POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Resilient Deep Neural Network for FPGA space applications

Supervisors

Prof. LAVAGNO LUCIANO

Prof. CASU MARIO ROBERTO

Prof. LAZARESCU MIHAI TEODOR

Eng. MINNELLA FILIPPO

Candidate

LA CARPIA FRANCESCO

APRIL 2023/2024

Abstract

The effect of radiation on electronic devices can generate errors on various scales that can be catastrophic. In space missions, satellite devices are often used to collect numerous pieces of information on board before transmitting them to Earth. The loss of this information would lead to a high waste of resources and the failure of the entire space mission. The use of resilience techniques is aimed at preventing such errors in a radiation-rich environment such as space. The adoption of machine learning and artificial intelligence techniques in edge computing systems is growing due to their efficiency, reduced latency and enhanced decision-making capabilities. Going into more detail, FPGAs are dataflow accelerators that compute operations in parallel, enabling the execution of numerous computations with low energy consumption. Moreover, being programmable devices, it is possible to reconfigure the device multiple times, ensuring high levels of flexibility. For the reasons just mentioned, convolution operations present in many deep learning algorithms are a perfect fit for such devices. The beginning of this study involved a thorough analysis of the potential hazards that such an environment may pose to an FPGA platform, with a specific focus on Single Event Upsets (SEUs). The manifestation of such errors in SRAM-based FPGAs can lead to malfunctions at varying degrees, often with consequences ranging from moderate to severe. The study then proceeded to analyze the solutions most used to address these types of problems. Solutions include system-level, structure-level, individual cell design of logic elements and FPGA configuration netlist design: the most common methodologies are ECC, TMR and partial or total reconfiguration of the configuration memory. Currently, these solutions provide good resilience to errors caused by radiation. It is possible to combine various techniques that exploit the intrinsic resilience of neural networks with customized hardware and software solutions to maintain performance levels unchanged. Following this preliminary analysis, it was possible to develop a ship detection application using satellite images. The algorithm of choice for this task is SSD, allowing real-time recognition of objects at varying scales by using custom-designed default boxes (or prior boxes), which dimension depends on the sizes of the objects to be identified. The network backbone was quantized using the Brevitas framework with 8-bit quantization for weights and activations and 16-bit for biases. Subsequently, I simulated in software the presence of bit-flip errors at feature maps level, evaluating scenarios where errors result in catastrophic consequences in the network's inference. Error injection was performed using PyTorch Hook functions, allowing access to intermediate modules of the network during the inference process. This approach allows observation of differences in network performance based on the type of bit being flipped (MSB or LSB). The thesis was conducted in collaboration

with AIKO, an emerging company that produces software technologies for space applications.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The use of edge computing devices is continuously growing nowadays, thanks to the ability to process quickly large amounts of data. FPGAs are programmable devices that provide a high level of flexibility, high number of operations per second and low energy consumption per operation.

Artificial intelligence has made embedded systems more efficient due to their ability to use large amounts of data to make predictions efficiently, reducing latency times and expanding the bandwidth through which data are processed. The introduction of these types of technologies has significantly reduced costs, and as a result, they are gaining increasing popularity.

The purpose of this study is to analyze the potential risks arising from the radiation environment, to investigate the techniques used to prevent catastrophic events, and to evaluate the resilience of deep neural networks to such issues.

The thesis was carried out in collaboration with AIKO, an emerging company that produces software technologies for space applications. The study is organized as follows: Chapter 1 analyzes the possible risks and errors that can occur in electronics devices in case of exposure to radiation in space; Chapter 2 examines the techniques currently used to correct or detect potential errors, with particular focus on deep neural network ones; Chapter 3 explains the design of the application; Chapter 4 analyzes the error injection technique.

1.1 Basic Concepts on SRAM-Based FPGA

There are various types of FPGA technologies, depending on their usage:

- EPROM, Flash memory technology: these devices support reprogramming but for a finite number of time.

- Antifuse technology: these devices can be programmed only once (OTP). They act as an open circuit when manufactured: programming involves turning them into a short circuit.
- Static RAM (SRAM) technology: they offer unlimited reprogramming capabilities. SRAM-based FPGAs programmability is not maintained when the device is powered off. Each antifuse is replaced with CMOS technology.

The third technology is the one that is most widely used in various applications due to the ability to change the configuration of memory cells as desired. Furthermore, it enables good computing performance and it is more cost-effective than other devices. The architecture of an FPGA is divided into two parts:

- Processing System (PS):
  - Processor. There can be one or more processors within the architecture.
  - Processing susbsytems. Used to parallelize operations (i.e. vector processors).
  - Cache Memory. Useful for achieving fast memory access.
- Programmable Logic (PL):
  - IOBs: input and output pins of the device.
  - CLBs: these blocks are composed of Flip-Flops (FFs) and Look-Up Tables (LUTs). The combination of these two components enables the designer to derive the required logic function from the input signals.
  - Routing resources: these resources are used to connect CLBs to IOBs. Typically, six pass transistors are utilized to establish interconnections between CLBs. Interconnections can be programmed using simple 1-bit SRAMs or single word or bit. GRM (Global Routing Matrix) represents all interconnections between pass transistors within a routing logic node.

