Intelligent Traffic System [1]

Pham Thanh Phong
Founder
ITM Vision
phongpham663@gmail.com

Ha Quang Phuoc
AI Engineer
ITM Vision
hqphuoc129@gmail.com

March 3, 2021

[1]Special thanks for the guidance of Msc.Vo Phi Son

CONTENTS

List of Figures

Chapter 1

OVERVIEW ITS SYSTEM

1.1 ITS

1.2 Jetson Platform

1.2.1 Introduction

NVIDIA ®Jetson ™is the world's leading platform for AI at the edge. Its high-performance, low-power computing for **deep learning** and computer vision makes it the ideal platform for compute-intensive projects. The Jetson platform includes a variety of Jetson modules together with NVIDIA JetPack ™SDK.

Each **Jetson module** is a computing system packaged as a plug-in unit (System on Module). NVIDIA offers a variety of Jetson modules with different capabilities.

JetPack bundles all of the Jetson platform software, starting with the NVIDIA ®Jetson ™Linux Driver Package (L4T).L4T provides the Linux kernel, bootloader, NVIDIA drivers, flashing utilities, sample filesystem, and more for the Jetson platform.

1.2.2 Jetson Developer Kits and Modules

Jetson Developer Kits include a non-production specification Jetson module attached to a reference carrier board. Together with **JetPack SDK**, it is used to develop and test software for your use case. Jetson developer kits are not intended for production use.

Jetson modules are suitable for deployment in a production environment throughout their operating lifetime. Each Jetson module ships with no software pre-installed; you attach it to a carrier board designed or procured for your end product, and flash it with the software image you've developed.

1.2.3 Jetson Nano

NVIDIA announced the **Jetson Nano Developer Kit** at the 2019 NVIDIA GPU Technology Conference (GTC) available now for embedded designers, researchers, and DIY makers, delivering the power of modern AI in a compact, easy-to-use platform with full software programmability. Jetson Nano delivers 472 GFLOPS of compute performance with a quad-core 64-bit ARM CPU and a 128-core integrated NVIDIA GPU. It also includes 4GB LPDDR4 memory in an efficient, low-power package with 5W/10W power modes and 5V DC input.

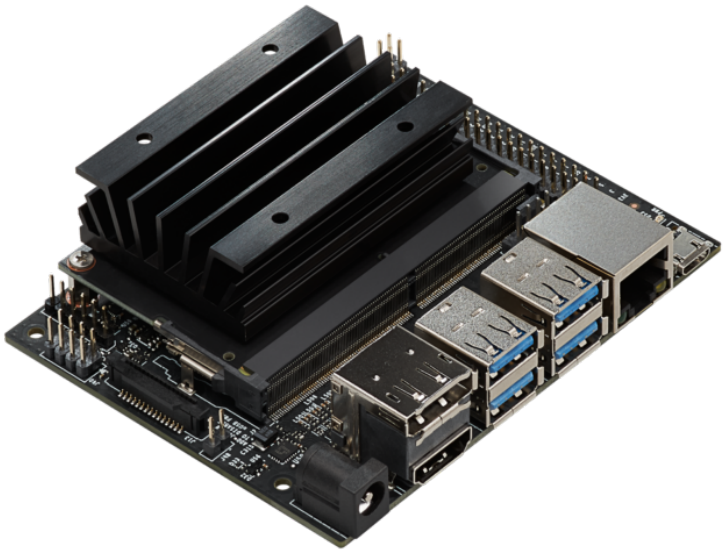

Figure 1.1: Jetson Developer Kit (80x100mm)

They newly released **Jetpack 4.2 SDK** provides a complete desktop Linux environment for Jetson Nano based on Ubuntu 18.04 with accelerated graphics, support for NVIDIA CUDA Toolkit 10.0, and libraries such as cuDNN 7.3 and TensorRT 5.The SDK also includes the ability to natively install popular open source Machine Learning (ML) frameworks such as TensorFlow, PyTorch, Caffe, Keras, and MXNet, along with frameworks for computer vision and robotics development like OpenCV and ROS.

Full compatibility with these frameworks and NVIDIA's leading AI platform makes it easier than ever to deploy AI-based inference workloads to Jetson. Jetson Nano brings real-time computer vision and inferencing across a wide variety of complex Deep Neural Network (DNN) models.

Processing	
CPU	64-bit Quad-core ARM A57 @ 1.43GHz
GPU	128-core NVIDIA Maxwell @ 921MHz
Memory	4GB 64-bit LPDDR4 @ 1600MHz | 25.6 GB/s
Video Encoder*	4Kp30 | (4x) 1080p30 | (2x) 1080p60
Video Decoder*	4Kp60 | (2x) 4Kp30 | (8x) 1080p30 | (4x) 1080p60
Interfaces	
USB	4x USB 3.0 A (Host) | USB 2.0 Micro B (Device)
Camera	MIPI CSI-2 x2 (15-position Flex Connector)
Display	HDMI | DisplayPort
Networking	Gigabit Ethernet (RJ45)
Wireless	M.2 Key-E with PCIe x1
Storage	MicroSD card (16GB UHS-1 recommended minimum)
Other I/O	(3x) I2C | (2x) SPI | UART | I2S | GPIOs

Figure 1.2: Jetson Nano Developer Kit technical specifications

The devkit is built around a 260-pin SODIMM-style System-on-Module (SoM). The SoM contains the processor, memory, and power management circuitry. The production compute module will include 16GB eMMC onboard storage and enhanced I/O with PCIe Gen2 x4/x2/x1, MIPI DSI, additional

GPIO, and 12 lanes of MIPI CSI-2 for connecting up to three x4 cameras or up to four cameras in x4/x2 configurations. Jetson's unified memory subsystem, which is shared between CPU, GPU, and multimedia engines, provides streamlined ZeroCopy sensor ingest and efficient processing pipelines.

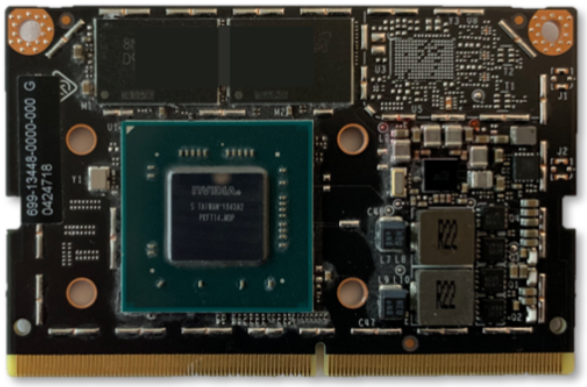

Figure 1.3: 45x70mm Jetson Nano compute module with 260-pin edge connector

1.2.3.1 Deep Learning Inference Benchmarks

Jetson Nano can run a wide variety of advanced networks, including the full native versions of popular ML frameworks like TensorFlow, PyTorch, Caffe/Caffe2, Keras, MXNet, and others. These networks can be used to build autonomous machines and complex AI systems by implementing robust capabilities such as image recognition, object detection and localization, pose estimation, semantic segmentation, video enhancement, and intelligent analytics.

The inferencing used batch size 1 and FP16 precision, employing NVIDIA's **TensorRT** accelerator library included with JetPack 4.2. Jetson Nano attains real-time performance in many scenarios and is capable of processing multiple high-definition video streams.

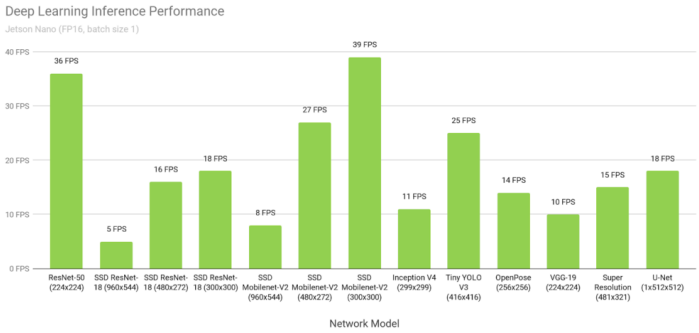

Figure 1.4: Performance of various deep learning inference networks with Jetson Nano and TensorRT

1.2.3.2 Multi-Stream Video Analytics

Jetson Nano processes up to eight HD full-motion video streams in real-time and can be deployed as a low-power edge intelligent video analytics platform for Network Video Recorders (NVR), smart cameras, and IoT gateways. NVIDIA's **DeepStream SDK** optimizes the end-to-end inferencing pipeline with ZeroCopy and TensorRT to achieve ultimate performance at the edge and for on-premises servers. The video below shows Jetson Nano performing object detection on eight 1080p30 streams simultaneously with a ResNet-based model running at full resolution and a throughput of 500 megapixels per second (MP/s).

Figure 1.5: DeepStream application running on Jetson Nano with ResNet-based object detector concurrently on eight independent 1080p30 video streams

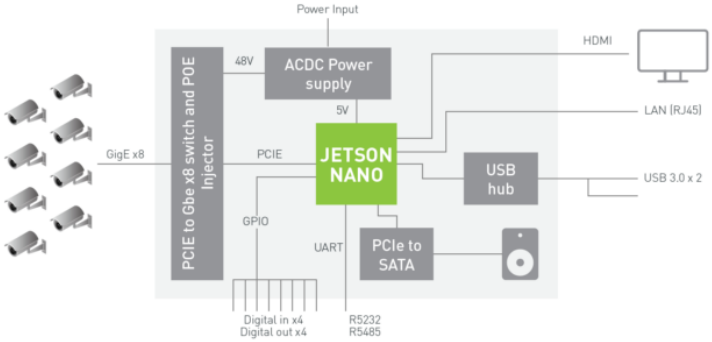

Figure 1.6: Reference NVR system architecture with Jetson Nano and 8x HD camera inputs

This figure shows an example NVR architecture using Jetson Nano for ingesting and processing up to eight digital streams over Gigabit Ethernet with deep learning analytics. The system can decode 500 MP/s of H.264/H.265 and encode 250 MP/s of H.264/H.265 video.

1.2.4 Jetson TX1

It's a small form-factor Linux system-on-module, destined for demanding embedded applications in visual computing. Designed for developers and makers everywhere, the miniature Jetson TX1 deploys teraflop-level supercomputing performance onboard platforms in the field. Backed by the Jetson TX1 Developer Kit, a premier developer community, and a software ecosystem including Jetpack, Linux For Tegra R23.1, CUDA Toolkit 7, cuDNN, and VisionWorks.

Jetson TX1's credit-card footprint and low power consumption mean that it's geared for deployment onboard embedded systems with constrained size, weight, and power (SWaP). Jetson TX1 exceeds the performance of Intel's high-end Core i7-6700K Skylake in deep learning classification with Caffe, and while drawing only a fraction of the power, achieves more than ten times the perf-per-watt.

1.2.4.1 Jetson TX1 Module

Built around NVIDIA's 20nm Tegra X1 SoC featuring the 1024-GFLOP Maxwell GPU, 64-bit quad-core ARM Cortex-A57, and hardware H.265 encoder/decoder, Jetson TX1 measures in at 50x87mm and is packed with performance and functionality. Onboard components include 4GB LPDDR4, 16GB eMMC flash, 802.11ac WiFi, Bluetooth 4.0, Gigabit Ethernet, and accepts 5.5V-19.6VDC input. Peripheral interfaces consist of up to six MIPI CSI-2 cameras (on a dual ISP), 2x USB 3.0, 3x USB 2.0, PCIe gen2 x4 + x1, independent HDMI 2.0/DP 1.2 and DSI/eDP 1.4, 3x SPI, 4x I2C, 3x UART, SATA, GPIO, and others. Needless to say, Jetson TX1 stands tall in the face of many an algorithmic and integration challenge.

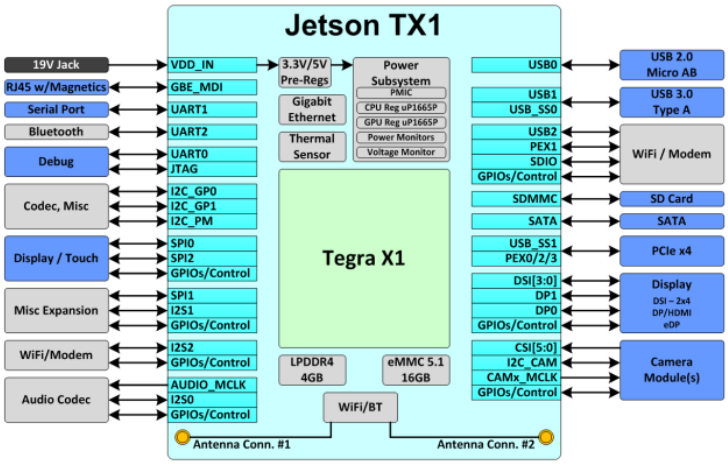

Figure 1.7: Jetson TX1 block diagram

The Jetson module utilizes a 400-pin board-to-board connector for interfacing with the Developer Kit's reference carrier board, or with a bespoke, customized board designed during your productization process. Tegra's chip-level capabilities and I/O are closely mapped to the module's pin-out. The pin-out will be backward-compatible with future versions of the Jetson module. Jetson TX1 comes with an integrated thermal transfer plate, rated between -25°C and 80°C, for interfacing with passive or active cooling solutions. Consult NVIDIA's Embedded Developer Zone for thorough documentation and detailed electromechanical specifications, in addition to visiting the active and open development community on Devtalk.

Figure 1.8: Left to right: Top of Jetson TX1 module, bottom (with connector), and complete assembly with TTP

Jetson TX1 draws as little as 1 watt of power or lower while idle, around 8-10 watts under typical CUDA load, and up to 15 watts TDP when the module is fully utilized, for example during gameplay and the most demanding vision routines. Jetson TX1 provides exceptional dynamic power scaling either based on workload via its automated governor, or by explicit user commands to gate cores and specify clock frequencies. The four ARM A57 cores automatically scale between 102 MHz and 1.9 GHz, the memory controller between 40MHz and 1.6GHz, and the Maxwell GPU between 76 MHz and 998 MHz. Touting 256 CUDA cores with Compute Capability 5.3 and Dynamic Parallelism, Jetson TX1's Maxwell GPU is rated for up to 1024 GFLOPS of FP16. When combined with support for up to 1200 megapixels/sec from either three MIPI CSI x4 cameras or six CSI x2 cameras, along with hardware H.265 encoder & decoder, integrated WiFi and HDMI 2.0, Jetson TX1 is primed for all-4K video processing.

1.2.4.2 Jetson TX1 Developer Kit

The Jetson TX1 Developer Kit contains a reference mini-ITX carrier board, 5MP MIPI CSI-2 camera module, two 2.4/5GHz antennas, an active heatsink & fan, an acrylic base plate, and a 19VDC power supply brick.

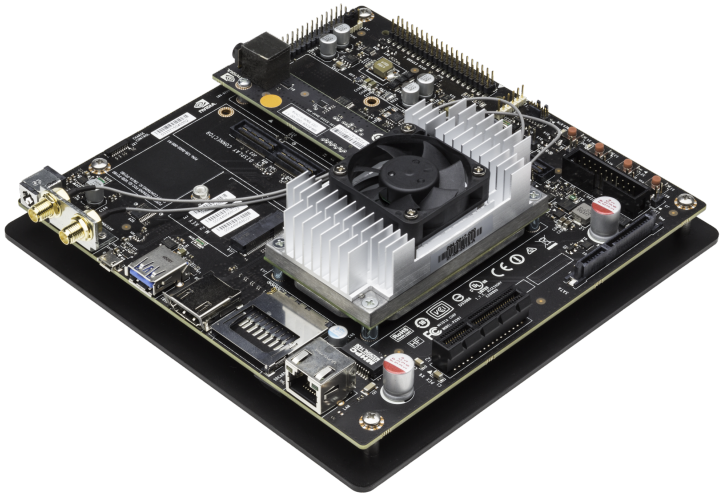

Figure 1.9: Jetson TX1 Developer Kit, including module, reference carrier and camera board

The PCIe lanes on the Jetson TK1 Developer Kit are routed from the module to a PCIe x4 desktop slot on the carrier for easy prototyping, in addition to an M.2-E mezzanine with PCIe x1 for wireless radios. Available on the Embedded Developer Zone, NVIDIA shares the schematics and design files for the reference carrier along with the 5MP CSI-2 camera module, including routing and signal integrity guidelines. Board software support bundled by Jetpack provides easy flashing and device configuration. Out of the box, the Jetson TX1 Developer Kit provides the experience of a desktop PC, but in a small embedded form factor that only draws a fraction of the power.

1.2.4.3 Vision Works

Jetson TX1 marks the first release of VisionWorks available to developers through Jetpack 2.0 and the **Embedded Developer Zone**. VisionWorks provides primitives and building blocks that are highly optimized for Tegra using tuned CUDA kernels.

