performances comparison between TensorRT optimization on NVIDIA Jetson Xavier and the previous ones.

than 9x faster than on i5. For OpenVINO we are considering accuracy-aware quantized models: the TensorRT optimized models are more than 2x faster than the OpenVINO optimized ones.

What emerges from these results is that it is definitely worth of investing in an edge device like the Jetson Xavier if you are able to optimize your models to run on it. At least you will reach a significant improvement of the inference time speed.

4.5 DeepStream

Finally I present the performance that can be reached using DeepStream optimization. It is important to remind that with DeepStream we optimize the entire stream of data. TensorFlow and OpenVINO metrics are collected on i5, while DeepStream ones are obtained running on Jetson Xavier. The inference time is optimized using TensorRT.

You will see in the results tables that with OpenVINO we optimize inference making the models run up to 2.62x faster on i5. However the impact is reduced if we take into consideration the total processing time: OpenVINO optimizes only the inference time and it doesn't influence the other steps such as pre-processing and post-processing.

In Table 4.11 we can see the huge impact of DeepStream optimization on the License Plate detector. The DeepStream single stage pipeline involving the LP detector only is 8.34x faster than the TensorFlow one. And in this case we are not talking about inference time only: we are referring to the total processing time. Inference time is optimized exploiting TensorRT engines, therefore we reach the performances I presented in the previous section.

The impact on the License Plate reader is similar, as you can see in Table 4.12. The DeepStream single stage pipeline involving the LP reader only is

Model	Inference Time (ms/it)	Processing Time (ms/it)
LP detector - TensorFlow	59.08	153.7
LP detector - OpenVINO	22.47	115.1
LP detector - DeepStream	5.16	18.42

Table 4.11. Performances of the DeepStream single stage pipeline involving the LP detector only. Comparison with TensorFlow and OpenVINO ones.

8.10x faster than the TensorFlow one.

Model	Inference Time (ms/it)	Processing Time (ms/it)
LP reader - TensorFlow	79.3	91.66
LP reader - OpenVINO	47.95	67.16
LP reader - DeepStream	8.45	11.32

Table 4.12. Performances of the DeepStream single stage pipeline involving the LP reader only. Comparison with TensorFlow and OpenVINO ones.

Finally I have tested the end-to-end License Plate pipeline performances and they reflect the improvements we have seen for the single-stage ones: as you can see in Table 4.13, with DeepStream the LP pipeline is 8.30x faster than with TensorFlow.

Moreover you can observe that the total processing time is now of 32.15 milliseconds per iteration, about 31.10 frames per second: this means that we can reach real-time inference. In this way it's proven that it is worth investing in an edge device such as the NVIDIA Jetson Xavier: with this hardware and DeepStream, it is possible to reach significant improvements in the total processing time.

Since we are using TensorRT for inference, there is no loss of accuracy.

Note that the Jetson Xavier is a good hardware among edge devices, but NVIDIA offers several more powerful GPUs that can improve performances even more: on premise and on cloud an higher speed can be reached.

Model	Processing Time (ms/it)
E2E - TensorFlow	266.85
E2E - OpenVINO	181.54
E2E - DeepStream	32.15

Table 4.13. Performance of the DeepStream end-to-end LP pipeline. Comparison with TensorFlow and OpenVINO ones.

Chapter 5

Conclusion

In this thesis I explored the following optimization frameworks: Apache TVM, OpenVINO, TensorRT and DeepStream. For each of them, I implemented and tested different optimization pipelines, exploiting their main functionalities.

The first approach based on Apache TVM was promising: results were good on standard architectures and the tool covers a wide range of hardware we can optimize our models for. Anyway, the AllRead's models performances were quite bad when comparing the TVM runtime with the original TF one on Intel i5. In particular the optimization of the LP pipeline's models didn't lead to any improvement.

Since Intel CPUs are the most common hardware among AllRead's clients, OpenVINO was defined as the next optimization framework to try. This tool definitely improved all the AllRead's models I tested. The performances of four different OpenVINO optimization pipelines were compared and the License Plate detector reached the best results: the optimized LP detector is 2.62x faster on Intel i5 and 3.92x faster on Intel i7 in terms of inference time.

Edge devices are a solution that is getting more and more popular in the last years and the GPUs offered by NVIDIA are valid hardware in this context. Keeping this in mind we explored and tested TensorRT. The LP pipeline models optimized with this framework and run on NVIDIA Jetson Xavier are reaching the best performance in terms of inference time.

Inference is only one of the stages that compose an end-to-end streaming analytics pipeline: therefore optimizing the inference time will affect the total processing time only partially. A clear example is the OpenVINO one. With this tool we optimized inference making the models run up to 2.62x faster

on i5. However the impact is reduced if we take into consideration the total processing time.

To tackle this aspect, we decided to use DeepStream to optimize the entire stream of data while taking advantage of TensorRT inference. The results obtained with this SDK are significant: we can reach real time inference on Jetson Xavier, with a speed of 31.10 frames per second.

This thesis demonstrates that a hardware specific optimization of the neural networks that a company uses in production can be a game changer in the industrial context. It allows to deploy models that are significantly faster, keeping accuracy unchanged. This solution can be used both to improve the product and to cut hardware costs.

Bibliography

[1] Grand View Research, Artificial Intelligence Market Size, Share & Trends Analysis Report By Solution, By Technology (Deep Learning, Machine Learning, Natural Language Processing, Machine Vision), By End Use, By Region, And Segment Forecasts, 2022 - 2030, 2022

[2] Daniel Zhang et al., The AI Index 2022 Annual Report, 2022

[3] Grand View Research, Optical Character Recognition Market Size, Share & Trends Analysis Report By Type (Software, Services), By Vertical (BFSI, Transport & Logistics), By End Use (B2B, B2C), By Region, And Segment Forecasts, 2022 - 2030, 2022

[4] Max Jaderberg et al., Reading Text in the Wild with Convolutional Neural Networks, 2014

[5] Lluìs Gòmez et al., Single Shot Scene Text Retrieval, 2018

[6] Marçal Rusiñol et al., Automatic Structured Text Reading for License Plates and Utility Meters, 2019

[7] Martìn Abadi et al., TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems, 2015

[8] T. Chen et al., TVM: An Automated End-to-End Optimizing Compiler for Deep Learning, 2018

[9] J. Roesch et al., Relay: A New IR for Machine Learning Frameworks, 2018

[10] Intel, https://docs.openvino.ai/latest/index.html

[11] NVIDIA, DeepStream SDK, https://developer.nvidia.com/deepstream-sdk

[12] A. G. Howard et al., MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications, 2017

[13] Olaf Ronneberger, Philipp Fischer, Thomas Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation, 2015

[14] Kentaro Wada, https://github.com/wkentaro/labelme

[15] Open Neural Network Exchange (ONNX),

https://github.com/onnx/onnx
[16] tf2onnx - Convert TensorFlow, Keras, Tensorflow.js and Tflite models to ONNX, https://github.com/onnx/tensorflow-onnx