1.2 Space radiation effect on electronic devices

Primary cosmic rays originating from the Sun scatter upon interacting with gaseous and other matter, producing secondary ionizing particles which both create the space radiation environment. In addition, electrons and protons produced by solar fusion and other novas and supernova travel towards Earth's orbit by spreading as the solar wind. Heavy ions then interact with Earth's magnetic field to create Van Allen Radiation belts, which contain trapped electrons in the outer belt and protons in the inner belt. Solar Flares also generate solar energetic particles (SEP), which mainly consist of protons, heavy ions and electrons. Finally, Galactic

Cosmic rays (GCR) represent a significant portion of the total radiation particles, contributing to overall radiation exposure[1]. Therefore, space vehicles and satellites are continuously exposed to a radiation environment: this can lead to degradations of component performance, but also to several malfunctions of electronic and electrical systems. As a result, the lifespan of satellites is shortened, often resulting in critical failures or complete destruction of the space vehicle. In 1994 Clemetine satellite was used to collect images from the Moon and a near-Earth asteroid. The on-board computer sent out a mistaken instruction which resulted into a failure that made it imperative to stop the project. This fault was likely induced by a disturbance arising from exposure to the space radiation environment (SEU). There are different possible consequences that electronic devices could have in radiation conditions [2][3][4][5]:

- CUMULATIVE EFFECT:
    - · Total Ionizing Dose (TID);
    - · Displacement damage defects (DDD).
- SINGLE EVENT EFFECT(SEE):
    - – SOFT ERRORS:
        - · Single Event Upset (SEU);
        - · Single Event Transient (SET).
    - – HARD ERRORS:
        - · Single Event BurnOut (SEBO);
        - · Single Event Gate Rupture (SEGR);
        - · Single Event Latchup (SEL).

1.2.1 Cumulative effect

TID TID mainly occurs due to long-term exposure to ionizing radiation, primarily from particles trapped in the Van Allen belts. It measures the energy absorbed by the device, with the Gray (GY) being the SI unit for radiation dose, where 1 GY equals 100 rads. There are two possible effects of TID:

- Over time, ionizing radiation deposits energy in the oxide region of the MOS transistor, leading to the creation of electron-hole pairs. Due to the low probability of recombination, these pairs migrate within the electric field, causing electrons to separate from the holes. This process results in structural defects.

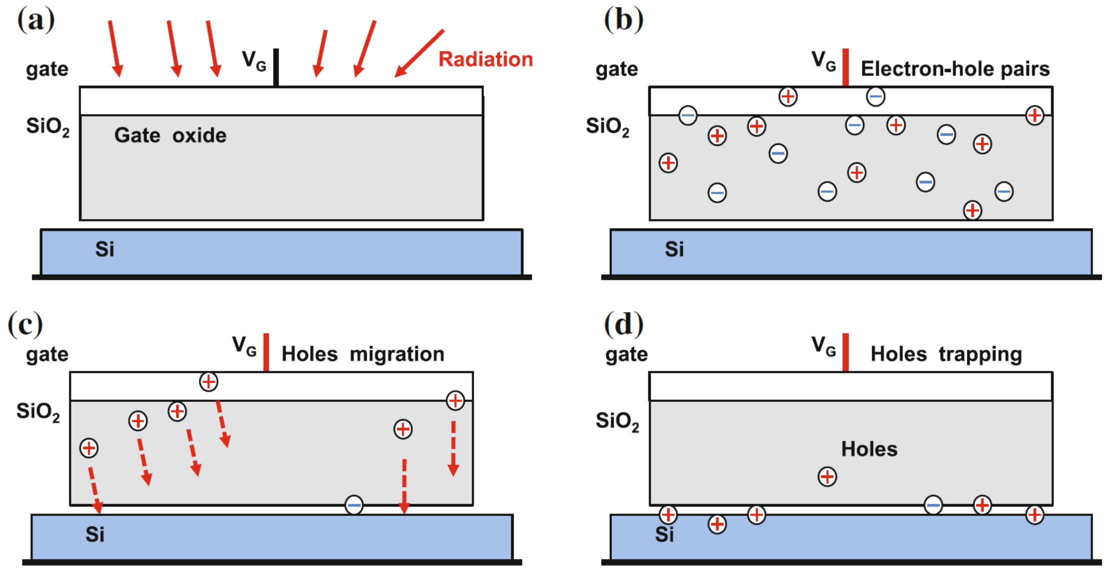

Figure 1.1: TID: **a** radiation, **b** electron–hole pair, **c** holes migration, **d** holes trapping

- It affects the mobility of the electrons in the conductive channel of MOS transistors, with a consequent reduction of their V_{th}: this effect is due to the presence of trapped charges in the SiO_2 and Si/SiO_2 interface.

DDD Intense radiation, such as protons or ions, moves silicon atoms from their normal positions in the crystal structure. This effect happens only if the impact of energy particles overtakes the displacement energy. The main types of defects are vacancy, divacancy, interstitial, Schottky and Frenkel, shown in Figure 1.2. The electrical properties of the component are changed, leading in a degradation of the device performance.

1.2.2 Single Event Effect

A Single Event Effect (SEE) occurs when a single energetic particle generates the presence of charge inside the device. The amount of energy that is conveyed inside the material is defined as LET (linear energy transfer) and it is measured in MeV/μm. LET_{th} defines the maximum energy threshold until a fault is not detected within the component. Errors can be classified into two categories:

- Soft errors: the radiation impact can generate a bit-flip inside a memory cell, flip-flop, latch or register. This error is soft because the circuit is not

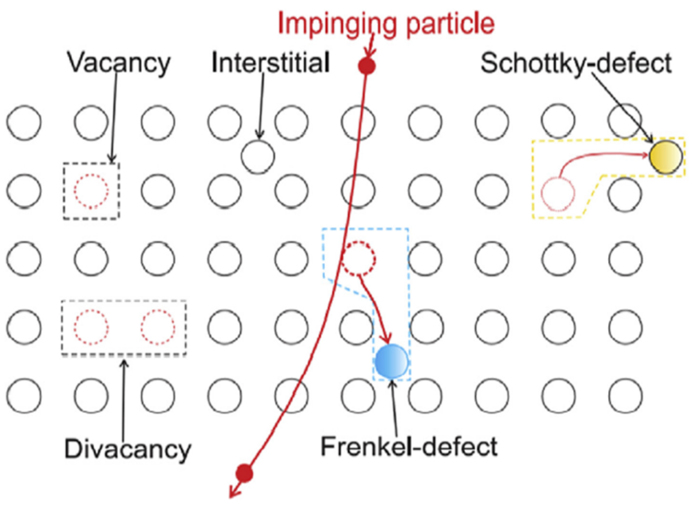

Figure 1.2: DDD: vacancy, divacancy, interstitial, Schottky and Frenkel defects

permanently damaged by the ionizing emission. Some techniques can be applied to prevent the occurrence of a catastrophic event.