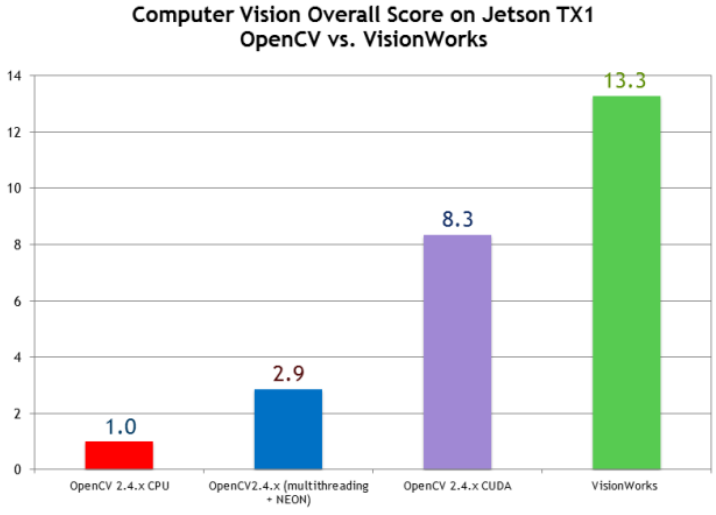

Figure 1.10: Benchmarks demonstrate the large speedup of VisionWorks vs. OpenCV running on the Jetson TX1 CPU and GPU

VisionWorks is more than 10x faster than upstream CPU-only OpenCV, is 4.5x faster than OpenCV4Tegra with NEON extensions, and is 1.6x faster than OpenCV's GPU module. The Overall Computer Vision Score was collected from the geometric mean performance of all the overlapping primitives between OpenCV and VisionWorks. Each primitive was measured across image sizes 720p and larger, and across all permutations of argument parameters.

Figure 1.11: Jetson taps into the NVIDIA ecosystem to deliver unprecedented scalability and developer-friendly support

In addition to more than 50 filtering, warping, and image-enhancement primitives, VisionWorks also offers numerous higher-level building blocks as well, such as LK optical flow, stereo block-matching (SBM), Hough lines & circles, and Harris (Corner) feature-detection & tracking. VisionWorks provides a full implementation of OpenVX 1.1. Developers can leverage VisionWorks to deploy camera-ready algorithms and vision pipelines, already tuned for Jetson.

1.2.4.4 Development Platforms

The NVIDIA Jetson ecosystem is rich with tools and support for enabling your research and development of applications and products with Jetson TX1. In the larger scheme, NVIDIA software toolkits for accelerated computing, deep learning, computer vision, and graphics are portable from the datacenter to the workstation to embedded SoC, allowing enterprise users to seamlessly scale and deploy their applications to devices in the field. Using Jetson, developers can leverage NVIDIA's shared architecture and power-efficient technology to roll out high-performance embedded systems with ease and flexibility.

1.2.5 Jetson TX2

Jetson is the world's leading low-power embedded platform, enabling server-class AI compute performance for edge devices everywhere. Jetson TX2 features an integrated 256-core NVIDIA Pascal GPU, a hex-core ARMv8 64-bit CPU complex, and 8GB of LPDDR4 memory with a 128-bit interface. The CPU complex combines a dual-core NVIDIA Denver 2 alongside a quad-core Arm Cortex-A57. The Jetson TX2 module fits a small Size, Weight, and Power (SWaP) footprint of 50 x 87 mm, 85 grams, and 7.5 watts of typical energy usage.

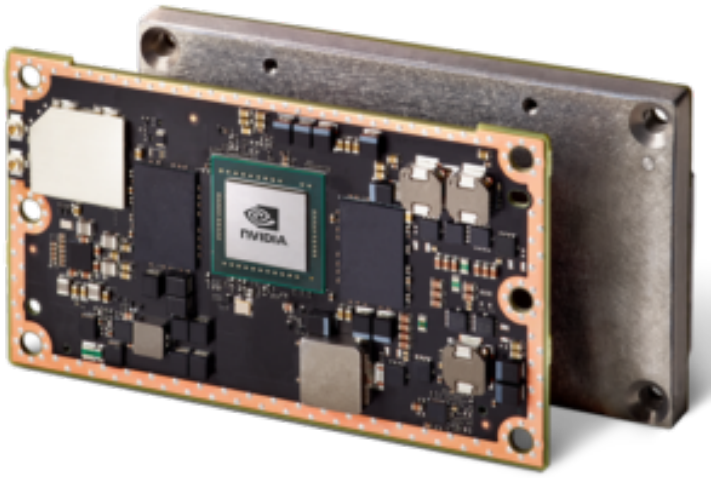

Figure 1.12: NVIDIA Jetson TX2 embedded system-on-module with Thermal Transfer Plate (TTP)

Internet-of-Things (IoT) devices typically function as simple gateways for relaying data. They rely on cloud connectivity to perform their heavy lifting and number-crunching. Edge computing is an emerging paradigm which uses local computing to enable analytics at the source of the data. With more than a TFLOP/s of performance, Jetson TX2 is ideal for deploying advanced AI to remote field locations with poor or expensive internet connectivity. Jetson TX2 also offers near-real-time responsiveness and minimal latency—key for intelligent machines that need mission-critical autonomy. Jetson TX2 is based on the 16nm NVIDIA Tegra "Parker" system on a chip (SoC). Jetson TX2 is twice as energy efficient for **Deep Learning** inference than its predecessor, Jetson TX1, and offers higher performance than an Intel Xeon Server CPU. This jump in efficiency redefines possibilities for extending advanced AI from the cloud to the edge.

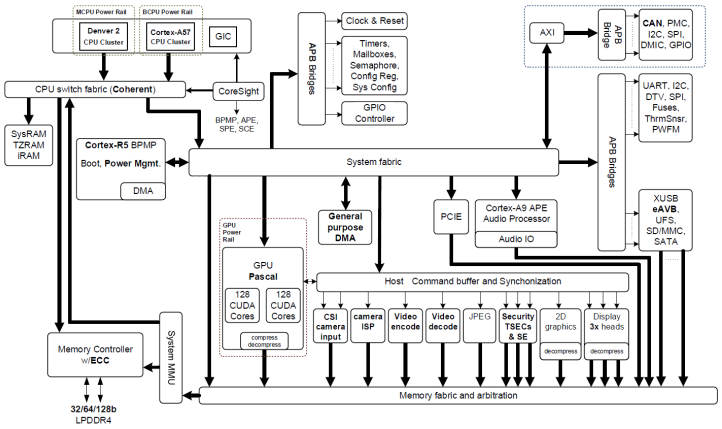

Figure 1.13: NVIDIA Jetson TX2 Tegra "Parker" SoC block diagram featuring integrated NVIDIA Pascal GPU, NVIDIA Denver 2 + Arm Cortex-A57 CPU clusters, and multimedia acceleration engines

Jetson TX2 has multiple multimedia streaming engines to keep its Pascal GPU fed with data by offloading sensor acquisition and distribution. These multimedia engines include six dedicated MIPI CSI-2 camera ports that provide up to 2.5 Gb/s per lane of bandwidth and 1.4 gigapixels/s processing by dual Image Service Processors (ISP), as well as video codecs supporting H.265 at 4K 60 frames per second.

Jetson TX2 accelerates cutting-edge deep neural network (DNN) architectures using the NVIDIA cuDNN and TensorRT libraries, with support for **Recurrent Neural Networks (RNNs)**, **Long Short-Term Memory networks (LSTMs)**, and online **Reinforcement Learning**. Its dual-CAN bus controller enables autopilot integration to control robots and drones that use DNNs to perceive the world around them and operate safely in dynamic environments. Software for Jetson TX2 is provided through NVIDIA's JetPack 3.0 and Linux For Tegra (L4T) Board Support Package (BSP).

	NVIDIA Jetson TX1	NVIDIA Jetson TX2
CPU	Arm Cortex-A57 (quad-core) @ 1.73GHz	Arm Cortex-A57 (quad-core) @ 2GHz + NVIDIA Denver2 (dual-core) @ 2GHz
GPU	256-core Maxwell @ 998MHz	256-core Pascal @ 1300MHz
Memory	4GB 64-bit LPDDR4 @ 1600MHz | 25.6 GB/s	8GB 128-bit LPDDR4 @ 1866Mhz | 59.7 GB/s
Storage	16GB eMMC 5.1	32GB eMMC 5.1
Encoder*	4Kp30, (2x) 1080p60	4Kp60, (3x) 4Kp30, (8x) 1080p30
Decoder*	4Kp60, (4x) 1080p60	(2x) 4Kp60
Camera†	12 lanes MIPI CSI-2 | 1.5 Gb/s per lane | 1400 megapixels/sec ISP	12 lanes MIPI CSI-2 | 2.5 Gb/sec per lane | 1400 megapixels/sec ISP
Display	2x HDMI 2.0 / DP 1.2 / eDP 1.2 | 2x MIPI DSI	
Wireless	802.11a/b/g/n/ac 2×2 867Mbps | Bluetooth 4.0	802.11a/b/g/n/ac 2×2 867Mbps | Bluetooth 4.1
Ethernet	10/100/1000 BASE-T Ethernet	
USB	USB 3.0 + USB 2.0	
PCIe	Gen 2 | 1×4 + 1 x1	Gen 2 | 1×4 + 1×1 or 2×1 + 1×2
CAN	Not supported	Dual CAN bus controller
Misc I/O	UART, SPI, I2C, I2S, GPIOs	
Socket	400-pin Samtec board-to-board connector, 50x87mm	
Thermals‡	-25°C to 80°C	
Power††	10W	7.5W
Price	$299 at 1K units	$399 at 1K units

Figure 1.14: Comparison of Jetson TX1 and Jetson TX2

1.2.5.1 Twice the Performance, Twice the Efficiency

TensorRT optimizes production networks to significantly improve performance by using graph optimizations, kernel fusion, half-precision floating point computation (FP16), and architecture autotuning. In addition to leveraging Jetson TX2's hardware support for FP16, NVIDIA TensorRT is able to process multiple images simultaneously in batches, resulting in higher performance.

TensorRT optimizes production networks to significantly improve performance by using graph optimizations, kernel fusion, half-precision floating point computation (FP16), and architecture autotuning. In addition to leveraging Jetson TX2's hardware support for FP16, NVIDIA TensorRT is able to process multiple images simultaneously in batches, resulting in higher performance.

To benchmark the performance of Jetson TX2 and JetPack 3.0 we compare it against a server class CPU, Intel Xeon E5-2690 v4, and measure the deep learning inference throughput (images per second) using the GoogLeNet deep image recognition network. As shown below figure, Jetson TX2 operating at less than 15 W of power outperforms the CPU operating at nearly 200 W, enabling data center level AI capabilities on the edge.

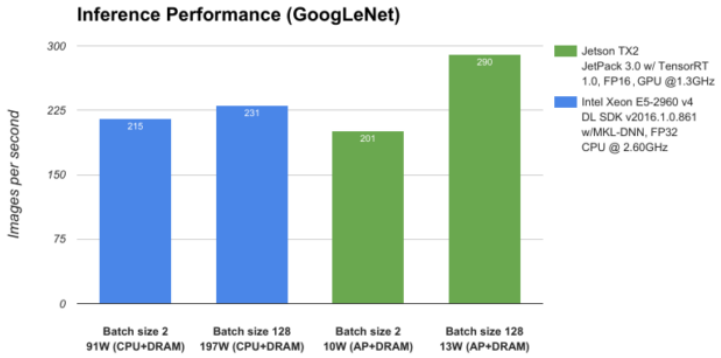

Figure 1.15: Performance of GoogLeNet network architecture profiled on NVIDIA Jetson TX2 and Intel Xeon E5-2960 v4

This exceptional AI performance and efficiency of Jetson TX2 stems from the new Pascal GPU architecture and dynamic energy profiles (Max-Q and Max-P), optimized deep learning libraries that come with JetPack 3.0, and the availability of large memory bandwidth.

1.2.5.2 Max-Q and Max-P

Jetson TX2 was designed for peak processing efficiency at 7.5W of power. This level of performance, referred to as Max-Q, represents the peak of the power/throughput curve. Every component on the module including the power supply is optimized to provide highest efficiency at this point. The Max-Q frequency for the GPU is 854 MHz, and for the Arm A57 CPUs it's 1.2 GHz. The L4T BSP in JetPack 3.0 includes preset platform configurations for setting Jetson TX2 in Max-Q mode. JetPack 3.0 also includes a new command line tool called **nvpmodel** for switching profiles at run time. While Dynamic Voltage and Frequency Scaling (DVFS) permits Jetson TX2's Tegra "Parker" SoC to adjust clock speeds at run time according to user load and power consumption, the Max-Q configuration sets a cap on the clocks to ensure that the application is operating in the most efficient range only. The table below here shows the performance and energy efficiency of Jetson TX2 and Jetson TX1 when running the GoogLeNet and AlexNet deep learning benchmarks. The performance of Jetson TX2 operating in Max-Q mode is similar to the performance of Jetson TX1 operating at maximum clock frequency but consumes only half the power, resulting in double the energy efficiency.

Although most platforms with a limited power budget will benefit most from Max-Q behavior, others may prefer maximum clocks to attain peak throughput, albeit with higher power consumption and reduced efficiency. DVFS can be configured to run at a range of other clock speeds, including underclocking and overclocking. Max-P, the other preset platform configuration, enables maximum system performance in less than 15W. The Max-P frequency is 1122 MHz for the GPU and 2 GHz for the CPU when either Arm A57 cluster is enabled or Denver 2 cluster is enabled and 1.4 GHz when both the clusters are enabled. You can also create custom platform configurations with intermediate frequency targets to allow balancing between peak efficiency and peak performance for your application.

		NVIDIA Jetson TX1	NVIDIA Jetson TX2		
		Max Clock (998 MHz)	Max-Q (854 MHz)	max-P (1122 MHz)	Max Clock (1302 MHz)
GoogLeNet batch=2	Perf	141 FPS	138 FPS	176 FPS	201 FPS
	Power (AP+DRAM)	9.14 W	4.8 W	7.1 W	10.1 W
	Efficiency	15.42	28.6	24.8	19.9
GoogLeNet batch=128	Perf	204 FPS	196 FPS	253 FPS	290 FPS
	Power (AP+DRAM)	11.7 W	5.9 W	8.9 W	12.8 W
	Efficiency	17.44	33.2	28.5	22.7
AlexNet batch=2	Perf	164 FPS	178 FPS	222 FPS	250 FPS
	Power (AP+DRAM)	8.5 W	5.6 W	7.8 W	10.7 W
	Efficiency	19.3	32	28.3	23.3
AlexNet batch=128	Perf	505 FPS	463 FPS	601 FPS	692 FPS
	Power (AP+DRAM)	11.3 W	5.6 W	8.6 W	12.4 W
	Efficiency	44.7	82.7	69.9	55.8

Figure 1.16: Power consumption measurements for GoogLeNet and AlexNet architectures for max-Q and max-P performance levels on NVIDIA Jetson TX1 and Jetson TX2. The table reports energy efficiency for all tests in images per second per Watt consumed

Jetson TX2 performs GoogLeNet inference up to 33.2 images/sec/Watt, nearly double the efficiency of Jetson TX1 and nearly 20X more efficient than Intel Xeon.

1.2.5.3 End-to-End AI Applications

Integral to Jetson TX2's efficient performance are two **Pascal Streaming Multiprocessors (SMs)** with 128 CUDA cores each. The Pascal GPU architecture offers major performance improvements and power optimizations. TX2's CPU Complex includes a dual-core 7-way superscalar NVIDIA Denver 2 for high single-thread performance with dynamic code optimization, and a quad-core Arm Cortex-A57 geared for multithreading.

The coherent Denver 2 and A57 CPUs each have a 2MB L2 cache and are linked via high-performance interconnect fabric designed by NVIDIA to enable simultaneous operation of both CPUs within a Heterogeneous Multiprocessor (HMP) environment. The coherency mechanism allows tasks to be freely migrated according to dynamic performance needs, efficiently utilizing resources between the CPU cores with reduced overhead.

Jetson TX2 is the ideal platform for the end-to-end AI pipeline for autonomous machines. Jetson is wired for streaming live high-bandwidth data: it can simultaneously ingest data from multiple sensors and perform media decoding/encoding, networking, and low-level command & control protocols after processing the data on the GPU. Figure below here showns common pipeline configurations with sensors attached using an array of high-speed interfaces including CSI, PCIe, USB3, and Gigabit Ethernet. The CUDA pre- and post-processing stages generally consist of colorspace conversion (imaging DNNs typically use BGR planar format) and statistical analysis of the network outputs.

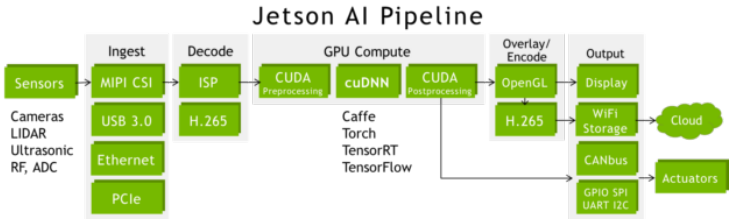

Figure 1.17: End-to-end AI pipeline including sensor acquisition, processing, command & control

With double the memory and bandwidth than Jetson TX1, Jetson TX2 is able to capture and process additional streams of high-bandwidth data simultaneously, including stereo cameras and 4K ultra-HD inputs and outputs. Through the pipeline deep learning and computer vision fuse together multiple sensors from varying sources and spectral domains, increasing perception and situational awareness during autonomous navigation.

1.2.5.4 Jetson TX2 Developer Kit

NVIDIA provides the Jetson TX2 Developer Kit complete with a reference mini-ITX carrier board (170 mm x 170mm) and a 5-megapixel MIPI CSI-2 camera module. The Developer Kit includes documentation and design schematics and free software updates to JetPack-L4T. Down here is the picture of the development kit, showing the Jetson TX2 module and standard PC connections including USB3, HDMI, RJ45 Gigabit Ethernet, SD card, and a PCIe x4 slot, which make it easy to develop applications for Jetson.

Figure 1.18: NVIDIA Jetson TX2 Developer Kit including module, reference carrier, and camera module

To move beyond development to custom deployed platforms, you can modify the reference design files for the Developer Kit carrier board and camera module to create a custom design. Alternatively, Jetson ecosystem partners offer off-the-shelf solutions for deploying Jetson TX1 and Jetson TX2 modules, including miniature carriers, enclosures, and cameras.