- Hard errors: the impact of this kind of error can lead the device not to work properly. In space applications, the probability of having a hard error increases due to the presence of a radiation-rich setting.

SOFT ERRORS

SET Single event transient are errors which occur in combinational circuits. These can result in a soft error if the spike voltage or current signal propagates in a memory element.

SEU The reverse-biased junction is the most sensitive part of transistors. When an ionizing radiation event happens, carriers are rapidly collected by the electric field creating a large current or voltage transient at that node. The diffusion process follows the collection phase. If the event occurs nearby the junction, the probability that a soft error occurs in that node increases. SEU will occur when the charge sensitivity overtakes a certain value Q_{crit}. There are different categories of SEU errors:

- Single bit upset (SBU): the error generates a single bit-flip in the affected cell.
- Multiple bit upset (MBU): if the radiation energy is high, there could be more cell bit-flips.

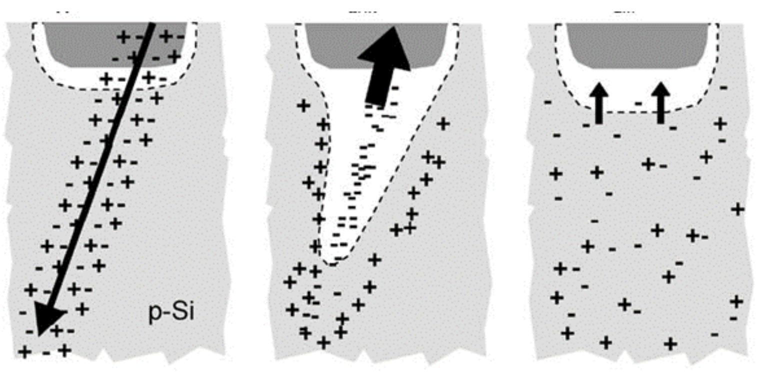

Figure 1.3: Charge accumulation and collection phase in a reverse-biased junction

- Single event functional interrupt (SEFI): in this case a crucial control circuitry area is affected from soft error, like joint test action group (JTAG) or SelectMAP communication port. This implies the interruption of regular tasks performed by the electronic device.

HARD ERRORS

SEL A collision with a particle can activate the parasitic bipolar transistors in the complementary CMOS circuit between the well and substrate, leading to a latch-up. The positive feedback loop in the bipolar junction transistor initiates the spread of high current. The resulting latch-up event may be temporary or permanent, depending on the intensity of the current generated.

SEBO A radiation emission can turn on real BJTs or parasitic BJT structures in a MOSFET. This induces high current generation, as thermal failure in that component section.

SEGR The dielectric region of transistors falls in local breakdown, leading to overheating and destruction of the gate region. Holes from ionization rays increase the electric field across the MOSFET gate oxide to its breakdown point. The leakage current raises and generates thermal failure in the component.

1.3 DNN convolutions in FPGAs

In FPGA devices SEE could affect different region of the device:

- Routing logic: an upset of one of the SRAM cells used for defining the configuration of a single node.

- CLBs LUT: errors impact on the logic component useful to obtain a certain circuit behavior.
- User memory: bit-flip in one of the SRAM cell dedicated for data storage.
- IOBs: I/O drivers will not behave as expected.

The application developed in this study leverages deep neural networks. Therefore, the study aims to investigate how the presence of bit-flip errors can lead to catastrophic errors in the network output, assessing the degradation of metrics after error injection. The study focuses on analyzing potential errors within the feature maps that the network might encounter during convolution operations. As described in [6], the process of computing outputs in a neural network involves defining three tasks:

- Computation task. These are the most resource-intensive tasks, aimed at determining the outputs of each intermediate activation.
- Parameter task. This task ensures access to convolution parameters for the computation task.
- Window buffer task. This task is essential for supplying the computation task with formatted intermediate activations.

Parameters can be stored either in on-chip or off-chip memory, depending on the memory capacity needed and the platform specifications. In the case of using only on-chip memory, direct access to BRAM is possible, significantly reducing latency and power consumption. Ideally, intermediate activations could be stored in on-chip memory to minimize data access time and energy consumption. To accurately evaluate the memory demand, it is crucial to consider the maximum size among the intermediate tensors. However, this is not always feasible as their sizes often surpass the capacity of on-chip memory. In the case study, activations are presumed to be computed concurrently, assuming operation within a pipeline architecture. Thus, computing the maximum size among intermediate tensors is unnecessary. When activations, weights and biases are stored in off-chip memory, direct access to them is facilitated through DMA, bypassing the CPU. Given an intermediate activation tensor, only the relevant parameters needed for the input window to execute the computational task pipeline are required. Therefore, it is essential to store all the lines necessary to generate an input window (Figure 1.4), with each window buffer sized to accommodate the required activations. When the buffer reads an input activation, the previous input window is discarded. The size of a convolution window needed for a convolution operation requires storing data equivalent to $fhi \cdot fwi$ (filter dimension of the i-th layer). As the data for the input window is not contiguous and cannot be directly addressed in the buffer, it is stored

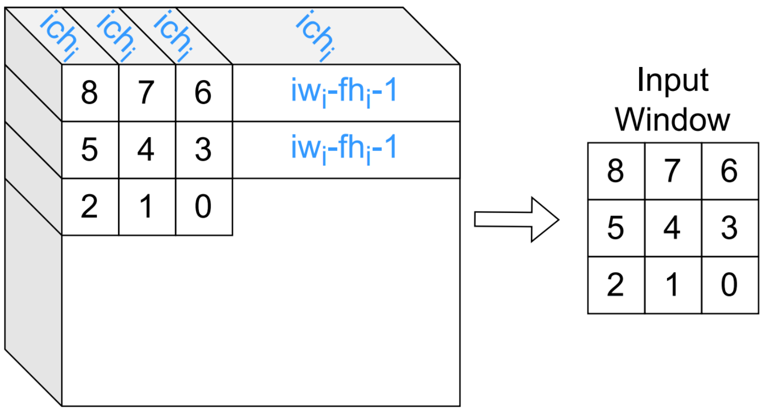

Figure 1.4: Input window generated from intermediate activation tensor

sequentially in a FIFO with only one read port available. To ensure the required bandwidth, the FIFO must be divided into $fhi \cdot fwi$ parts, connected sequentially as illustrated in Figure 1.5. For this reason, we will assume that radiation injection

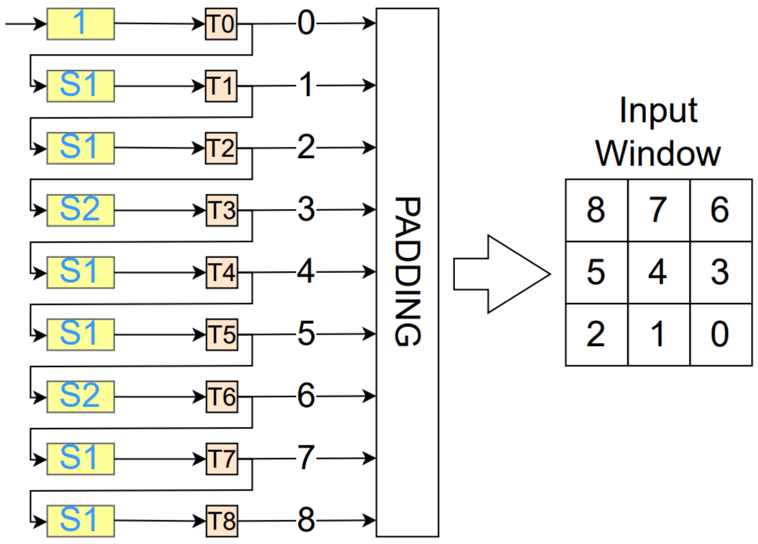

Figure 1.5: Line buffer used for input window operations

will impact the line buffers of the pipeline.

Chapter 2

SEU error correction and detection technique

2.1 Radiation mitigation technique

It is possible to implement countermeasures to mitigate catastrophic errors caused by Single Event Upsets (SEUs) due to radiation. There are various solutions to address such issues, which can be summarized on different levels [7]:

- System level.
  - Reset: apply reset to “persistently flipped” memory elements.
  - Reconfiguration.
  - HW redundancy at system level.
  - Time redundancy at system level.
- Structure level.
- Cell layout (RHBD) level.
- Netlist design level.
  - EDAC: parity, checksums, Hamming codes.
  - Fault masking: TMR.
  - Deadlock free: FSM.

To guarantee radiation effect protection there are also some drawbacks/costs:

- More silicon area, less integration.

- Lower speed.
- More weight.
- Higher power consumption.
- Higher design complexity, longer development times.
- Export constraints dependencies.
- Higher technology prices (expensive components, tests and tools)

However, the cost of losing on-board experiments or the entire satellite is much higher.

2.1.1 System level

RESET A radiation particle affects a specific cell within a logic element of the component (CLBs). In such cases, it is preferable to apply a reset to the relevant memory elements susceptible to this risk, allowing them to return to their original values.

RECONFIGURATION The bitstream generated from the synthesis of an FPGA establishes the configuration of logic gates, routing connections and other internal components. The presence of a ionizing particle may lead to a bit-flip in CRAM memory cells. For this reason, it is necessary to reconfigure the memory to its original state using an internal or external backup memory resilient to radiation [8]. There are two types of reconfiguration:

- Static Reconfiguration. The device is reconfigured only when powered on. This type of reconfiguration is employed in long-term use applications, where frequent changes are not necessary.
- Dynamic Reconfiguration. It involves modifying the device while it is operational, ensuring uninterrupted system performance. This technique is commonly referred to as scrubbing.

To prevent radiation from causing catastrophic errors, it is essential to employ the second type of reconfiguration. The protocols facilitating reconfiguration are JTAG, SelectMAP, ICAP or PCAP for *Ultrascale* or *Ultrascale+* devices. Scrubbing can be further classified in different ways:

- Scrubber circuitry location:

  - Internal scrubbing. Scrubbing is performed using a controller that accesses an external memory to read the golden values through a configuration interface (Figure 2.1). This technique is employed to detect and correct single or multiple bit-flip errors (SEC-DED).

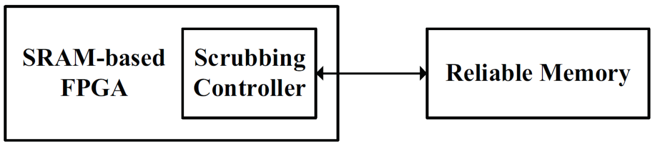

Figure 2.1: Internal scrubber architecture

  - External scrubbing. The controller is external to the SRAM-based FPGA. External scrubbers have better performance than internal one. They typically operate in blind or readback mode (Figure 2.2).

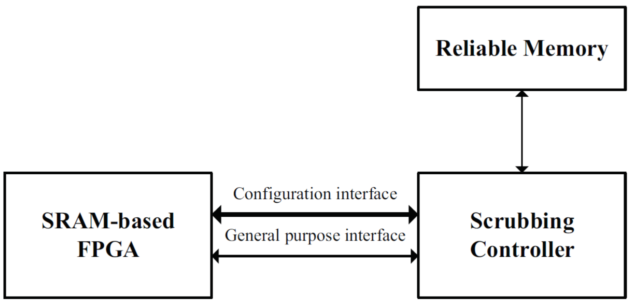

Figure 2.2: External scrubber architecture

  - Hybrid scrubbers. They have both an external and an internal controller. The second one corrects single bit-flip errors and detects multiple bit-flip errors, while the external controller handles the correction process (Figure 2.3).