Chapter 2

OBJECT DETECTION

2.1 Overview

2.1.1 Traditional Methods

Traditional object detection methods are built on handcrafted features and shallow trainable architectures.

Features used in traditional methods: color feature, HOG feature, edge feature, optical flow features, texture features,...

The pipeline of traditional object detection models can be mainly divided into three stages: informative region selection, feature extraction, and classification.

2.1.2 Deep Learning Based Method

2.1.2.1 ANN

An ANN consists of interconnected groups of nodes. Each group is called a layer and each node is called a neuron. The connection between neurons, replicating synapses in biological brains, transfers information between layers. A very simple ANN can be seen in where two inputs are going through a hidden layer in order to produce an output.

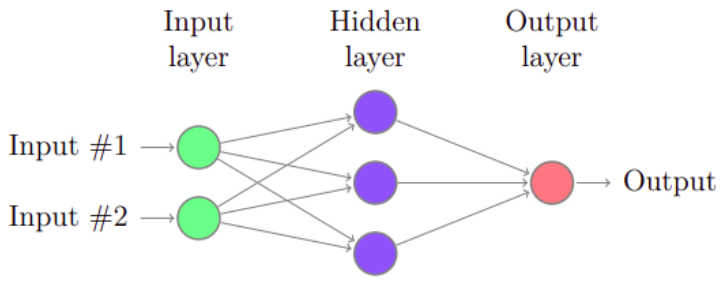

Figure 2.1: Example of an Artificial Neural Network with two input, one hidden and output layer

2.1.2.2 Neurons

The artificial neurons in neural networks is a mathematical model of a biological neuron, modelled to replicate the function of being excitable depending on input signals. The equation of a neuron, visualized as one of the blue circles, is simply a weighted sum of all the inputs plus a bias term, i.e.

$$s(x) = \sum_{i=0}^{n} w_i x_i + b_i \tag{2.1}$$

where n is the number of inputs and w_i the weights in the neuron and b_i the bias term. The

combination of weights and input signals, giving the neuron its value together with the bias term, represents the excitability.

2.1.2.3 Activation Functions

The activation functions are essential components of ANNs, used to perform nonlinear mappings of the input data and are typically applied element-wise to all neurons in a hidden layer. This section describes a few commonly used activation functions and their properties. The activation functions commonly used in the intermediate layers of ANNs are presented.

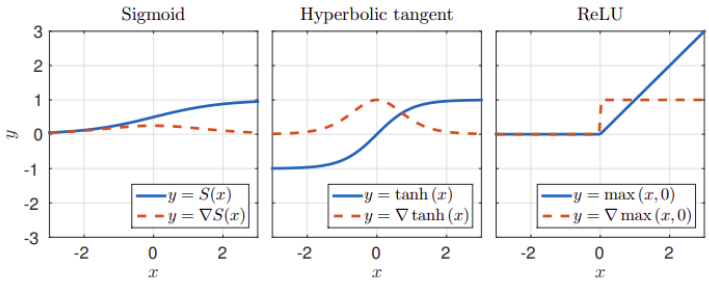

Figure 2.2: The sigmoid, hyperbolic tangent and ReLU activation functions and their derivatives

1. **Sigmoid**

   The sigmoid activation functions is defined as:

   $$S(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

   It takes values from 0 to 1 and is linear close to the origin. It has a "squashing" property such that large positive or negative input values result in values close to 1 or 0 respectively.

2. **Hyperbolic Tangent**

   The hyperbolic tangent:

   $$tan(x) = \frac{sin(x)}{cos(x)} = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.3}$$

   is also commonly used as activation function in ANNs. It has very similar properties to the sigmoid function except tan(x) takes values from -1 to 1.

3. **Rectified Linear Unit**

   The Rectified Linear Unit (ReLU) is an activation function commonly used with deep neural networks, and is simply given by:

   $$f(x) = max(x, 0) \tag{2.4}$$

   Compared to the commonly used sigmoid activation function, the ReLU has the advantage of not saturating for large inputs. While sigmoidal functions take values in the range (0, 1), ReLUs take values in $[0, \infty)$ making them less prone to be either “on” or “off”.

4. **Softmax**

   The softmax function is a normalising function commonly used as an activation function for the last layer of a neural network. Given an input vector x with values x_i, the corresponding output element y_i is calculated.

   $$y_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \tag{2.5}$$

By construction, the elements of the output vector sums to 1. This property is useful when the output of the neural network should encode probabolities for different cases, such as in classification tasks.

2.1.2.4 Convolution Neural Networks

Convolutional Neural Networks (CNNs) are feedforward neural networks with a layout and architecture specifically designed to handle data arranged in a spatial grid (tensors), such as 2D or 3D images. The inspiration of the architecture comes from the mechanism of biological visual perception. The networks, like any other ANN, are composed of neurons with learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with an activation function. The architecture is typically composed of several layers, which gives them the characterisation of being "deep" and thus research work on CNNs fall under the domain of deep learning. Essentially, the network computes a mapping function that relates image pixels to a final desired output.

In a general CNN, the input is assumed to be an RGB image, i.e. consisting of three channels, corresponding to the red, green and blue color intensity values. Consecutive layers of the CNN may consist of even more channels referred to as *feature maps*. The number of feature maps typically increase through the layers of a CNN, while the spatial dimension of them decreases until reaching the desired output size. The over-all idea behind this structure is that the representation of the input image is gradually increased in abstraction as it progresses through the layers. Later layers contain more information about the "what" and "how" of objects and things in an image, and less of "where". Similar to the definition of a feature map, a *feature vector* at a specific layer of a CNN is defined to be the elements across all feature maps at a given spatial location. An example of the structure of a CNN is presented.

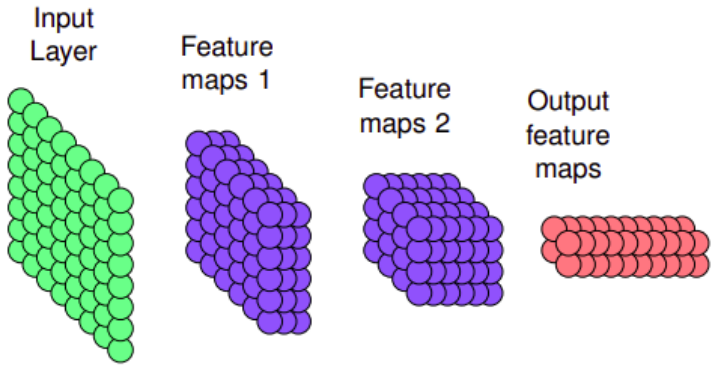

Figure 2.3: Illustration of how neurons are structured in a CNN

1. **Components of CNNs**

   Besides the fully connected layer presented in, there are other types of layers and connections that can be used to construct deep convolutional networks. Typically, the purposes behind some of these components are to reduce the dimensions of intermediate layers, reshaping spatial dimensions, simulating fully connected layers and more. Following are some sections describing important components in the CNN framework used in this thesis.

   ❒Convolution

   The discrete 2D convolution operation, is defined by a convolution kernel k of size k *x* k. Given an N *x* M input image (tensor) $\mathbf{X}$, the convolution kernel is run along all the pixels in the image, multiplying the surrounding pixel values with the kernel and adding them. The resulting output is a *(N -k +1) x (M - k -1)* image $\mathbf{Y}$, where the output at pixel location *i,j* is calculated:

$$Y_{i,j} = \sum_{i'=1}^{k} \sum_{j'=1}^{k} k_{i',j'} X_{\frac{i-(k+1)}{2+j'}, \frac{j-(k+1)}{2+j'}} \tag{2.6}$$

The operator is denoted using * and thus

$$Y = k * X \tag{2.7}$$

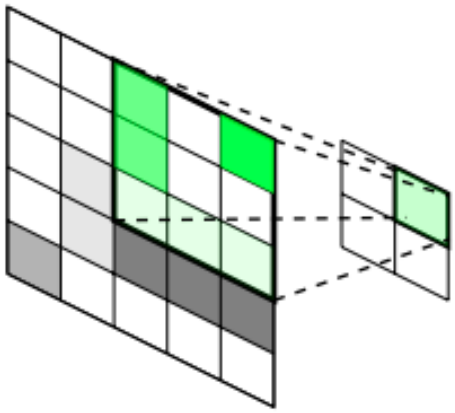

Figure 2.4: Illustration of the convolution operation using a 3 x 3 kernel

The convolution operation between two layers in a CNN is defined by a kernel k_{ij} for each pair of feature maps in the two layers. Similar to the fully connected layer, the result from all convolutions related to feature map i in the later layer is summed and a bias b_i is distributed and added across the entire feature map. Let X_j, be a feature map in a layer with N≥j features maps, then the feature map Y_i in the next layer is calculated

$$Y_i = \sigma_i(\sum_{j=1}^{N} k_{ij} * X_j + b_i 1) \tag{2.8}$$

where **1** is an all-ones matrix and σ_i is an element-wise activation function. The elements of all kernels k_{ij} and the biases b_i are learnable parameters and are updated through the process of training the network.

As mentioned, the convolution operation has the effect of reducing the dimensions of the feature maps. However, there are some methods which can be used to avoid or modify this effect. A common strategy to preserve the spatial dimensions is to apply zero-padding, i.e. covering the feature maps before the convolution operation with a border of $\frac{k-1}{2}$ zeros, which counters the spatial reduction exactly. On the other hand, to reduce the spatial dimensions of intermediate layer activations further, a *stride* can be associated with the convolution. The stride is applied by sliding the convolution kernel by s number of steps between each "multiply and sum" operation and the result is that the feature maps are reduced in size by a factor s. Most commonly, the convolutions are applied without stride, i.e. s = 1.

An important concept introduced by applying a convolution layer to a CNN is the *receptive field*. The receptive field is a measure of how much information from the input image is available to the feature vectors of a specific layer in a CNN. If the first component of a CNN is a convolutional layer with kernel width k_1, the receptive field of the first layer is k1 x k1 – i.e. each element in the second layer was provided information from the k1×k1 nearest pixels in the input image. Following this pattern, a series of n convolution layers with kernel sizes $k_1, ..., k_n$ results in a receptive field of

$$(1 + \sum_{i=1}^{n}(k_i - 1)) \times (1 + \sum_{i=1}^{n}(k_i - 1)) \tag{2.9}$$

❒Up-convolution or backwards convolution

An operation analogous to the convolution operation, but reversed, referred to as up-convolution can be used for upsampling. It is achieved by convolution operation reversed. Hence, if

upsampling by a factor f is desired, it can be formulated as a convolution with a fractional input stride of $\frac{1}{f}$. Such a layer can associate a single input activation to multiple output activations. This "upconvolution" layer has learnable filter parameters that could correspond to bases for reconstructing shapes of an object. Hence, an end-to-end learning mechanism can be constructed by repeated downsampling and then upsampling to achieve dense predictions, and this technique has been successfully used for dense pixel-level predictions.

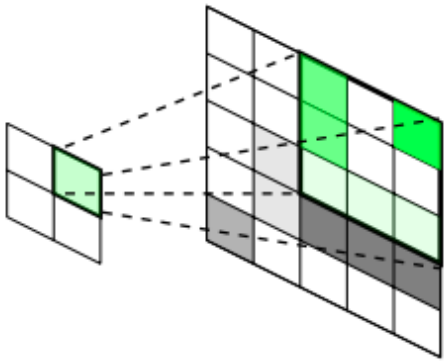

Figure 2.5: Illustration of the upconvolution operation

❒Pooling

Pooling layers are non-learnable layers used to reduce the spatial dimensions of the feature maps as they pass through the network. Similar to the convolution layer, they are associated with some kernel of size k x k and a stride s. There are two commonly used types of pooling layers; the max-pooling layer and the averagepooling layer. The max-pooling layer,performs a max () operation with the elements of the feature map at each position of the kernel, thus discarding the information of the non-max neurons.

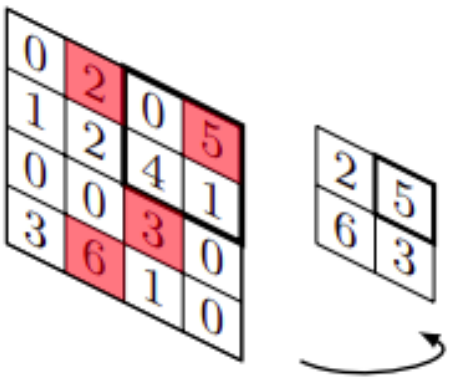

Figure 2.6: Illustration of the max-pooling operation

The average-pooling layer performs an average at each position of the kernel, i.e. a normal convolution with the kernel values all set to $\frac{1}{k^2}$. Typically, the stride of the pooling layers is set to s=k , thus achieving a dimensional reduction of a factor s without using any zero-padding.

Apart from reducing the spatial dimensions of the feature maps, pooling layers also provide an efficient way of increasing the receptive field in a CNN. A pooling layer with stride s have the effect of increasing the receptive field of a factor s. This effect can also we achieved by including stride in a convolutional layer.

❒Unpooling

A reverse operation to pooling, “unpooling” aims to recover the original size of activations that

were lost due to a pooling operation. A version of unpooling specifically for reversing the max-pooling operation, records the maximum activations selected during the pooling operation and uses them to recreate the original activations by placing the recorded values in their original positions. This strategyis useful to recover the structure of objects and have a parameter-free upsampling.

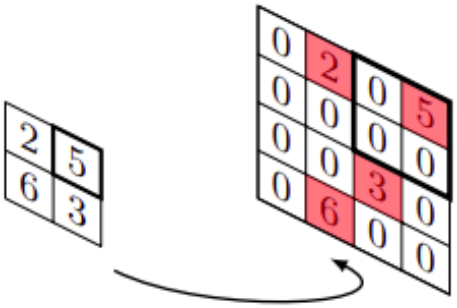

Figure 2.7: Illustration of the max-unpooling operation

❒1 x 1 Convolutions

Technically, 1 × 1 convolutional kernels are no different from any k × k kernel in the way they are applied. However, there is a conceptual difference between them in that k × k convolutions are usually thought of as edge/feature detectors while 1 × 1 kernels can only combine activations of each feature vector. The interpretation of such an operation is that it is simulating the effect of a fully connected layer, applied to each feature vector. Since the convolution operation is not dependant on the spatial size of its input, transforming fully connected layers to 1 × 1 convolutions is a useful way of generalizing the network for different input sizes.

❒Batch Normalisation

Training deep neural networks can be quite tricky in practice due to the fact that the distribution of each intermediate layer's inputs changes during training. Such a change is caused by the changes in the parameters of the previous layers. This problem is referred to as *internal covariate shift*. When the input distribution changes, the activations tend to move into the saturated regimes, and this effect is amplified as the network depth increases, thus also causing the vanishing gradient problem. To tackle this, a normalisation technique was introduced, where normalisation of the inputs to the activation layers are done over the mini-batch. The batch normalising transform algorithm is briefly presented. Consider a mini-batch of n inputs in a mini-batch $\beta = \{x_1, x_2, ..., x_n\}$. Let the normalized values be $\{\hat{x_1}, \hat{x_2}, ..., \hat{x_n}\}$ and the resulting linear transformation of the batch normalisation can be represented by $\{y_1, y_2, ..., y_n\}$. Then, the batch normalising transform given by

$$\mathbf{BN}_{\gamma,\beta} : \{x_1, x_2, ..., x_n\} \rightarrow \{y_1, y_2, ..., y_n\} \tag{2.10}$$

where γ andβ are learnable parameters that correspond to the scaling and shifting in the transformation. Hence the mini-batch mean and variance which are given by

$$\mu\beta \leftarrow \frac{1}{n}\sum_{i=1}^{n} x_i \tag{2.11}$$

$$\sigma_\beta^2 \leftarrow \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu\beta)^2 \tag{2.12}$$

are used to achieve the normalisation. Thus, the normalisation looks like

$$\hat{x}_i \leftarrow \frac{x_i - \mu\beta}{\sqrt{\sigma_\beta^2 + \in}} \tag{2.13}$$

and the resulting transformation can be given by

$$y_i \leftarrow \gamma \hat{x}_i + \beta \tag{2.14}$$

2.1.2.5 Spped/Accuracy Trade-offs

The relation between these two is very complex, as it is very hard to maintain them both at a high level. In the case of real-time environment where the focus is on Advanced driver-assistance systems (ADAS), a decision needs to be made fast, and by that means that the detection of pedestrians and vehicle needs to be fast. In a trade-off there are two alternatives, decisions are made fast which results to higher error-rate or slow which then results in higher accuracy.

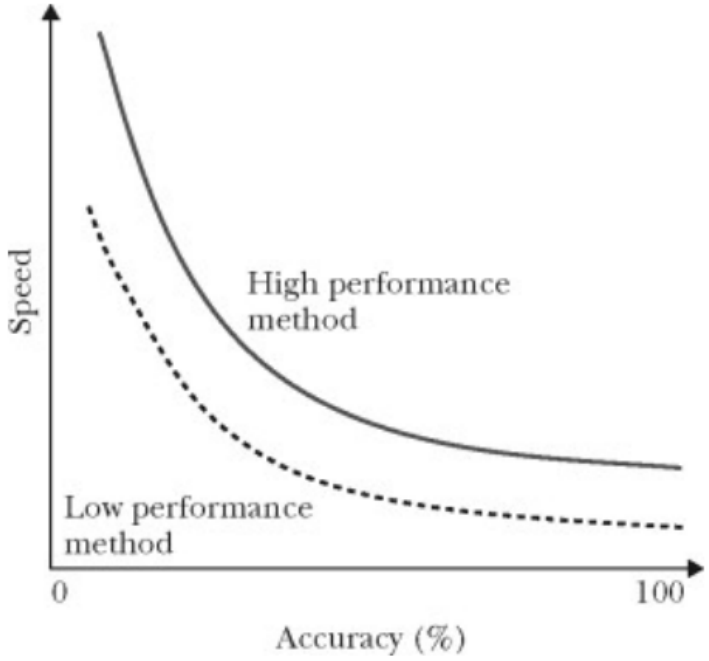

Figure 2.8: Speed/Accuracy trade-of

As shown above, the performance of each axis depend on the other, so if speed decreases than the accuracy increases. The trade-off between speed and accuracy needs both at a balanced level, otherwise it will be fast with weak detection or good detection with low speed. The trade-off between speed and accuracy needs to be at a balanced level.

2.1.2.6 Categorize

Thanks to Deep Neural Network, a more significant gain is obtained with the introduction of regions with convolutional neural network (CNN) features (R-CNN). DNNs, or the most representative CNNs, act in a quite different way from traditional approaches. They have deeper architectures with the capacity to learn more complex features than the shallow ones. Also, the expressivity and robust training algorithms allow to learn informative object representations without the need to design features manually.

Since the proposal of R-CNN, a great deal of improved models have been suggested, including fast R-CNN that jointly optimizes classification and bounding box regression tasks, faster R-CNN that takes an additional subnetwork to generate region proposals, and you only look once (YOLO) that accomplishes object detection via a fixed-grid regression.

2.1.2.7 Region Proposal Based Method

Generate region proposals at first and then classify each proposal into different object categories. Frameworks may be included such as: R-CNN, Faster R-CNN, region-based fully convolutional network R-FCN, feature pyramid networks (FPN), and Mask R-CNN.

Region proposal-based frameworks are composed of several correlated stages, including region proposal generation, feature extraction with CNN, classification, and bounding box regression, which are usually trained separately. Even in the recent end-to-end module Faster R-CNN, an alternative training is still required to obtain shared convolution parameters between RPN and detection network. As a result, the time spent in handling different components becomes the bottleneck in the real-time application.

2.1.2.8 Regression / Classification Based Method

Regression/classification based framework solves object detection problem by regarding it as regression or classification problem. Some frameworks may be accounted for examples are: MultiBox, AttentionNet, G-CNN, YOLO, Single Shot MultiBox Detector (SSD), YOLOv2, YOLOv3.

One-step frameworks based on global regression/classification, mapping straightly from image pixels to bounding box coordinates and class probabilities, can reduce time expense.

2.2 Comparison YOLO SSD Faster R-CNN

2.2.1 Faster-RCNN

2.2.1.1 Architecture

The architecture of Faster R-CNN is shown in the next figure. It consists of 2 modules:

- **RPN:** For generating region proposals.
- **Faster R-CNN:** For detecting objects in the proposed regions.

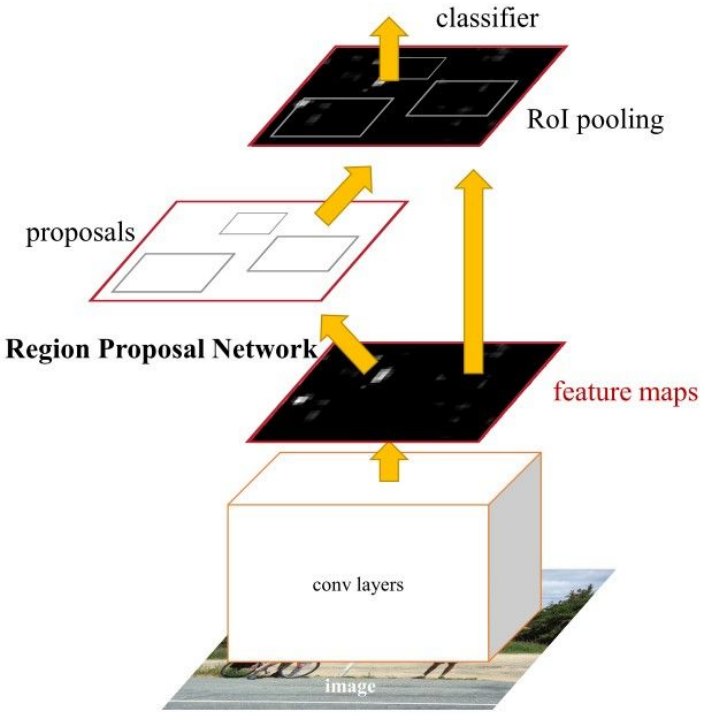

Figure 2.9: Pipeline of Faster R-CNN

2.2.1.2 Region Proposal Generation

The RPN module is responsible for generating region proposals. It applies the concept of attention in neural networks, so it guides the Fast R-CNN detection module to where to look for objects in the image.

The network slides over the conv feature map and fully connects to an n×n spatial window. A low-dimensional vector (512-dimensional for VGG16) is obtained in each sliding window and fed into two sibling FC layers, namely, box-classification layer (cls) and box-regression layer (reg). This architecture is implemented with an n × n conv layer followed by two sibling 1 × 1 conv layers. To increase nonlinearity, ReLU is applied to the output of the n × n conv layer.

The cls layer outputs a vector of 2 elements for each region proposal. If the first element is 1 and the second element is 0, then the region proposal is classified as background. If the second element is 1 and the first element is 0, then the region represents an object. In other words, it represents a binary classifier that generates the objectness score for each region proposal.

RPN produces better region proposals compared to generic methods like Selective Search and EdgeBoxes, which are implemented in RCNN and Fast - RCNN.

The RPN processes the image using the same convolutional layers used in the Fast R-CNN detection network. Thus, the RPN does not take extra time to produce the proposals compared to the algorithms like Selective Search and reduce training time as the RPN and the Fast R-CNN can be merged into a single network.

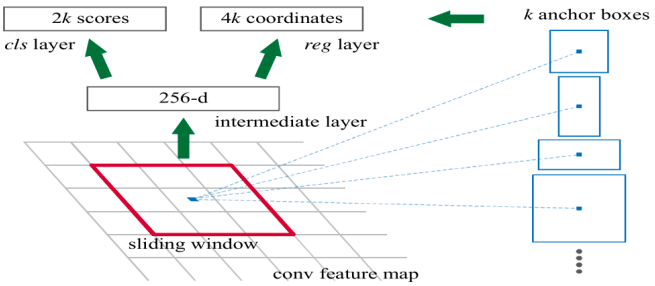

Figure 2.10: Architecture of RPN

2.2.1.3 Detection Head

1. **Anchors**

   The feature map of the last shared convolution layer is passed through a rectangular sliding window of size n*n where n = 3 for the VGG-16 net. For each window, K region proposals are generated. Each proposal is parametrized according to a reference box which is called an anchor box. The 2 parameters of the anchor boxes are scale and aspect ratio. Anchors of three scales and three aspect ratios are adopted, therefore K regions are produced from each region proposals. The multi-scale anchors are key to share features across the RPN and the Fast R-CNN detection network.

   For training the RPN, each anchor is given a positive or negative **objectness score** based on the Intersection-over Union (IoU).

   The next 4 conditions use the IoU to determine whether a positive or a negative **objectness score** is assigned to an anchor. Based on this, classification label is produced.

   - An anchor that has an IoU overlap higher than **0.7** with any ground-truth box is given a positive objectness label.
   - If there is no anchor with an IoU overlap higher than **0.7**, then assign a positive label to the anchor(s) with the highest IoU overlap with a ground-truth box.
   - A negative **objectness score** is assigned to a **non-positive** anchor when the IoU overlap for all ground-truth boxes is less than **0.3**. A negative objectness score means the anchor is classified as background.
   - Anchors that are neither positive nor negative do not contribute to the training objective.

2. **Loss Function**

$$L(p_i, t_i) = \frac{1}{N_{cls}} \sum_i L_c ls(p_i, p_i*) + \lambda \frac{1}{N_{reg}} \sum_i p_i * L_{reg}(t_i, t_i*) \tag{2.15}$$

Where p_i is the predicted probability of the i^{th} anchor being an object. The ground truth label

p_i* is 1 if the anchor is positive, otherwise 0. t_i stores four parameterized coordinates of the predicted boudaring box while t_i* is related to the ground truth box overlapping with a positive anchor. $L_c ls$ is a binary log loss and $L_r eg$ is a smoothed L_1 loss. These two terms are normalized with the mini-batch size ($N_c ls$) and the number of anchor locations ($N_r eg$).

2.2.1.4 Observations

1. **Pros**

   With the proposal of Faster R-CNN, region proposal-based CNN architectures for object detection can really be trained in an end-to-end way.

2. **Cons**

   However, the alternate training algorithm is very time-consuming and RPN produces object-like regions (including backgrounds) instead of object instances and is not skilled in dealing with objects with extreme scales or shapes.

2.2.2 SSD

2.2.2.1 Feature Extraction

SSD uses VGG16 to extract feature maps. VGG16 is a convolutional neural network model proposed by K. Simonyan and A. Zisserman from the University of Oxford in the paper "Very Deep Convolutional Networks for Large-Scale Image Recognition". The model achieves 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to 1000 classes. It was one of the famous model submitted to ILSVRC-2014. It makes the improvement over AlexNet by replacing large kernel-sized filters (11 and 5 in the first and second convolutional layer, respectively) with multiple 3×3 kernel-sized filters one after another. It is one of the most preferred choices in the community for extracting features from images. The configuration of the VGGNet is publicly available and has been used in many other applications and challenges as a baseline feature extractor.

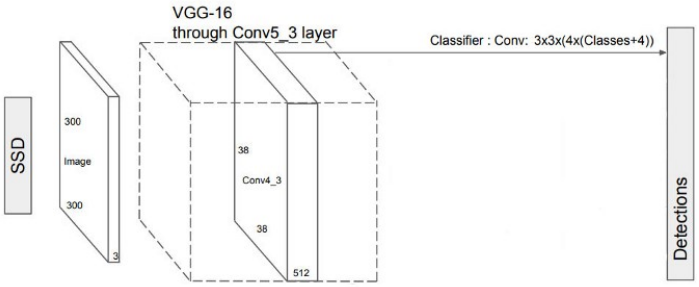

Figure 2.11: Feature Extractor of SSD

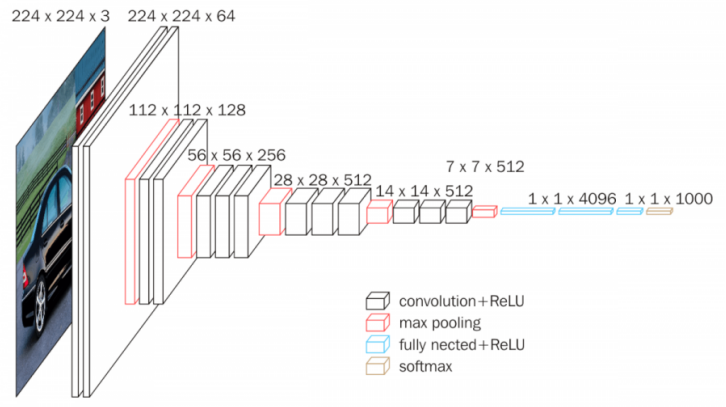

Figure 2.12: Architecture of VGG16

2.2.2.2 Multi-scale feature maps for detection

Given a specific feature map, instead of fixed grids adopted in YOLO, SSD uses bounding box regression technique,it takes the advantage of a set of default anchor boxes with different aspect ratios and scales to discretize the output space of bounding boxes. To handle objects with various sizes, the network fuses predictions from multiple feature maps with different resolutions.

SSD adds several feature layers to the end of VGG16 backbone network to predict the offsets to default anchor boxes and their associated confidences. Final detection results are obtained by conducting NMS on multiscale refined bounding boxes.

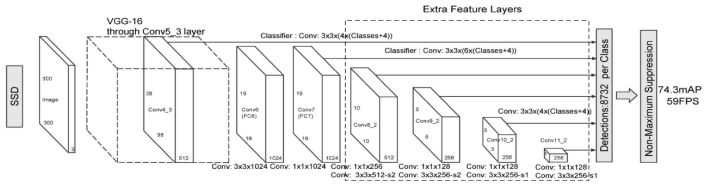

Figure 2.13: Architecture of SSD

2.2.2.3 Default Boundary Box

The default boundary boxes are equivalent to anchors in Faster R-CNN. Default boundary boxes are chosen manually. SSD defines a scale value for each feature map layer. Combining the scale value with the target aspect ratios, we compute the width and the height of the default boxes. For layers making 6 predictions, SSD starts with 5 target aspect ratios: 1, 2, 3, 1/2, and 1/3. Then the width and the height of the default boxes are calculated as:

$$w = scale.\sqrt{aspectratio} \tag{2.16}$$

$$h = \frac{scale}{\sqrt{aspectratio}} \tag{2.17}$$

Then SSD adds an extra default box with scale:

$$scale = \sqrt{scale.scaleatnextlevel} \tag{2.18}$$

Where aspect ratio is equal to 1.

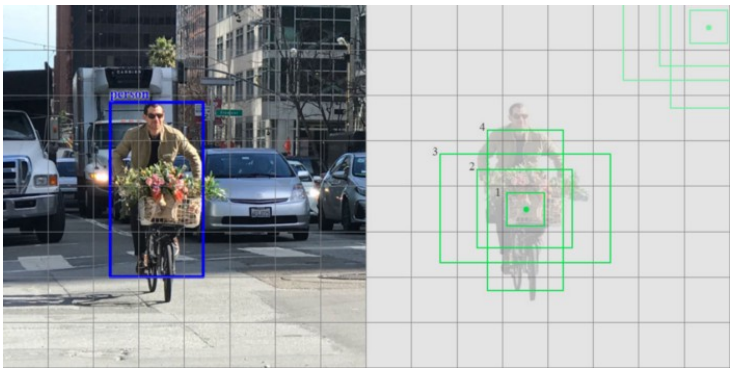

Figure 2.14: Default Boundary Box

2.2.2.4 Matching Strategy

SSD predictions are classified as positive matches or negative matches. SSD only uses positive matches in calculating the localization cost (the mismatch of the boundary box). If the corresponding default boundary box (not the predicted boundary box) has an IoU greater than 0.5 with the ground truth, the match is positive. Otherwise, it is negative. Once we identify the positive matches, we use the corresponding predicted boundary boxes to calculate the cost. This matching strategy nicely partitions what shape of the ground truth that a prediction is responsible for.

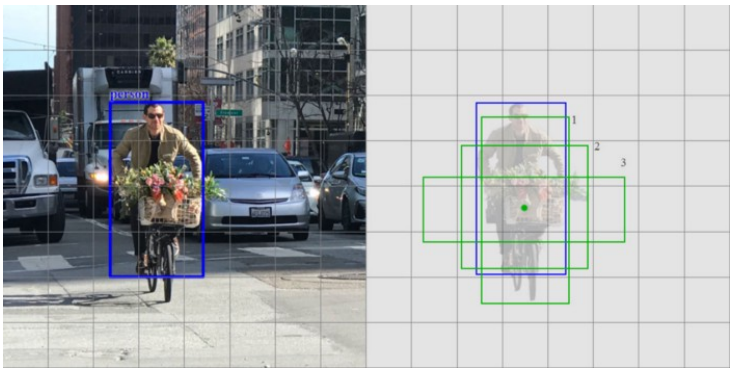

Figure 2.15: Matching with ground truth

2.2.2.5 Loss Function

There are two types of loss functions here: Confidence loss and Location loss.

- There are two types of loss functions here: Confidence loss and Location loss.

  The localization loss between the predicted box **l** and the ground truth box **g** is defined as the smooth L_1 loss with c_x, c_y as the offset to the default bounding box **d** of width **w** and height **h**.

$$L_loc(x,l,g) = \sum_{i \in Pos}^{N} \sum_{m \in c_x, c_x, w, h} x_{ij}^k smooth_{L_1}(l_i^m - \hat{g}_j^m) \tag{2.19}$$

$$\hat{g}_j^{c_x} = \frac{(g_j^{c_x} - d_i^{c_x})}{d_i^w}; \hat{g}_j^{c_y} = \frac{(g_j^{c_y} - d_i^{c_y})}{d_i^h} \tag{2.20}$$

$$\hat{g}_j^w = \log(\frac{g_j^w}{d_i^w}); \hat{g}_j^h = \log(\frac{g_j^h}{d_i^h}) \tag{2.21}$$

$$x_{ij}^p = \begin{cases} 1 & \text{if IoU} > 0.5 \text{ between default box i and ground true box j on class p} \\ 0 & \text{Otherwise} \end{cases} \tag{2.22}$$

- The confidence loss is a measure of the confidence that an algorithm quantifies if a bounding box in an image consists of any object or class. For every positive match prediction, we penalize the

loss according to the confidence score of the corresponding class. For negative match predictions, we penalize the loss according to the confidence score of the class "0": class "0" classifies no object is detected.The alpha term balances the contribution of the location losses. The main objective in neural neutral is to analyse the parameters that reduce the loss predictions.

It is calculated as the softmax loss over multiple classes confidences c (class score).

$$L_{conf}(x,c) = -\sum_{i \in Pos}^{N} x_{ij}^{p} \log(\hat{c}_i^0) where \hat{c}_i^p = \frac{exp(c_i^p)}{(\sum)_p exp(c_i^p)} \tag{2.23}$$

where N is the number of matched default boxes.