- Operation type:
  - Blind scrubbing. Golden data is read from the radiation-resilient memory and rewritten into CRAM after a certain interval of time.
  - Readback scrubbing. Golden data and CRAM values are read by the scrubber and compared through a controller using ECC. If different values are detected, the original values are rewritten into the SRAM configuration frame.
  - Error-driven scrubbing. Reconfiguration starts when an error is detected.

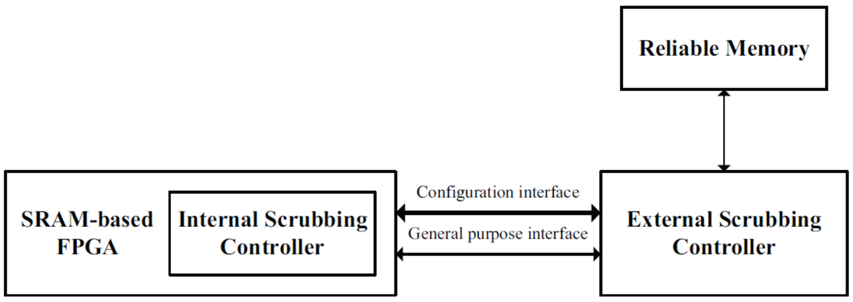
Reliable Memory
SRAM-based FPGA
Internal Scrubbing Controller
Configuration interface
General purpose interface
External Scrubbing Controller

Figure 2.3: Hybrid scrubber architecture

  - Task-driven scrubbing. Reconfiguration occurs when critical tasks are executed. This technique is used when minimizing performance impact on the system is desired.

- Scrubber implementation:
  - Hardware scrubbing.
  - Software scrubbing.

- Scrubbing granularity:
  - Partial reconfiguration (configuration memory frame or design modules).
  - Total reconfiguration (device).

HW REDUNDANCY AT SYSTEM LEVEL To ensure continuos system functionality, if one of the hardware components is damaged due to radiation, a redundant component can take over and continue operations. This system architecture is known as Dual Modular Redundancy (DMR), where the output of two parallel blocks is compared (Figure 2.4). There are two primary types of DMR systems:

- Active standby. Two blocks work in parallel, and their output signals are compared using a comparator. When an error is detected in the main block, the other module takes over.

- Cold standby. In this case, only one of the two blocks is active. The second is activated only when an error is recognized in the primary block, recovering data from the context. From an energy consumption perspective, this solution is better than the previous one, but it requires a certain time interval to restore the system's functionality.

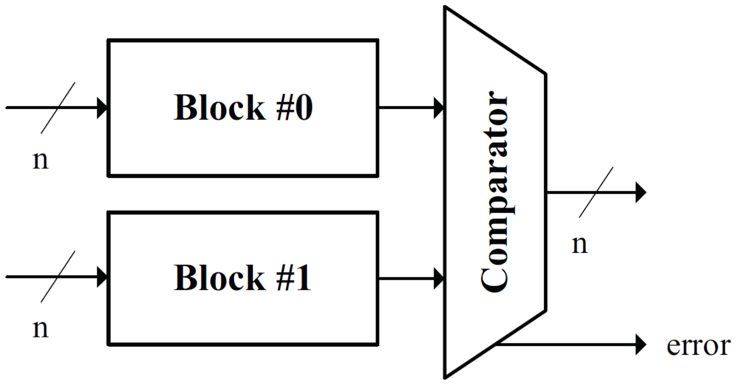

Figure 2.4: DMR architecture

TIME REDUNDANCY AT SYSTEM LEVEL In the event of radiation-induced damage to a system block, the processor captures the information from the memory write instruction but prevents it from being executed by bypassing the memory write enable signal. Subsequently, the processor re-executes all instructions from the last checkpoint. The Comparison and Retry (CR) mechanism compares the address and data with those recorded from the first execution. If they match, the main memory is updated, and the process continues. Otherwise, if a discrepancy is detected, a fault is identified and a mismatch signal is triggered [9].

2.1.2 Structure level

In space applications, radiation shielding systems are often employed to reduce the probability that soft and hard errors cause severe damage to the system. The selection of the material is crucial, considering factors such as system weight, costs and the level of protection these systems provide. Aluminum is a widely used material that fully satisfies the requirements for such applications.

2.1.3 Cell layout (RHBD) level

This technique is employed to mitigate the sensitivity to Single Event Upsets (SEU) in memory cells within programmable platforms. The design occurs at transistor level within the logic gates of the FPGA. Various techniques are employed to prevent bit-flips resulting from temporary charge overflows: most involve the introduction of feedback loops between transistors and additional memory stages. The use of these techniques is often directly implemented by vendors, simplifying the work for IC designers (RHBP).

2.1.4 Netlist design techniques

PARITY It is possible to employ single-bit error correction techniques. The introduction of a parity bit enables the recognition and correction of individual bit-flip errors. The additional bit is positive or negative depending on the implementation.

HAMMING CODE It is a correction mechanism similar to the previous one, which allows to identify the error position when a single bit-flip happens. Given k bits used to represent a number, a predefined number of bits is added so that the total number of bits is equal to n. The algorithm then introduces $n - k$ control bits. This approach enables the correction and detection of single errors, and with the addition of another parity bit, it can detect double errors. Hence, it becomes a SEC-DED algorithm.

CHECKSUM This method is used to validate the accuracy of data blocks. Similar to the parity bit, the error position is not identified. Checks of this nature can be performed both in software and hardware. The calculations for checksum verification must be executed quickly and with minimal overhead [10][11].