- The final loss function is computed as:

$$L(x,c,l,g) = \frac{1}{N}(L_{conf}(x,c) + \propto L_{loc}(x,l,g)) \tag{2.24}$$

where N is the number of positive matches and α is the weight for the localization loss.

2.2.2.6 Hard negative mining

SSD still requires negative sampling so it can learn what constitutes a bad prediction. So, instead of using all the negatives, we sort those negatives by their calculated confidence loss. SSD picks the negatives with the top loss and makes sure the ratio between the picked negatives and positives is at most 3:1. This leads to faster and more stable training.

2.2.2.7 Observations

1. **Pros**

   SSD is a single-shot detector. It has no delegated region proposal network and predicts the boundary boxes and the classes directly from feature maps in one single pass.

   To improve accuracy, SSD introduces:

   - Small convolutional filters to predict object classes and offsets to default boundary boxes.
   - Separate filters for default boxes to handle the difference in aspect ratios.
   - Multi-scale feature maps for object detection.

   SSD can be trained end-to-end for better accuracy. SSD makes more predictions and has better coverage on location, scale, and aspect ratios. With the improvements above, SSD can lower the input image resolution to 300 × 300 with a comparative accuracy performance. Integrating with hard negative mining, data augmentation, and a larger number of carefully chosen default anchors, SSD significantly outperforms the Faster R-CNN in terms of accuracy on PASCAL VOC and COCO while being three times faster.

2. **Cons**
   Shallow layers in a neural network may not generate enough high level features to do prediction for small objects. Therefore, SSD does worse for smaller objects than bigger objects.

   The need of complex data augmentation also suggests it needs a large number of data to train. For example, SSD does better for Pascal VOC if the model is pretrained on COCO dataset.

2.2.3 YOLO

2.2.3.1 Network Architecture

The whole system can be divided into two major components: Feature Extractor and Detector; both are multi-scale. When a new image comes in, it goes through the feature extractor first so that we can obtain feature embeddings at three (or more) different scales. Then, these features are feed into three (or more) branches of the detector to get bounding boxes and class information.

Figure 2.16: Major components

2.2.3.2 Feature Extraction

The feature extractor YOLO V3 uses is called Darknet-53. Darknet-53 contains 53 layers and borrows the ideas of skip connections to help the activations to propagate through deeper layers without gradient diminishing from ResNet. But the Darknet-53 claims to be more efficient than ResNet101 or ResNet152.

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
	Convolutional	32	1 × 1	
1×	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
	Convolutional	64	1 × 1	
2×	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
	Convolutional	128	1 × 1	
8×	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
	Convolutional	256	1 × 1	
8×	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
	Convolutional	512	1 × 1	
4×	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 2.17: Feature Extraction of YOLOv3

Inside the block, there's just a bottleneck structure (1x1 followed by 3x3) plus a skip connection. If the goal is to do multi-class classification as ImageNet does, an average pooling and a 1000 ways fully

connected layers plus softmax activation will be added. However in the case of object detection, the classification head won't be included, instead, the "detection" head will be added to this feature extractor. Features from last three residual blocks are used in the later detection.

2.2.3.3 Multi-scale Detector

1. **IOU (Intersection Over Union)**

   Intersection over Union is an evaluation metric used to measure the accuracy of an object detector on a particular dataset.

   Intersection over Union is a ratio. In the numerator we compute the area of overlap between the predicted bounding box and the ground-truth bounding box The denominator is the area of union, or more simply, the area encompassed by both the predicted bounding box and the ground-truth bounding box. Dividing the area of overlap by the area of union yields our final score — the Intersection over Union.

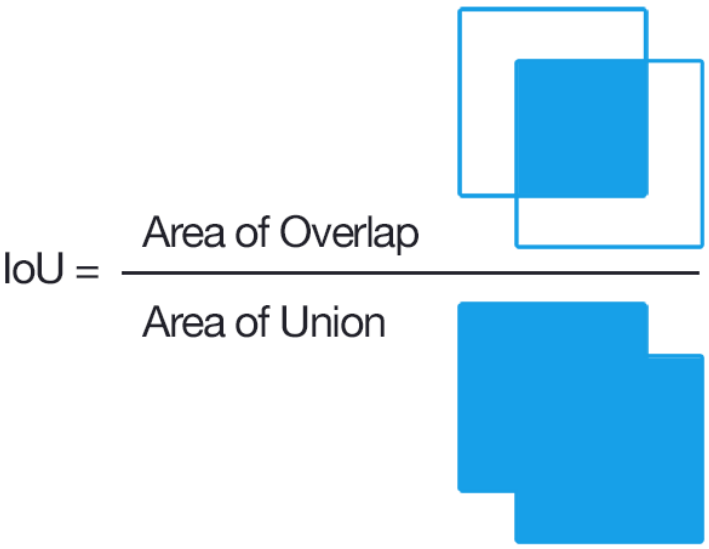

Figure 2.18: IOU formula

2. **Anchor Box**

   The goal of object detection is to get a bounding box and its class. Bounding box usually represents in a normalized xmin, ymin, xmax, ymax format.

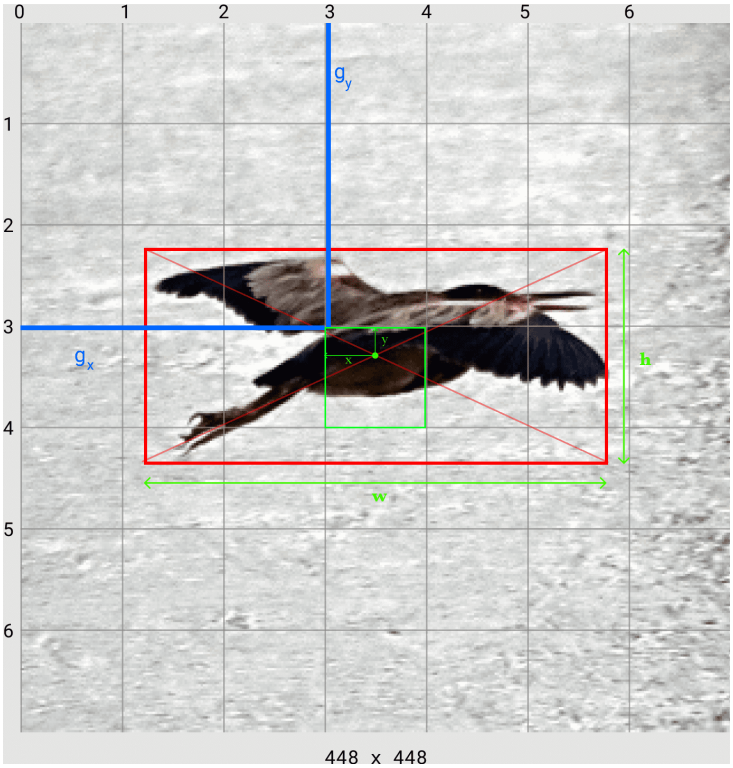

Figure 2.19: Example for anchor box

The input image is divided into an S x S grid of cells. For each object that is present on the image, one grid cell is said to be "responsible" for predicting it. Anchor boxes are assigned to each cell of the grid. And once we defined those anchors, we can determine how much does the ground truth box overlap with the anchor box and pick the one with the best IOU and couple them together.

In YOLO v3, we have three anchor boxes per grid cell. And we have three scales of grids:

- The location offset against the anchor box: tx,ty,tw,th. This has 4 values.
- The confidence score to indicate if this box contains an object. This has 1 value. The confidence score is defined as $P_{object} * IOU_{pred}^{truth}$ which indicates how likely there is exist objects ($P_{object} \geq 0$) and show confidence of its prediction (IOU_{pred}^{truth}).
- The conditional class probabilities $P(Class_i|Object)$ to tell us which class this box belongs to. This has number of values according to number of classes.

3. **Loss Function**

During training the following loss function introduced in YOLO paper is optimize.

$$\lambda_{coord} \sum_{i=0}^{s^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(x_i - \hat{x_i})^2 + (y_i - \hat{y_i})^2] \tag{2.25}$$

$$\lambda_{coord} \sum_{i=0}^{s^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w_i}})^2 + (\sqrt{h_i} - \sqrt{\hat{h_i}})^2] \tag{2.26}$$

$$\sum_{i=0}^{s^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} (C_i - \hat{C_i})^2 \tag{2.27}$$

$$\lambda_{noobj} \sum_{i=0}^{s^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} (C_i - \hat{C_i})^2 \tag{2.28}$$

$$\sum_{i=0}^{s^2} \mathbb{1}_{i}^{obj} \sum_{c \in classes} (p_i(c) - \hat{p_i}(c))^2 \tag{2.29}$$

The loss function contain 4 parts : centroid loss, anchor box's width and height loss, confidence loss, classification loss.

❒**Centroid Loss:** the smaller this loss is, the closer the centroids of predictions and ground truth are. Since this is a regression problem, we use mean square error here. Besides, if there're no object from the ground truth for certain cells, we don't need to include the loss of that cell into the final loss. Therefore we also multiple by object mask. Object mask is either 1 or 0, which indicates if there's an object or not.

❒**Anchor's box width and height loss:** this term penalizes the bounding box with inacurate height and width. The square root is present so that erors in small bounding boxes are more penalizing than errors in big bounding boxes.

❒**Confidence loss:** tries to make the confidence score equal to the IOU between the object and the prediction when there is one object or close to 0 when there is no object in the cell.

❒**Classification loss:** is calculated by binary cross-entropy loss.

In YOLOv3, they made some modifications to the loss function, confidence loss now uses binary cross-entropy loss to calculated instead of mean square error. The classification loss is now changed into multi-label classification instead of multi-class classification, therefore independent logistic classifiers replaced the softmax for better class prediction.Because some dataset may contains labels that are related to each other.

2.2.3.4 Post Processing

The final component in this detection system is a post-processor. In YOLO, non maximum suppression is used to eleminate duplicate results.

2.2.3.5 Modifications of YOLO

For older version of YOLO, spatial constraint is one major drawback of the algorithm as a grid can analyse only up to two blocks and these two blocks can contain only one class. This results in a reduction in the detection of the nearby objects. It is quite difficult to detect small images in a group image with the help of this algorithm. As it uses the last-stage feature map, which has only the coarse information as it passes through the neural network, the accuracy has a limitation.

Bochkovskiy et al proposed the YOLOv4algorithm with significant changes from the previous version, and much better accuracy. Published in April 2020 it is the latest, most advanced iteration of YOLO, and the first developed by the original authors (Redmon et al). To achieve better accuracy, they designed a deeper and complex network, where they used Dense Block. It contains multiple convolutional layers, with batch normalization, ReLU, after which convolution takes place.

For the backbone of the feature extraction, they used the CSPDarknet-53, which uses the CSP connections along with Darknet-53 from the previous YOLOv3. Spatial Pyramid Pooling (SPP) is used as the neck over CSPDarknet-53 as it increases the receptive field, differentiates the most significant feature and does not cause a reduction in speed. In place of the Feature Pyramid Network (FPN) in YOLOv3, here they used Path Aggregation Network (PANet). For the head, they used the original YOLOv3 network.

Apart from the architecture, it consists of a training strategy to get better accuracy without the extra cost to hardware, which is termed as Bag of Freebies. With these, we can get better performance for "free". Another set of strategies which give better results at low cost, but not completely free, were termed as the Bag of Specials.

With those modification, YOLO is the most commonly used algorithm that is used for detecting natural images as this is the most efficient algorithm for new domains and inputs that are unexpected.

2.2.4 Comparision YOLO, SSD and Faster R-CNN

2.2.4.1 Pascal VOC 2007/2012

As YOLO is not skilled in producing object localizations of high IoU, it obtains a very poor result on VOC 2012. However, with the complementary information from Fast R-CNN (YOLO+FRCN) and the aid of other strategies, such as anchor boxes, BN, and fine-grained features, the localization errors are corrected (YOLOv2).

2.2.4.2 Microsoft COCO

Overall, region proposal-based methods, such as Faster R-CNN and R-FCN, perform better than regression/ classification-based approaches, namely, YOLO and SSD, due to the fact that quite a lot of localization errors are produced by regression/classification-based approaches.

2.2.4.3 Time analysis

Regression-based models can usually be processed in real time at the cost of a drop in accuracy compared with region proposal-based models. Also, region proposal-based models can be modified into real-time systems.

2.2.4.4 Application in Vehicle Detection System

Lecheng Ouyang et al implemented vehicle target detection based on YOLOv3 in complex scenes and showed advantages over traditional target detection algorithms in accuracy and speed. They showed how YOLOv3 can be used for vehicle detection, and it gives an accuracy of 89.16% at 21fps on the VOC dataset.

Jeong-ah Kim et al has put three algorithm into test by the the vehicle type classification for the vehicle type recognition was based on the classification of the Korea Expressway Corporation. They found out that Faster R-CNN may not suitable for real-time application, SSD's accuracy is low and sometimes fails to detect a vehicle while YOLOv4 yields the most efficient results.[7]

2.2.4.5 Other Experiments

Deepa et al compared all three models for real-time tennis ball tracking and from their work, they concluded that SSD is much efficient and comparatively more accurate accurate algorithm with less computation speed for this particular task of detecting the tennis ball tosses. It worth noticing that, the YOLO version they used is version 1, which performs badly with small objects like tennis ball. This is no longer a shortage to the latest version of YOLO [6].

According to [**], when they compared the YOLOv3 algorithm with the SSD, it was shown by evaluation that YOLOv3 had better performance than SSD. If the SSD has an input resolution at 300*300, it would have the same inference as the YOLOv3 at the input resolution 416*416, the precision of YOLOv3 is higher. YOLOv3 is faster than SSD, which is applicable to real-time.

2.2.4.6 Conclusion

Choosing YOLOv4 for ITS system is considered most efficient choice so far, therefore we use YOLOv4 to operate as a detector for our system.

2.3 Metrics and Evaluations

2.3.1 Average Precision(AP) and Mean Average Precision(mAP)

- Recall is the Ratio of the correct predictions and the total number of correct items in the set. It is expressed as percentage of the total correct(positive) items correctly predicted by the model. In other words, recall indicates how good is the model at picking the correct items.

$$Recall = \frac{TP}{TP + FN} \tag{2.30}$$

- Precision is measured over the total predictions of the model. It is the ratio between the correct predictions and the total predictions. In other words, precision indicates how good the model is at whatever it predicted.

$$Precision = \frac{TP}{TP + FP} \tag{2.31}$$

- To get True Positives(TP) and False Positives(FP), we use IoU. Using IoU, we now have to identify if the detection(a Positive) is correct(True) or not(False). The most commonly used threshold is 0.5 — i.e. If the IoU is greater than 0.5, it is considered a True Positive, else it is considered a false positive. The COCO evaluation metric recommends measurement across various IoU thresholds, but for simplicity, we will stick to 0.5, which is the PASCAL VOC metric.

- Since every part of the image where we didn't predict an object is considered a negative, measuring "True" negatives(TN) is a bit futile. So we only measure "False" Negatives(FN) ie. the objects that our model has missed out.

- The average precision (AP) is a way to summarize the precision-recall curve into a single value representing the average of all precisions. The AP is calculated according to the next equation. Using a loop that goes through all precisions/recalls, the difference between the current and next recalls is calculated and then multiplied by the current precision. In other words, the AP is the weighted sum of precisions at each threshold where the weight is the increase in recall.

$$AP = \sum_{n=0}^{k=n-1} [Recalls(k) - Recalls(k+1)] * Precisions(k) \tag{2.32}$$

$$Recalls(n) = 0, Precisions(n) = 1, n = number\ of\ thresholds. \tag{2.33}$$

- The mean Average Precision or mAP score is calculated by taking the mean AP over all classes and/or overall IoU thresholds, depending on different detection challenges that exist.

$$AP = \frac{1}{n} \sum_{n=1}^{k=n} AP_k \tag{2.34}$$

$$AP_k = the\ AP\ of\ class\ k, n = the\ number\ of\ classes \tag{2.35}$$

Chapter 3

MULTIPLE OBJECT TRACKING

3.1 Overview

Multiple object tracking can be viewed as a multi-variable estimation problem. Given an image sequence, we employ s_t^i to denote the state of the i-th object in the t-th frame, $\mathbf{S}_t = (s_t^1, s_t^2, ..., s_t^{M_t})$ to denote states of all the M_t objects in the t-th frame. Let $s_{i_s:i_c}^i = \{s_{i_s}^i, ..., s_{i_e}^i\}$ be the sequential states of the i-th object, where i_s and i_e are respectively the first and last frame in which target i exist, and $S_{1:t} = \{S_1, S_2, ..., S_t\}$ to denote all the sequential states of all the objects from the first frame to the t-th frame. Note that the object number may vary from frame to frame.

Correspondingly, following the most commonly used tracking by detection, or Detection Based Tracking (DBT) paradigm, we utilize o_t^i to denote the collected observations for the i-th object in the t-th frame. $\mathbf{O}_t = (\mathbf{o}_t^1, \mathbf{o}_t^2, ..., \mathbf{o}_t^{M_t})$ to denote the collected observations for all the M_t objects in the t-th frame, and $\mathbf{O}_{1:t} = \{\mathbf{O}_1, \mathbf{O}_2, ..., \mathbf{O}_t\}$ to denote all collected sequential observations of all the objects from the first frame to the t-th frame. The objective of multiple object tracking is to find the "optimal" sequential

states of all the objects, which can be generally modeled by performing MAP (maximal a posteriori) estimation from the conditional distribution of the sequential states given all the observations:

$$\hat{S}_{1:t} = \underset{S_{1:t}}{argmax} P(\mathbf{S}_{1:t}|\mathbf{O}_{1:t}) \tag{3.1}$$

Different MOT algorithms from previous works can now be thought as designing different approaches to solving the above MAP problem, either from a probabilistic inference perspective or a deterministic optimization perspective.

3.2 Categorization

Categorization of MOT bases on: initialization method, processing mode and type of output.

- Initialization method: MOT devides into detection based tracking and detection free tracking
  - Detection based tracking: Given a sequence, type specific object detection or motion detection (based on background modeling) is applied in each frame to obtain object hypotheses, then (sequential or batch) tracking is conducted to link detection hypotheses into trajectories. Detection based tracking focuses on specific objects and its performance relies heavily on accuracy of object detectors. Detection based tracking is more popular as it can deal with new discovered and disappear objects automatically.
  - Detection free tracking requires requires manual initialization of objects in each frame.

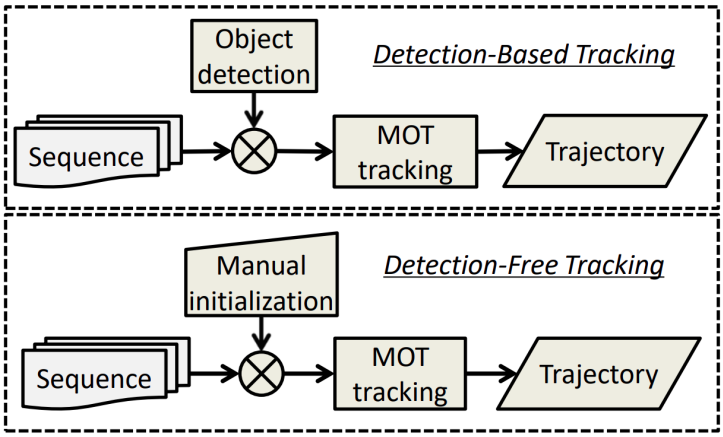

Figure 3.1: Pipeline of detection based tracking and detection free tracking

- Processing mode: based on processing mode, MOT can be divided into online tracking and offline tracking problem.
  - An online model receives video input on a frame-by-frame basis, and has to give an output for each frame. This means that, in addition to the current frame, only information from past frames can be used. Online tracking takes up-to-time observation and updates trajectories on the fly there for it is suitable for real time application.
  - Offline models, on the other hand, have access to the entire video, which means that information from both past and future frames can be used. The task can then be viewed as an optimization problem, where the goal is to find a set of paths that minimize some global loss function. Offline tracking takes a batch of frames to process therefore it products delay results. Since offline trackers have access to more information, one can expect better performance from these models.

Detection free tracking process data sequentially while in Detection based tracking, tracklet or detection response are often associated in batch.

Recently with the rise of deep learning, MOT can also devided into non-deep learning based methods and deep learning based methods.

In deep learning based methods, first CNN-based object detectors are applied such as Faster R-CNN and YOLOv3 to localize all objects of interest in input images. Then in a separate step,they crop the images according to the boxes and feed them to an identity embedding network to extract re-ID features which are used to link the boxes over time. The linking step usually follows a standard practice which first computes a cost matrix according to the re-ID features and Intersection over Unions (IoU) of the bounding boxes and then uses the Kalman Filter and Hungarian algorithm to accomplish the linking task. MOT methods based on deep learning can be further devided into two-stage method and one-stage method.

The main advantage of the two-step methods is that they can develop the most suitable model for each task separately without making compromise. In addition, they can crop the image patches according to the detected bounding boxes and resize them to the same size before estimating re-ID features. This helps to handle the scale variations of objects.The main drawback due to the fact that they are usually very slow because the two tasks need to be done separately without sharing.

For one stage method, the core idea is to simultaneously accomplish object detection and identity embedding (re-ID features) in a single network in order to reduce inference time. However, the accuracy of the one-shot trackers is usually lower than that of the two-step ones.

3.3 MOT Components Overview

3.3.1 Appearance Model

Appearance model includes visual representation and statistical measuring. Appearance model is important cue for affinity computation in MO. Visual representation: local features, region features, Probabilistic Occupancy Map (POM), depth features, Statistical measuring: single cue & multiple cues (five kinds of fusion strategies: Boosting, Concatenating, Summation, Product, and Cascading).

3.3.2 Motion Model

Motion model includes linear & non-linear model. It aims to estimates the potential position of objects in the future frames, thereby reducing the search space.

3.3.3 Interaction Model

Interaction model, also known as mutual motion model, captures the influence of an object on other objects. In the crowd scenery, an object would experience some "force" from other agents and objects. Interaction model includes social force model & crowd motion pattern model.

3.3.4 Exclusion Model

Exclusion model includes detection-level exclusion & trajectory-level exclusion. Exclusion is a constraint employed to avoid physical collisions when seeking a solution to the MOT problem. It arises from the fact that two distinct objects cannot occupy the same physical space in the real world.

3.3.5 Occlusion Handling

Occlusion is perhaps the most critical challenge in MOT. It is a primary cause for ID switches or fragmentation of trajectories. In order to handle occlusion, various kinds of strategies have been proposed such as: part-to-whole, hypothesize-and-test, buffer-and-recover.

3.3.6 Inference Models

- **Probabilistic Inference**
  Approaches based on probabilistic inference typically represent states of objects as a distribution with uncertainty. The goal of a tracking algorithm is to estimate the probabilistic distribution of target state by a variety of probability reasoning methods based on existing observations. This kind of approach typically requires only the existing, i.e. past and present observations, thus they are especially appropriate for the task of online tracking.

  Although the probabilistic approaches provide a more intuitive and complete solution to the problem, they are usually difficult to infer.

3.3.6.1 Bayesian Estimation

Bayesian estimation refers to the task of recursively estimating the state x_t at time k, from observations z_k, where z_k are the measurements obtained up to and including time k The algorithm of estimating the state is divided into two steps, the time update step (also called prediction) and the measurement update step. Commonly used algorithms/filters to perform the Bayesian estimation are different variations of the Kalman Filter (KF), such as Extended KF and Unscented KF. One can also use Monte Carlo samples, so called Particle filters. The choice of filter is dependent on the type of distributions and the nonlinearities and uncertainties of the motion and measurement models.

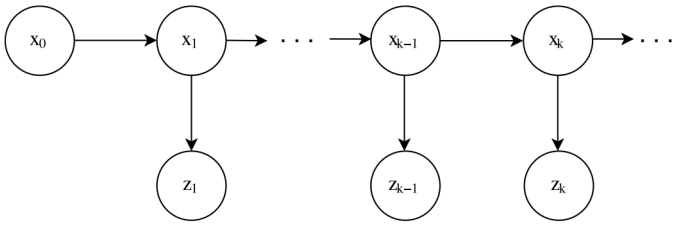

Figure 3.2: Illustration of Bayesian filtering and dependencies on time and measurement updates

3.3.6.2 Time Update

In the time update step, the idea is to predict the state x_k given measurements up to time k - 1, and z^{k-1} this is commonly done using the Chapman–Kolmogorov equation

$$p(x_k|z^{k-1}) = \int p(x_k|x_{k-1})p(x_{k-1}|z^{k-1})dx_{k-1} \tag{3.2}$$

The transition density $p(x_k|x_{k-1})$ is defined from the choice of the motion models $x_k = f(x_{k-1}, v_{k-1})$ where v_k is a random noise process included in order to handle uncertainties and model errors. Namely the time update predicts the motion of the object.

3.3.6.3 Measurement Update

The predicted state is updated with the information from the measurement at time k. The connection between the state and the measurement is given by a measurement model $z_k = h(x_k, w_k)$, where w_k is noise. The measurement model gives rise to the likelihood of the measurement $p(z_k|x_k)$. Since the state is estimated, it is common that the state is described by its distribution. Thus, it also includes information about the uncertainty of the estimation. We denote the prediction distribution $p_{k|k-1}(x_k|z^{k-1})$ and the posterior distribution $p_{k|k}(x_k|z^k)$. Subscript k | k - 1 means that the variable was computed for time k given measurement up to time *k - 1*. Similarity *k | k* where measurements up to time k was used. From Bayes' theorem it follows that:

$$p_{k|k}(x_k|z^k) \propto p(z^k|x_k)p(x_k) \propto p(z_k|x_k)p_{k|k-1}(x_k|z^{k-1}) \tag{3.3}$$

3.3.6.4 Kalman Filter

Kalman filters have a wide variety of applications. Some of the most common applications are navigation and control for vehicles, radar tracking for anti-ballistic missiles, process control, etc.

Kalman Filter is a way of recursively finding the Bayesian estimate $\hat{x_k}$ of the true state x. Thus it is usually more accurate than filters that compute their estimates using only the current measurement. First, at time k, a prediction is made according to the Chapman–Kolmogorov equation. When a new measurement is received, an update of said prediction is calculated based on Bayes' theorem. How much the update relies on the prediction and the new measurement is determined by the Kalman gain, which is a way to weight the two update steps against each other, depending on their respective uncertainty. It can be shown that the (linear) Kalman filter is optimal (in the sense of minimizing the mean square error) in the cases where the noise is Gaussian.

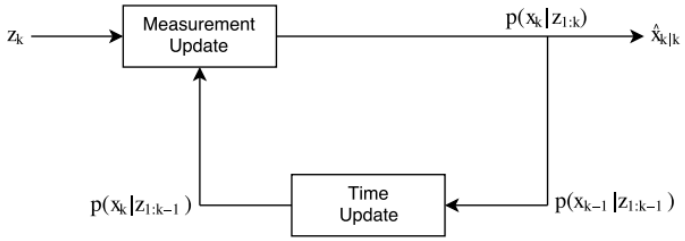

Figure 3.3: Time and measurement update

❒ Time Update

As mentioned, during the time update step a prediction of the state is performed using a motion model $f(\hat{x_{k-1}}, v_{k-1})$. In the case of linear motion

$$x_k = F_{k-1}X_{k-1|k-1} + q_{k-1} \tag{3.4}$$

The time update of the mean and covariance can be computed as

$$\hat{x}_{k|k-1} = F_{k-1}\hat{x}_{k-1|k-1} \tag{3.5}$$

$$P_{k|k-1} = F_{k-1}P_{k-1|k-1}F_{k-1}^T + Q_{k-1} \tag{3.6}$$

where Q_{k-1} is the process noise covariance at time k-1

❒ Measurement Update

Given a new measurement z_k at time k with measurement covariance R_k we can update the predicted state. In the case of linear measurement models and independent Gaussian noise, we can describe the measurement model $z_k = h(\hat{x_k}, w_k)$ as

$$z_k = H_k x_k + w_k \tag{3.7}$$

The update equations of Kalman Filter are

$$\hat{x}_k|k = \hat{x}_{k|k-1} + K_k v_k P_{k|k} = P_{k|k-1} - K_k S_k K_k^T \tag{3.8}$$

where the Kalman gain K_k innovation v_k, the innovation covariance, S_k at time k are

$$K_k = P_{k|k-1}H_k^T S_k^{-1} v_k = z_k - H_k\hat{x}_{k|k-1}S_k = H_k P_{k|k-1}H_k^T + R_k \tag{3.9}$$

The innovation captures the new information that the new measurement brings and the Kalman gain determines how much we should rely on this information.

3.3.6.5 Unscented Kalman Filter

The Kalman filter is derived from linear models, thus is only optimal for this case and not for nonlinear models. If the models are nonlinear, linearization can be used, known as the Extended Kalman Filter (EKF). The EKF works well in many cases, although it may perform poorly if the model is significantly nonlinear within the uncertainties of the models. However, there are other methods of solving the estimation task that are more robust to nonlinearities. The derivation of the Kalman filter contains multiple integrals of the type

$$\int g(x)\mathcal{N}(x; \hat{x}; P)dx = E[g(x)] \tag{3.10}$$

where E denotes the expected value. Furthermore *g(x)* is either a motion or measurement model, which can be non linear. Integrals for example in the Chapman-Kolmogorov equation may look like

$$\int g(x_{k-1})\mathcal{N}(x_{k-1}; \hat{x}_{k-1|k-1}; P_{k-1|k-1})dx_{k-1} \tag{3.11}$$

These integrals can be solved for linear models, however with nonlinear models this is not the case. Thus, these integrals needs to be approximated, one way of doing so is to use the Monte Carlo method. The idea is to generate independent and identicallyy distributed samples $x^{(1)}, x^{(2)}, ..., x^{(N)}$ from its distribution *p(x)*, that we can approximate

$$\int g(x)p(x)dx \approx \frac{1}{N}\sum_{i=1}^{N} g(x_{(i)}) \tag{3.12}$$

However as the Monte Carlo method relies on picking random samples, it might require a lot of samples in order to represent the true distribution. As an alternative, there are so called σ -point methods which use deterministic samples chosen in clever ways to cover a large area of the space even though the number of samples is small. We will focus on one of these, the so called Unscented Kalman Filter (UKF). Two advantages of this filter are that it is efficient since it uses quite few samples, and it also only has one tuning parameter.

❒ Time Update

With a state vector x_k of dimension n, the idea is to generate *2n + 1* σ-point $\mathcal{X}$

$$\mathcal{X}_{k-1}^{(0)} = \hat{x}_{k-1|k-1} \tag{3.13}$$

$$\mathcal{X}_{k-1}^{(i)} = \hat{x}_{k-1|k-1} + \sqrt{\frac{n}{1-W_0}}P_{i,k-1|k-1}^{(1/2)} with i = 1, 2, ..., n \tag{3.14}$$

$$\mathcal{X}_{k-1}^{(i+n)} = \hat{x}_{k-1|k-1} - \sqrt{\frac{n}{1-W_0}}P_{i,k-1|k-1}^{(1/2)} with i = 1, 2, ..., n \tag{3.15}$$

where W_0 is the tuning parameter (namely the weight of $\mathcal{X}$). $P^{(1/2)}$ is the matrix such that

$$P = P^{(1/2)}(P^{(1/2)})^T \tag{3.16}$$

and $P_i^(1/2)$ is its *ith* column. The UKF prediction equations are then given by

$$\hat{x}_{k|k-1} = \sum_{i=0}^{2n} f(\mathcal{X}_{k-1}^{(i)})W_i \tag{3.17}$$

$$P_{k|k-1} = Q_{k-1} + \sum_{i=0}^{2n}(f(\mathcal{X}_{k-1}^{(1)}) - \hat{x}_{k|k-1})(.)^T W_i \tag{3.18}$$

where $W_i = \frac{1-W_0}{2n}$ for i = 1,2,...,n.

❒ Measurement Update

Similarly, in the measurement update step, the σ-points are chosen to

$$\mathcal{X}_{k}^{(0)} = \hat{x}_{k|k-1} \tag{3.19}$$

$$\mathcal{X}_{k}^{(i)} = \hat{x}_{k|k-1} + \sqrt{\frac{n}{1-W_0}}P_{i,k|k-1}^{(1/2)} with i = 1, 2, ..., n \tag{3.20}$$

$$\mathcal{X}_{k}^{i+n} = \hat{x}_{k|k-1} - \sqrt{\frac{n}{1-W_0}}P_{i,k|k-1}^{(1/2)} with i = 1, 2, ..., n \tag{3.21}$$

W_0 is again a tuning parameter. We can now compute the necessary moments

$$\hat{z}_{k|k-1} = \sum_{i=0}^{2n} h(\chi_k^{(i)})W_i \tag{3.22}$$

$$P_{xy} = \sum_{i=0}^{2n}(\chi_k^{(i)} - \hat{x}_{k|k-1})(h(\chi_k^{(i)}) - \hat{z}_{k|k-1}) \tag{3.23}$$

$$S_k = R_k + \sum_{i=0}^{2n}(h(\chi_k^{(i)}) - \hat{z}_{k|k-1})(.)^T W_i \tag{3.24}$$

from which the estimate and its covariance can be calculated

$$\hat{x}_{k|k} = \hat{x}_{k|k} + P_{xy}S_k^{-1}(z_k - \hat{z}_{k|k-1}) \tag{3.25}$$

$$P_{k|k} = P_{k|k-1} - P_{xy}S_k^{-1}P_{xy}^T \tag{3.26}$$

- **Deterministic Optimization**

  As opposed to the probabilistic inference methods, approaches based on deterministic optimization aim to find the maximum a posteriori (MAP) solution to MOT. To that end, the task of inferring data association, the target states or both, is typically cast as an optimization problem. Approaches within this framework are more suitable for the task of offline tracking because observations from all the frames or at least a time window are required to be available in advance. In practice, deterministic optimization or energy minimization is employed more popularly compared with probabilistic approaches.

3.4 Detection Based Tracking Pipeline

3.4.1 Detection Based Tracking

A common methodology is to split the tracking into two phases: prediction of object location, and matching of detections and predictions. That is, for each new frame, the complete tracking model does the following: **(1) Detect objects of interest, (2) Predict new locations of objects from previous frames, (3) Associate objects between frames by similarity of detected and predicted locations.**

3.4.2 Prediction

A multitude of approaches have been suggested for how to predict new locations of tracked objects. Some works include computing the optical flow to determine the new position, or using recurrent neural networks, Kalman filters or particle filters to model the velocity of objects, and with this predict the position in future frames. Filtering problem is tackled in this step and methods such as Kalman filter or particle filter are among best solutions.