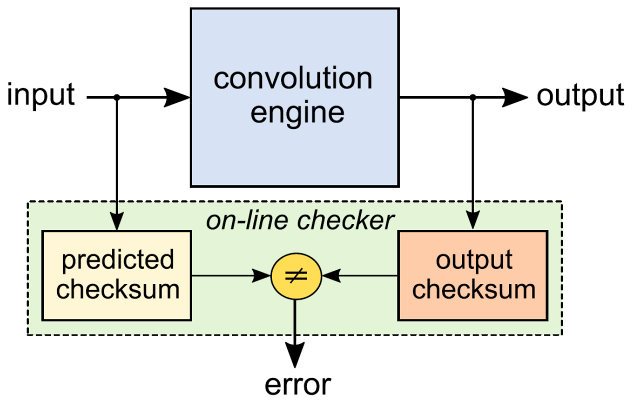

Figure 2.5: Checksum error detection mechanism

TRIPLE MODULAR REDUNDANCY To improve resilience against Single Event Upsets (SEU), TMR can mask the presence of bit-flip errors. When a fault happens Triple Modular Redundancy (TMR) involves tripling the number of blocks to ensure that a correct output is obtained [12]. This system can be implemented on different scales, depending on the type of the triplicated logic block:

- Local-TMR: Only flip-flops constituting the registers are triplicated. The majority voter is placed at the end of the structure (Figure 2.6).

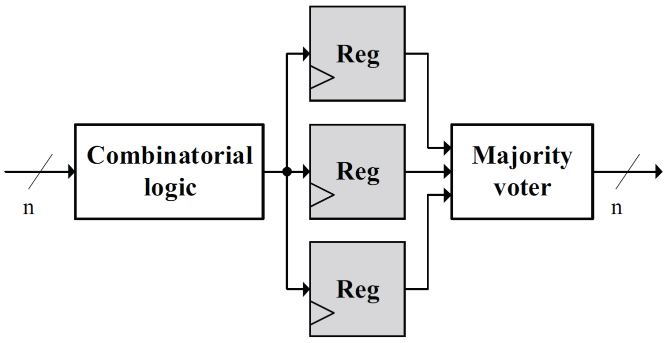

Figure 2.6: Local-TMR architecture

- Block-TMR: Combinational logic or flip-flops are replicated (Figure 2.7).

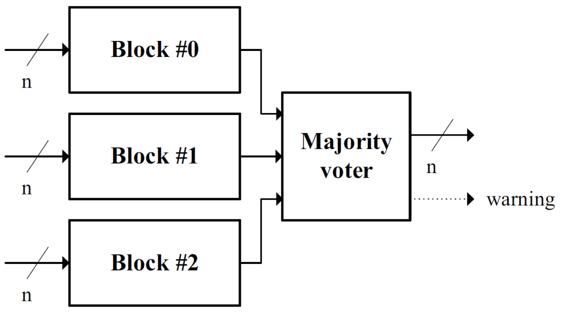

Figure 2.7: Block TMR architecture

- Distributed-TMR: all functional logic is triplicated except for global routes: clocks, resets and high-fanout enables (Figure 2.8). When using counters an additional feedback loop is introduced. This allows a direct comparison between the majority voter's output and the input of the combinatorial logic. It is useful to skip the resynchronization of the entire system to match the other two signals (Figure 2.9).

- Global-TMR: the entire design is triplicated, including all global routes (Figure 2.10).

FSM The use of a Finite State Machine can be anticipated to ensure that the system can recover to its original working state after the occurrence of a Single Event Upset (SEU).

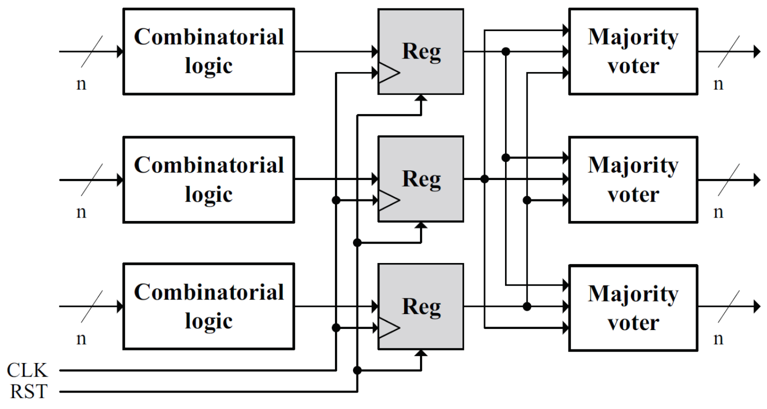

Figure 2.8: Distributed-TMR architecture

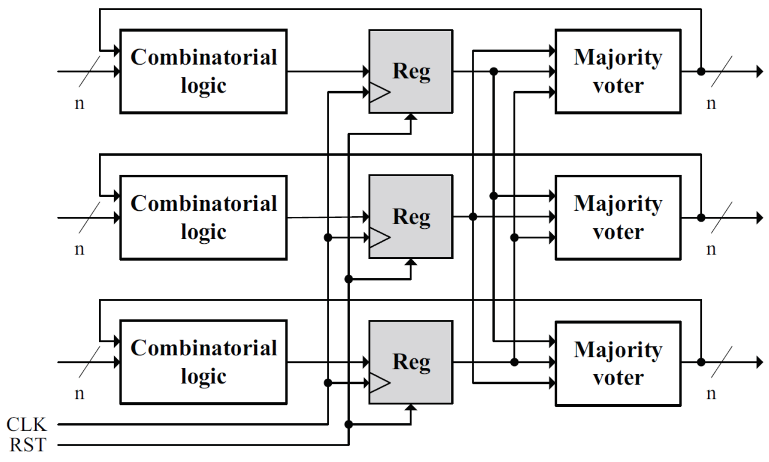

Figure 2.9: Distributed TMR architecture with feedback loop

2.2 DNN resilient techniques

In recent years, the use of artificial intelligence has become increasingly widespread in various applications. Neural networks exhibit internal resilience due to various factors:

- Weight Distribution. Neural networks learn from data and adjust their weights during the training process. Since the weight distribution can be spread across many neurons and many weights, a bit-flip error on a single pixel can have a limited impact on the overall capacity of the network.
- Redundancy and Parallelism. Neural networks often contain multiple neurons performing similar or parallel functions. This redundancy can contribute to

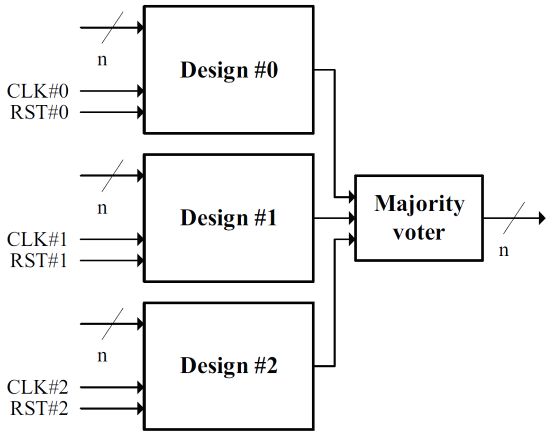

Figure 2.10: Global-TMR architecture

resilience, as an error in one neuron can be compensated for by the output of nearby neurons or alternative paths in the network.

- Activation Function. The activation function used in neurons can introduce a certain robustness. Activation functions like Rectified Linear Unit (ReLU) can mitigate the impact of small errors, as they respond only to positive values and ignore negative values.

Below are indicated the most commonly used resilience methodologies in applications based on neural networks [13].

FAULT-AWARE TRAINING During the training process, intentional errors are introduced to enhance the network's robustness and ensure correct output generation. However, the applicability of this technique depends on the completeness of the dataset being utilized. Moreover, the incurred overhead during error injection can be substantial, particularly when dealing with a significant number of corrupted chips [14].

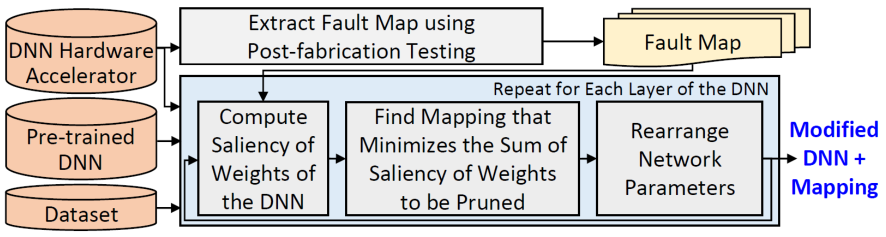
DNN Hardware Accelerator
Pre-trained DNN
Dataset
Extract Fault Map using Post-fabrication Testing
Fault Map
Repeat for Each Layer of the DNN
Compute Saliency of Weights of the DNN
Find Mapping that Minimizes the Sum of Saliency of Weights to be Pruned
Rearrange Network Parameters
Modified DNN + Mapping

Figure 2.11: Fault-aware training process

QUANTIZATED DNN As we said before, DNNs exhibit intrinsic resilience to bit-flip errors. In [15], it has been demonstrated that using reduced-precision parameters and activations enhances the network's resistance to potential bit-flip errors, reducing the bit error rate (BER) by approximately 50%. However, more aggressive quantization may lead the network to sacrifice resilience and accuracy.

FAULT-AWARE PRUNING It has been proven that in the presence of permanent errors arising from soft errors in the hardware architecture of DNN accelerators, it is feasible to eliminate error-affected operations using specific hardware design techniques. One such example involves reinforcing the MAC operators in a systolic array-based DNN hardware. While this approach primarily focuses on addressing local errors, it can be combined with fault-aware training for superior performance. Nonetheless, the overall performance improvement achieved through this combination may not be exceptionally high.

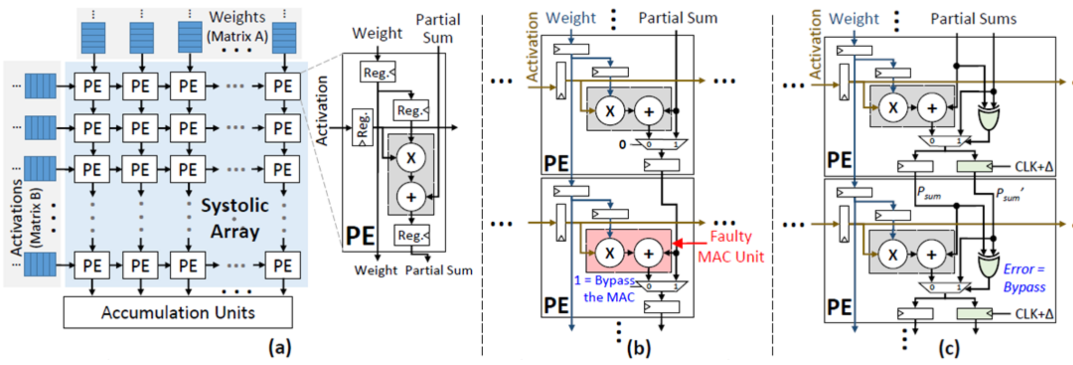

Figure 2.12: Fault-aware pruning process in a systolic array-based DNN hardware

FAULT-AWARE MAPPING This method proves effective in mitigating the propagation of hardware errors during computations. It entails strategically assigning computations with the most crucial parameter values, typically those with higher magnitude weights, to MAC units susceptible to errors. In this way, the drop in accuracy is minimized, especially when employed with fault-aware pruning techniques [16].