3.4.3 Association

The association task consists of determining what detection corresponds to what object, based on the predictions from the previous step, or alternatively if a detection represents a new object. Problems arise when tracked objects leave the scene or are otherwise not detected, new objects enter the scene, the detector produces false positives. This step tackles assignment problems. In the case where the number of detection is equal to the number of tracked objects, this is a case of the assignment problem, in which the goal is to find an optimal matching between two sets of object. Commonly, one set is called agents and the other tasks. There is a cost c_{ij} associated with assigning a task j to an agent i, and the objective is to find an assignment with minimum total cost, such that no agent is assigned more than one task, and no task is assigned to more than one agent. Formally, if A is the set of agents, T the set of tasks, and x_{ij} represents the assignment, taking value 1 if task j is assigned to agent i and 0 otherwise, then the objective is to minimize.

$$\sum_{i \in A} \sum_{j \in T} c_{ij} x_{ij} \tag{3.27}$$

$$]sum_{i \in A} x_{ij} = 1, j \in T \tag{3.28}$$

$$\sum_{j \in T} x_{ij} = 1, i \in A \tag{3.29}$$

$$x_{ij} \geq 0, i \in A, j \in T \tag{3.30}$$

One can also consider the same problem, but with the goal of maximising the cost rather than minimizing it.

3.4.4 Hungarian Method

The Hungarian algorithm, also known as Kuhn–Munkres algorithm, solves the assignment problem in polynomial time, with time complexity $O(n^3)$ where n is the number of agents (and task). The input to the algorithm is a cost matrix C, where $C(i,j)$ is the cost c_{ij} and in four steps the matrix is manipulated to give the optimal matching. In the case where the number of agents is not equal to the number of tasks, C can be padded with rows and columns with large values to make it square. Inversely, if the goal is to find a matching with maximum cost, the matrix can be padded with zeros. Then in the final matching, assignments corresponding to added rows or columns are discarded.

Similarity measures In order to compute the cost matrix C between detections and predictions, we need to define what it means for two bounding boxes to be (dis)similar. A simple way is to compute the intersection over union (IoU), also known as the Jaccard index, between two bounding boxes A and B as

$$IoU(A, b) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{3.31}$$

This has the benefit of combining distance and similarity in size and shape in a nice way, but unfortunately it gives 0 for any two bounding boxes that do not intersect. This means that if the predictions are too inaccurate, all values in the cost matrix could be 0, meaning that all assignments are equally good. Despite this, IoU has successfully been used either as the only measure or in tandem with appearance information.

More sophisticated tracking models combine different similarity measures to calculate the cost, most of which can loosely be separated into three categories: distance measures, shape measures, and appearance measures. These are summed or multiplied with different weights, yielding a single value indicating the overall similarity between bounding boxes. Distance and shape measures look only at the position and geometry of the bounding boxes, whereas appearance features are calculated from the actual pixels within each bounding box.

he three ingredients in the data association step including bounding box IoU, re-ID features and Kalman Filter. These are used to compute the similarity between each pair of detected boxes.We can see that only using box IoU causes a lot of ID switches. This is particularly true for crowded scenes and fast camera motion. Using re-ID features alone notably increases IDF1 and decreases the number of ID switches. In addition, adding Kalman filter helps obtain smooth (reasonable) tracklets which further decreases the number of ID switches. When an object is partly occluded, its re-ID features become unreliable. In this case, it is important to leverage box IoU, re-ID features and Kalman filter to obtain good tracking performance.

3.5 SORT

3.5.1 Estimation Models

The representation and the motion model used to propagate a target's identity into the next frame.They approximate the inter-frame displacements of each object with a linear constant velocity model which is independent of other objects and camera motion. The state of each target is modelled as:

$$x = [u, v, s, r, \dot{u}, \dot{v}, \dot{s}]^T \tag{3.32}$$

where u and v represent the horizontal and vertical pixel location of the centre of the target, while the scale s and r represent the scale (area) and the aspect ratio of the target's bounding box respectively. Note that the aspect ratio is considered to be constant. When a detection is associated to a target, the detected bounding box is used to update the target state where the velocity components are solved optimally via a Kalman filter. If no detection is associated to the target, its state is simply predicted without correction using the linear velocity model.

3.5.2 Data Association

In assigning detections to existing targets, each target's bounding box geometry is estimated by predicting its new location in the current frame. The assignment cost matrix is then computed as the

intersection-over-union (IOU) distance between each detection and all predicted bounding boxes from the existing targets. The assignment is solved optimally using the Hungarian algorithm. Additionally, a minimum IOU is imposed to reject assignments where the detection to target overlap is less than IoU_{min}.

3.5.3 Pipeline

For each frame, the following happens: objects are detected, new locations of already tracked objects are predicted using the predictors, and the detected and tracked objects are matched based on the similarities of the bounding boxes. The predictors are then updated with their associated detections, and the estimated positions are returned as trackings. Additionally, new predictors are initiated for any unmatched detection, and unused predictors are removed.

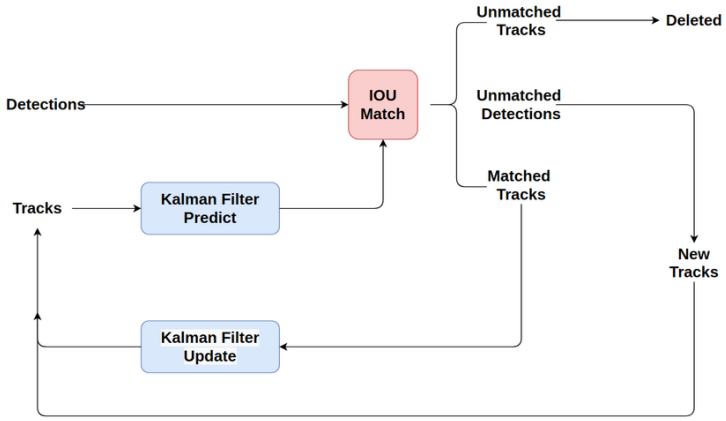

Figure 3.4: Pipeline of SORT

3.5.4 Pros & Cons

While achieving overall good performance in terms of tracking precision and accuracy, SORT returns a relatively high number of identity switches. This is, because the employed association metric is only accurate when state estimation uncertainty is low. Therefore, SORT has a deficiency in tracking through occlusions as they typically appear in frontal-view camera scenes. In addition, SORT leverage Linear Kalman Filter, which not always suits the real world application.

3.6 Deep SORT

DeepSORT is an extension of SORT (Simple Online and Realtime Tracking), which is a simple, popular, and fast multiple-object-tracking (MOT) algorithm. DeepSORT integrates appearance information to improve the performance of SORT by adding one pretrained association metric. The core idea of DeepSORT is to use a traditional single-hypothesis tracking method, which uses the Hungarian method for recursive Kalman filtering and frame-by-frame data association.

3.6.1 Improvements of DeepSORT over SORT

In DeepSORT, the authors solve data association problem based on Hungarian algorithm not only with IOU but also many other features: distance between detection and track, cosine distance between two feature vectors extracted from detection and track as two feature vectors of the same objects are more similar than those of different objects.

3.6.2 Data Association

To solve the frame-by-frame association problem, DeepSORT uses the Hungarian algorithm where both motion and appearance information are considered. This work integrate motion and appearance information through combination of two appropriate metrics.

To incorporate motion information they use the (squared) Mahalanobis distance between predicted Kalman states and newly arrived measurements:

$$d^{(1)}(i,j) = (d_j - y_i)^T S_i^{-1}(d_j - y_i) \tag{3.33}$$

where the projection of the i-th track distribution into measurement space is denoted by (y_i, S_i) and the j-th bounding box detection by d_j.The Mahalanobis distance takes state estimation uncertainty into account by measuring how many standard deviations the detection is away from the mean track location. Further, using this metric it is possible to exclude unlikely associations by thresholding the Mahalanobis distance at a 95% confidence interval computed from the inverse χ^2 distribution. This decision is denoted by an indicator that evaluates to 1 if the association between the *i-th* track and *j-th* detection is admissible.

$$b_{i,j}^{(1)} = \mathbb{1}[d^{(1)}(i,j) \leq t^{(1)}] \tag{3.34}$$

The second metric is integrated into assignment problem. For each bounding box detection d_j an appearance descriptor r_j is computed with $\| r_j \| = 1$. The second metric measures the smallest cosine distance between the *i-th* track and *j-th* detection in appearance space:

$$d^{(2)}(i,j) = min\{1 - r_j^T r_k^{(1)} | r_k^{(i)} \in R_i\} \tag{3.35}$$

A binary variable to indicate if an association is admissible according to this metric is introduced.

$$b_{i,j}^{(2)} = \mathbb{1}[d^{(2)}(i,j) \leq t^{(2)}] \tag{3.36}$$

In combination, both metrics complement each other by serving different aspects of the assignment problem. On the one hand, the Mahalanobis distance provides information about possible object locations based on motion that are particularly useful for short-term predictions. On the other hand, the cosine distance considers appearance information that are particularly useful to recover identities after longterm occlusions, when motion is less discriminative. To build the association problem they combine both metrics using a weighted sum:

$$c_{i,j} = \lambda d^{(1)}(i,j) + (1-\lambda) d^{(2)}(i,j) \tag{3.37}$$

where they call an association admissible if it is within the gating region of both metrics:

$$b_{i,j} = \Pi_{m=1}^{2} b_{i,j}^{(m)} \tag{3.38}$$

Gate matrix will be 1 only when both spatial and appearance gate functions are equal to 1 and otherwise 0 indicating whether (i, j) is a valid match for both spatial and appearance. In each new frame, the new detections are associated with existing tracks using this cost matrix and gate matrix.

The influence of each metric on the combined association cost can be controlled through hyperparameter λ.

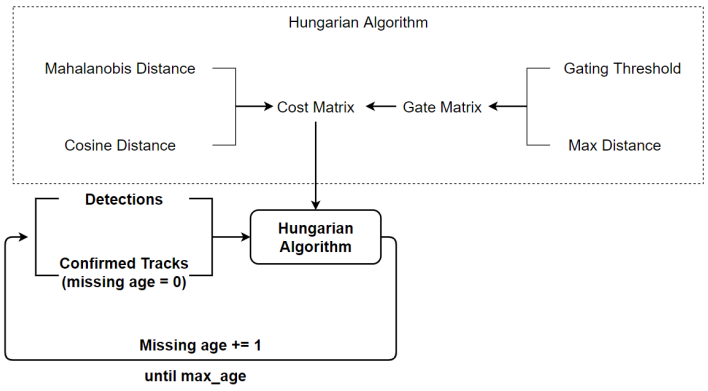

Figure 3.5: Data Association with two integrated metrics

3.6.3 Deep Appearance Feature Extraction

The appearance information is extracted by a convolutional neural network (CNN) trained on re-identification dataset, which is Wide Residual Network. This task is also called Cosine Metric Learning. Although WRN is a shallow neural network with 16 layers, its performance is still very impressive in comparison with other thousand-layers architectures with less training time and inference time.

Name	Patch Size/Stride	Output Size
Conv 1	$3 \times 3/1$	$32 \times 128 \times 64$
Conv 2	$3 \times 3/1$	$32 \times 128 \times 64$
Max Pool 3	$3 \times 3/2$	$32 \times 64 \times 32$
Residual 4	$3 \times 3/1$	$32 \times 64 \times 32$
Residual 5	$3 \times 3/1$	$32 \times 64 \times 32$
Residual 6	$3 \times 3/2$	$64 \times 32 \times 16$
Residual 7	$3 \times 3/1$	$64 \times 32 \times 16$
Residual 8	$3 \times 3/2$	$128 \times 16 \times 8$
Residual 9	$3 \times 3/1$	$128 \times 16 \times 8$
Dense 10		128
ℓ_2 normalization		128

Figure 3.6: Wide Residual Network Architecture

Consine softmax classifier is introduced and applied into training phase of this architecture

$$p(y = k|\mathbf{r}) = \frac{exp(\kappa.\omega_k^T \mathbf{r})}{\sum_{n=1}^{C} exp(\kappa.\omega_n^T \mathbf{r})} \tag{3.39}$$

3.6.4 Matching Cascade

Matching Cascade recursively take track generated in the previous frame to compute cost matrix and then solve assignment problem with cascade. Below is the pseudo code of this matching stategy

Listing 1 Matching Cascade

Input: Track indices $\mathcal{T} = \{1, \ldots, N\}$, Detection indices $\mathcal{D} = \{1, \ldots, M\}$, Maximum age $A_{\max}$

Compute cost matrix $C = \left[c_{i,j}\right]$

Compute gate matrix $B = \left[b_{i,j}\right]$

Initialize set of matches $\mathcal{M} \leftarrow \emptyset$

Initialize set of unmatched detections $\mathcal{U} \leftarrow \mathcal{D}$

for $n \in \{1, \ldots, A_{\max}\}$ do

  Select tracks by age $\mathcal{T}_n \leftarrow \left\{i \in \mathcal{T} \mid a_i = n\right\}$

  $\left[x_{i,j}\right] \leftarrow$ min cost matching $(C, \mathcal{T}_n, \mathcal{U})$

  $\mathcal{M} \leftarrow \mathcal{M} \cup \left\{(i,j) \mid b_{i,j} \cdot x_{i,j} > 0\right\}$

  $\mathcal{U} \leftarrow \mathcal{U} \backslash \left\{j \mid \sum_i b_{i,j} \cdot x_{i,j} > 0\right\}$

end for

return $\mathcal{M}, \mathcal{U}$

Figure 3.7: Listing of Matching Cascade

3.6.5 Track Life Cycle Management

Every time a new detection is successfully associated with an existing track, the detection is added to the track and the unassociated age of the track is zero. When new detections fail to associate with existing tracks in frame f, the new detections are initialized as Tentative tracks. The original Deep SORT algorithm checks that the Tentative tracks are associated with new detections in each of the $(f+1), (f+2), ..., (f+t_{tentative})$ frames. If successfully associated, the track is updated as a Confirmed track. Otherwise, the Tentative track is deleted immediately. As for the existing tracks that fail to associate with new detections in each frame, their unassociated ages will increase by one. If the unassociated age exceeds the max age threshold, the track will also be deleted.

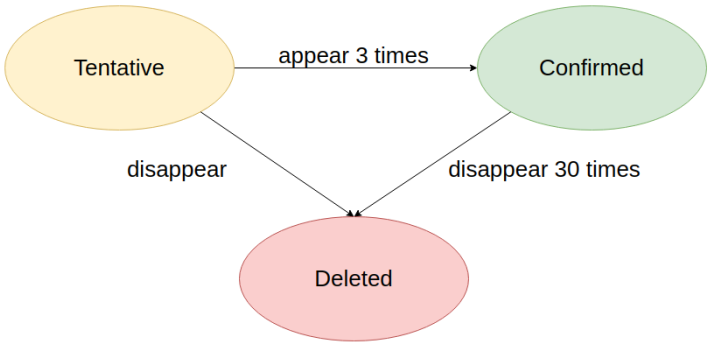

Figure 3.8: Track Life Cycle Management

3.6.6 Pipeline of DeepSORT

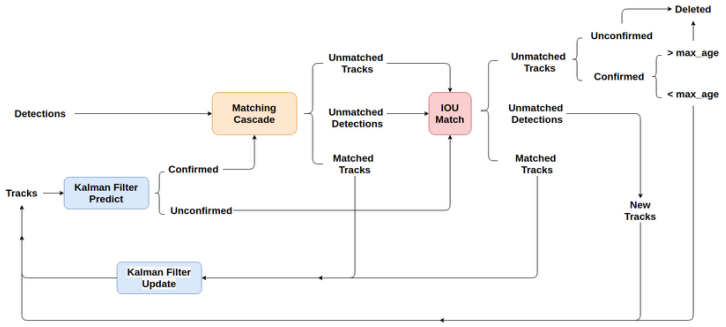

Figure 3.9: Pipeline of DeepSORT

- Firstly, Object Detection Algorithm is used to detect objects in current frame.
- Next, DeepSORT applies Kalman Filter to make prediction of new state base on previous state. This state is assigned with tentative value when initalized. If the value remains in the next 3 frame, the state is changed to confirmed state and will be monitoring in the next 30 frame. Otherwise, the state is released from monitoring.
- Using confirmed track, matching cascade is applied in order to link track with detection results based on distance metrics and appearance metrics.
- Tracks and detections not yet associated will go through next filter layer. Here Hungarian algorithm is used to solve assignment problem with IOU cost matrix.
- Classify detections and tracks prediction.
- Apply Kalman Filter to modify track value from associated detection and initialized new track.

3.7 Evaluations Metrics

3.7.1 Multiple Object Tracking Accuracy(MOTA)

$$[MOTA = 1 - \frac{\sum_t (m_t + fp_t + mme_t)}{\sum_t g_t}] where (m_t, fp_t, mme_t) \tag{3.40}$$

are the number of misses, of false positives and of mismatches respectively for time t. The MOTA can be seen as composed of 3 error ratios.

$$\bar{m} = \frac{\sum_t m_t}{\sum_t g_t} \tag{3.41}$$

the ratio of misses in the sequence, computed over the total number of objects present in all frames,

$$\bar{fp} = \frac{\sum_t fp_t}{\sum_t g_t} \tag{3.42}$$

the ratio of false positives, and

$$\bar{mme} = \frac{\sum_t mme_t}{\sum_t g_t} \tag{3.43}$$

the ratio of mismatches. Summing up over the different error ratios gives us the total error rate (E_{tot}), and $(1 - E_{tot})$ is the resulting tracking accuracy. The MOTA accounts for all object configuration errors made by the tracker, false positives, misses, mismatches, over all frames. It is similar to metrics widely used in other domains (such as the Word Error Rate (WER), commonly used in speech recognition) and gives a very intuitive measure of the tracker's performance at keeping accurate trajectories, independent of its precision in estimating object positions.

3.7.2 Multiple Object Tracking Precision(MOTP)

The Multiple Object Tracking Precision is the average dissimilarity between all true positives and their corresponding ground truth targets. For bounding box overlap, this is computed as

$$MOTP = \frac{\sum_{t,i} d_{t,i}}{\sum_t c_t} \tag{3.44}$$

where ct denotes the number of matches in frame t and $(d_{t,i})$ is the bounding box overlap of target i with its assigned ground truth object. MOTP thereby gives the average overlap between all correctly matched hypotheses and their respective objects and ranges between $td = 50\% \, and \, 100\%$. It is important to point out that MOTP is a measure of localization precision, not to be confused with the positive predictive value or relevance in the context of precision / recall curves used, e.g., in object detection. In practice, it mostly quantifies the localization accuracy of the detector, and therefore, it provides little information about the actual performance of the tracker.

3.7.3 Average multi object tracking accuracy(AMOTA)

For the traditional MOTA formulation at recall 10% there are at least 90% false negatives, which may lead to negative MOTAs. Therefore the contribution of identity switches and false positives becomes negligible at low recall values. In MOTAR we include recall-normalization term $-(1-r) * P$ in the nominator, the factor r in the denominator and the maximum. These guarantee that the values span the entire [0,1] range and brings the three error types into a similar value range. P refers to the number of ground-truth positives for the current class.

$$AMOTA = \frac{1}{n-1} \sum_{r \in \frac{1}{n-1}, \frac{2}{n-1} \ldots 1} MOTAR \tag{3.45}$$

$$MOTAR = \max(0, 1 - \frac{IDS_r + FP_r + FN_r - (1-r) * P}{r * P}) \tag{3.46}$$

3.7.4 Average multi object tracking precision(AMOTP)

Here $(d_{i,t})$ indicates the position error of track i at time t and (TP_t) indicates the number of matches at time t.

$$AMOTA = \frac{1}{n-1} \sum_{r \in \frac{1}{n-1}, \frac{2}{n-1} \ldots 1} \frac{\sum_{i,t} d_{i,t}}{\sum_t TP_t} \tag{3.47}$$

3.7.5 Identification precision(IDP), Identification recall(IDR), IDF1

We use the IDFN, IDFP, IDTP counts to compute identification precision (IDP), identification recall (IDR), and the corresponding F1 score IDF1. More specifically,

$$IDFN = \sum_{\tau \in AT} \sum_{t \in \mathcal{T}_\tau} m(\tau, \gamma_m(\tau), t, \Delta) \tag{3.48}$$

$$IDFP = \sum_{\gamma \in AC} \sum_{t \in \mathcal{T}_\gamma} m(\tau_m(\gamma), \gamma, t, \Delta) \tag{3.49}$$

$$IDTP = \sum_{\tau \in AT} len(\tau) - IDFN = \sum_{\gamma \in AC} len(\gamma) - IDFP \tag{3.50}$$

The fraction of computed (ground truth) detections that are correctly identified.

$$IDP = \frac{IDTP}{IDTP + IDFP} IDR = \frac{IDTP}{IDTP + IDFN} \tag{3.51}$$

IDF1 is the ratio of correctly identified detections over the average number of ground-truth and computed detections. ID precision and ID recall shed light on tracking trade-offs, while the IDF1 score allows ranking all trackers on a single scale that balances identification precision and recall through their harmonic mean.

$$IDF_1 = \frac{2IDTP}{2IDTP + IDFP + IDFN} \tag{3.52}$$

Chapter 4

INFERENCE

4.1 Optimization Methods

4.2 Architecture Optimization

In the heterogeneous computing approach, the workload is divided between the processors with disparate architecture, viz. the CPU and GPU, to bring the best of two together. Some works perform memory-level optimizations to GPU, such as using shared and texture memory, reducing accesses to global memory by using memory access coalescing and kernel fusion, etc. Some works scale the frequency of CPU and/or GPU to tradeoff performance with energy. These works exploit slack in application deadlines or latency-tolerance of end-users to reduce frequency. Many techniques exploit the error-tolerant nature of humans and neural network algorithms to trade off accuracy for achieving efficiency.

- At System and CPU-level, some optimization methods may be listed as CPU - GPU heterogeneous, pipelining, CPU frequency scaling, multithreading on CPU, etc
- At GPU-level, we can name a few such as: GPU frequency scaling, using TensorRT SDK, kernel fusion for reducing global memory transfer, executing kernels concurrently on different CUDA streams, memory access coalescing, use CUDA managed memory, etc
- For approximate computing approaches, there are some examples as: using partial image instead of entire image, dropping unused rows in convolution, quantization, reducing bitwidth/precision, etc

4.3 Algorithm Optimization

Algorithmic optimization methods come with a diversity as there are many works aims at improving efficiency of AI inference. A lot of works try to lowering image resolution, leverage knowledge distillation. Some uses transfer learning, pointwise convolution, tucker decompositions or truncated singular value decomposition. The list is enlonged with CNN pruning and other matrix-related optimizations for such as: converting matrix-vector product into matrix-matrix product, splitting matrix-multiplication in Fully Connected layer, matrix tiling, exploiting matrix sparsity. Some other optimizations may be count such as merge batch normalization, loop unrolling, use of a-priori modules to remove most of the negative samples.

4.4 TensorRT SDK

4.4.1 Definition

NVIDIA ®TensorRT ™is an SDK for high-performance deep learning inference. It includes a deep learning inference optimizer and runtime that delivers low latency and high throughput for deep learning inference applications.

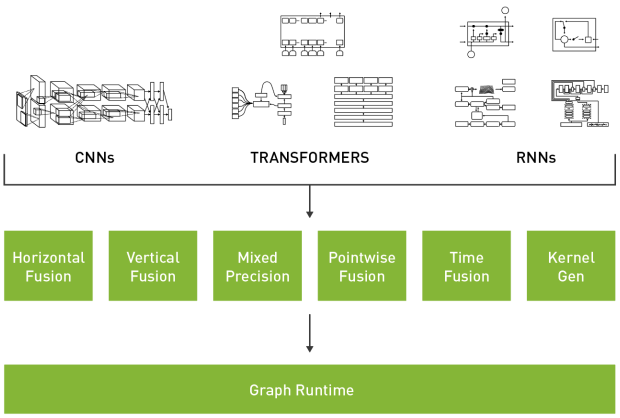

Figure 4.1: TensorRT Pipeline

TensorRT-based applications perform up to 40X faster than CPU-only platforms during inference. With TensorRT, you can optimize neural network models trained in all major frameworks, calibrate for lower precision with high accuracy, and deploy to hyperscale data centers, embedded, or automotive product platforms.

TensorRT is built on CUDA ®, NVIDIA's parallel programming model, and enables you to optimize inference leveraging libraries, development tools, and technologies in CUDA-X ™for artificial intelligence, autonomous machines, high-performance computing, and graphics.

TensorRT provides **INT8** and **FP16** optimizations for production deployments of deep learning inference applications such as video streaming, speech recognition, recommendation, fraud detection, and natural language processing. Reduced precision inference significantly reduces application latency, which is a requirement for many real-time services, as well as autonomous and embedded applications.

With TensorRT, developers can focus on creating novel AI-powered applications rather than performance tuning for inference deployment.

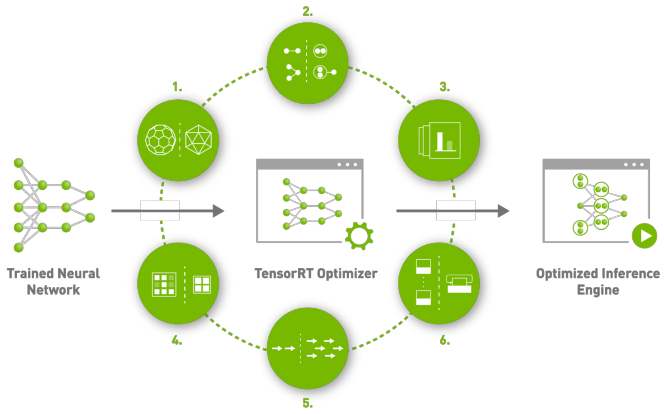

Figure 4.2: TensorRT Inference

1. **Weight & Activation Precision Calibration:** Maximizes throughput by quantizing models to *INT8* while preserving accuracy.
2. **Layer & Tensor Fusion:** Optimizes use of GPU memory and bandwidth by fusing nodes in a kernel.
3. **Kernel Auto-Tuning:** Selects best data layers and algorithms based on target GPU platform.

4. **Dynamic Tensor Memory:** Minimizes memory footprint and re-uses memory for tensors efficiently.
5. **Multi-Stream Execution:** Scalable design to process multiple input streams in parallel.
6. **Time Fusion:** Optimizes recurrent neural networks over time steps with dynamically generated kernels.

4.4.2 What is TensorRT?

The core of NVIDIA ®TensorRT ™is a C++ library that facilitates high-performance inference on NVIDIA graphics processing units (GPUs). It is designed to work in a complementary fashion with training frameworks such as TensorFlow, Caffe, PyTorch, MXNet, etc. It focuses specifically on running an already-trained network quickly and efficiently on a GPU for the purpose of generating a result (a process that is referred to in various places as scoring, detecting, regression, or inference).

Some training frameworks such as TensorFlow have integrated TensorRT so that it can be used to accelerate inference within the framework. Alternatively, TensorRT can be used as a library within a user application. It includes parsers for importing existing models from Caffe, ONNX, or TensorFlow, and C++ and Python APIs for building models programmatically.

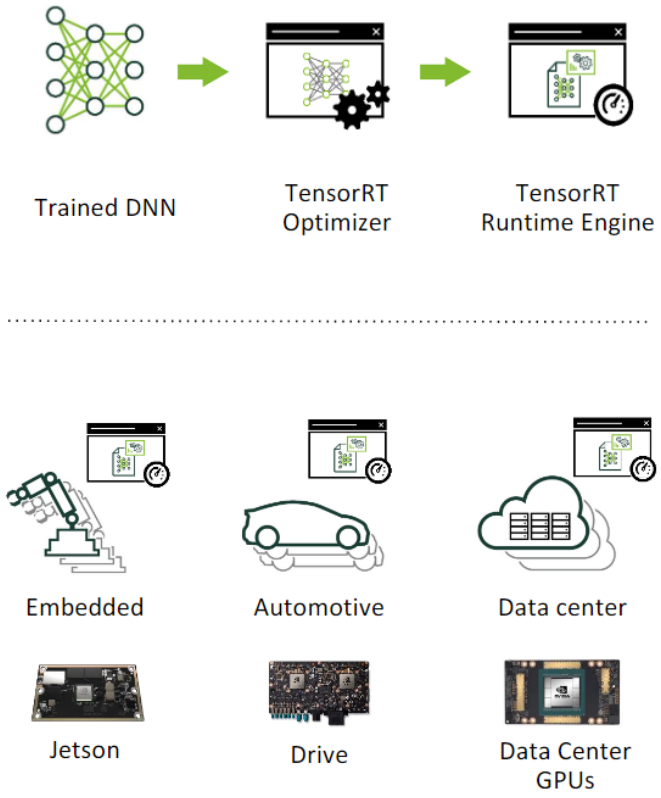

Figure 4.3: TensorRT defined as part high-performance inference optimizer and part runtime engine

TensorRT optimizes the network by combining layers and optimizing kernel selection for improved latency, throughput, power efficiency, and memory consumption. If the application specifies, it will additionally optimize the network to run in lower precision, further increasing performance and reducing memory requirements.

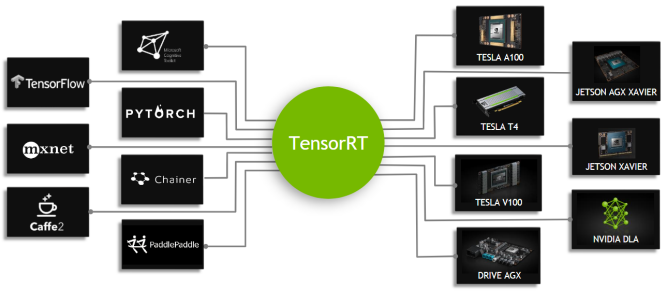

Figure 4.4: TensorRT Application

The TensorRT API includes implementations for the most common deep learning layers. For more information about the layers, see TensorRT Layers. You can also use the *C++ Plugin API* or *Python Plugin API* to provide implementations for infrequently used or more innovative layers that are not supported out-of-the-box by TensorRT.

4.4.3 How does TensorRT work?

To optimize your model for inference, TensorRT takes your network definition, performs optimizations including platform-specific optimizations, and generates the inference engine. This process is referred to as the build phase. The build phase can take considerable time, especially when running on embedded platforms. Therefore, a typical application will build an engine once, and then serialize it as a plan file for later use.

The build phase performs the following optimizations on the layer graph:

- Elimination of layers whose outputs are not used
- Elimination of operations which are equivalent to no-op
- The fusion of convolution, bias and ReLU operations
- Aggregation of operations with sufficiently similar parameters and the same source tensor (for example, the 1x1 convolutions in GoogleNet v5's inception module)
- Merging of concatenation layers by directing layer outputs to the correct eventual destination.

The builder also modifies the precision of weights if necessary. When generating networks in 8-bit integer precision, it uses a process called calibration to determine the dynamic range of intermediate activations, and hence the appropriate scaling factors for quantization.

In addition, the build phase also runs layers on dummy data to select the fastest from its kernel catalog and performs weight pre-formatting and memory optimization where appropriate.

4.4.4 What Capabilities Does TensorRT Provide?

TensorRT enables developers to import, calibrate, generate, and deploy optimized networks. Networks can be imported directly from Caffe, or from other frameworks via the UFF or ONNX formats. They may also be created programmatically by instantiating individual layers and setting parameters and weights directly.

TensorRT provides a C++ implementation on all supported platforms, and a Python implementation on x86, aarch64, and ppc64le.

The key interfaces in the TensorRT core library are:

❒ Network Definition

The Network Definition interface provides methods for the application to specify the definition of a network. Input and output tensors can be specified, layers can be added, and there is an interface for

configuring each supported layer type. As well as layer types, such as convolutional and recurrent layers, and a Plugin layer type allows the application to implement functionality not natively supported by TensorRT.

❒ Optimize Profile

The Builder Configuration interface specifies details for creating an engine. It allows the application to specify optimization profiles, maximum workspace size, the minimum acceptable level of precision, timing iteration counts for autotuning, and an interface for quantizing networks to run in 8-bit precision.

❒ Builder

The Builder interface allows the creation of an optimized engine from a network definition and a builder configuration.

❒ Engine

The Engine interface allows the application to execute inference. It supports synchronous and asynchronous execution, profiling, and enumeration and querying of the bindings for the engine inputs and outputs. A single-engine can have multiple execution contexts, allowing a single set of trained parameters to be used for the simultaneous execution of multiple batches.

TensorRT provides parsers for importing trained networks to create network definitions: **Caffe Parser, UFF Parser, ONNX Parser.**

4.5 Speed Estimation

4.5.1 Mapping

To match track generated from pixel space to real space, we use perspective transform which projects a set of points from one 2D plane to another. In this type of projection we are trying to approximate on a flat surface what is seen by the eye. This means usage of concepts such as vanishing point convergence and the appearance of foreshortening, which means that an object's dimensions along the line of sight are shorter than its dimensions across the line of sight.

Perspective projection comes in several "flavors." A one-point perspective contains only one vanishing point. A typical example is a set of train tracks. If you stand on them and look into the distance, they converge to a single "vanishing point." A two-point perspective has two vanishing points, and a three-point has three vanishing points. Both of those perspectives can be easily seen with sharp-angled buildings, and these perspectives are most often represented in outdoor scenes.

Here for ITS, all we have to do is provide the coordinates of four points — noncollinear, so defining the corners of a quadrilateral — in both coordinate systems. from these four points, can translate a (u,v) pixel location to a (lon,lat) real world location, and vice versa.

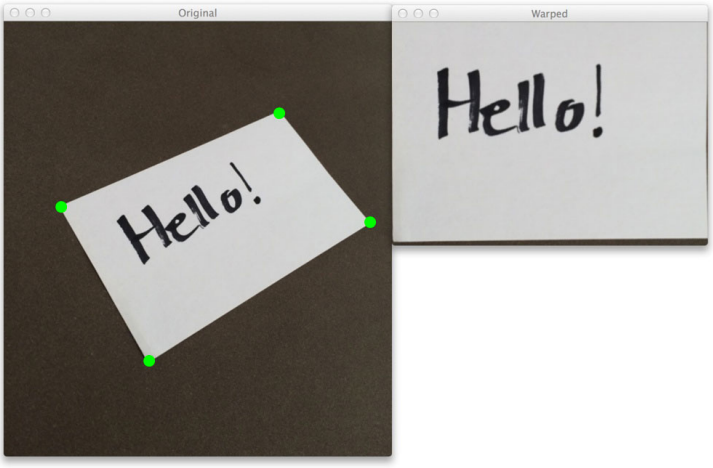

Figure 4.5: Perspective Transformation

Applying this transformation to the (u,v) coordinates of our tracked objects, we can now plot their real world positions as they travel around the roundabout.

4.5.2 Speed Calculation

Using longgitude - latitudes we have just calculated, we perform another coordinate transform, this time converting the vehicle latitude-longitudes into a local Cartesian coordinate system, with units in metres. This then allows us to easily calculate lengths and areas in these units. In order to calculate the vehicles' speeds we need only to divide the distance — calculated from the coordinates above — by the time.

Chapter 5

IMPLEMENTATION

5.1 System Overview

5.2 Hardware Design

5.3 Software Design