RANGE RESTRICTION In the computation of a neural network's output, the presence of bit-flip errors at the level of parameters and intermediate activations can lead to catastrophic outputs throughout the entire network. Therefore, it is crucial to monitor, where possible, the outputs generated by intermediate activations to ensure that convolution operations do not introduce errors. In the case of

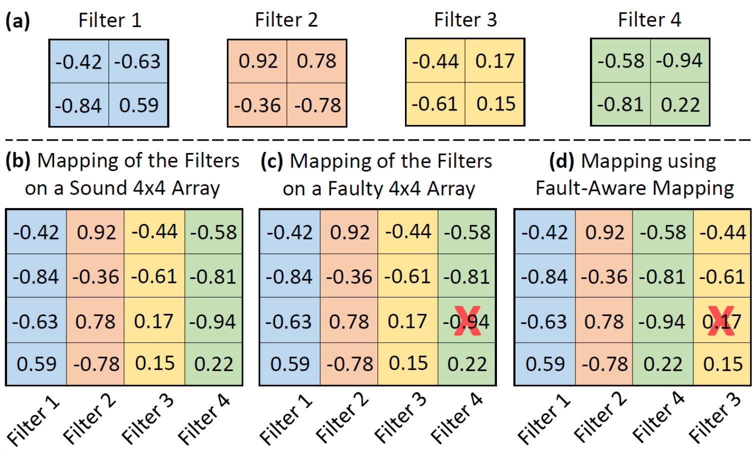

Figure 2.13: Fault-aware mapping working principle

certain layers, it is feasible to assess the range of values within which intermediate activations fall, either using statistical methods or by referring to the type of activation employed. When one of the intermediate activations exhibits bit-flip errors resulting in catastrophic outputs in the DNN, it becomes necessary to perform an activation clipping operation. This prevents permanent errors in user memory from propagating throughout the network. Various types of ranges can be chosen to effectively reduce the overall interval of activations [17][18].

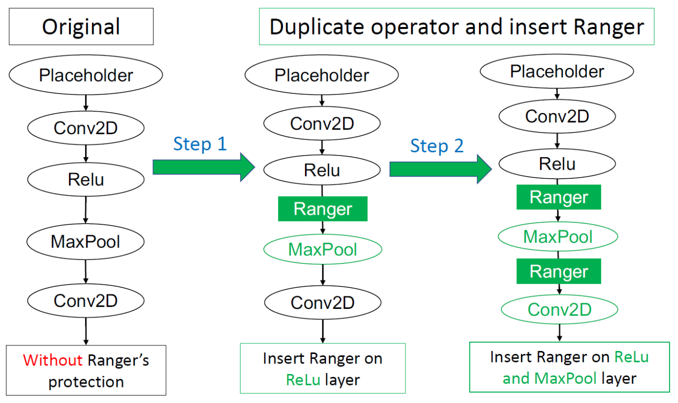

Figure 2.14: Range restriction layer insertion

RADIATION HARDENING This approach involves utilizing hardware components that are more resistant to bit-flip errors, leading to the occurrence of errors in memory cells due to radiation toward values to which DNNs are resilient (i.e., SRAM cells more frequently result in a '0' than a '1').

SELECTIVE TMR Another possible solution using DNN is Selective Triple Modular Redundancy (STMR), implemented in software as described in the paper [19]. This type of solution aims to reduce the number of triplications within the network, creating a redundant system where the network is more sensitive to errors. During execution, the system can dynamically select the correct response among the three copies based on comparisons or voting mechanisms.

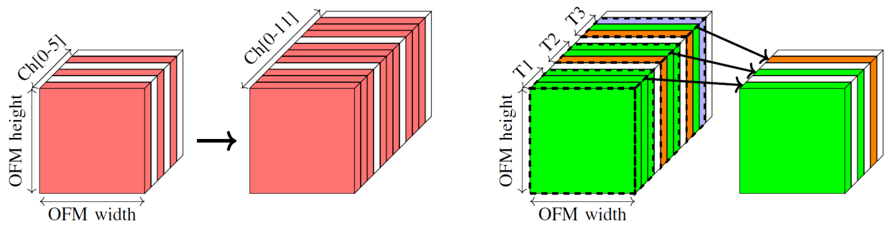

Figure 2.15: STMR mapping and majority vote

Chapter 3

Application and dataset description

The application in use aims to detect objects within satellite images. It is assumed that the images are captured by an onboard camera, which directly passes the input images to the FPGA platform without involving the processor in data computations. The algorithm employed for object detection is the Single Shot Multibox Detector (SSD) method, capable of real-time object recognition, unlike algorithms such as YOLO. The dataset under consideration consists of satellite images where objects to be detected are ships. The dataset was created for image segmentation tasks [20]. Initially, the dataset comprised 192,556 images, which could either contain or not ships. Due to its large size training, validation and test datasets were initially reduced by excluding images that do not contain objects, decreasing the total size to 42,556 elements. Each image has a size of 768x768 pixels. Subsequently, the dataset was further reduced for two reasons:

- The algorithm used requires the recognition of objects within the image and their size must be sufficiently large to prevent the regression loss from diverging during training.
- The dataset images contain objects with bounding boxes that were generated incorrectly, especially for objects with dimensions of few pixels.

For these two reasons, the size of the total dataset was reduced by considering only images with bounding boxes exceeding a certain threshold. The threshold considered is set to 23 for both sides of a bounding box. This threshold allows the recognition of ships with dimensions larger than 10 pixels. The total dataset now consists of 28,715 images, with 24,670 used for training, 3,084 for validation, and 961 for testing. The test dataset has been divided into three parts:

- Test 1: 462 images. All images contain at least one object, useful to evaluate the network's performance in ship recognition.
- Test 2: 499 images. In this case, only a portion of the images actually contains objects (95 in total). The remaining images are used to assess the network's performance in real-world scenarios, where a camera captures landscapes without ships.
- Test 3: 961 images. A comprehensive test combining the datasets from the previous two tests.

The image size has been reduced to 512x512 pixels to minimize the dimensions of intermediate activations and consequently the parameters of the network (the number of parameters in the classification and regression headers is significantly reduced). This reduction ensures excellent performance even for very small objects.

3.1 SSD algorithm

The SSD algorithm [21] achieves real-time object recognition by using default bounding boxes, also known as prior boxes. These prior boxes need to be carefully designed for effective object recognition. The generated prior boxes are distributed within intermediate activations. Specifically, the SSD algorithm can generate up to a maximum of six default boxes for each pixel of the intermediate activation. The default boxes are distributed only within the layers considered for extracting object features, as depicted in Figure 3.1 .
The selection of layers for feature map extraction is complex and depends on the

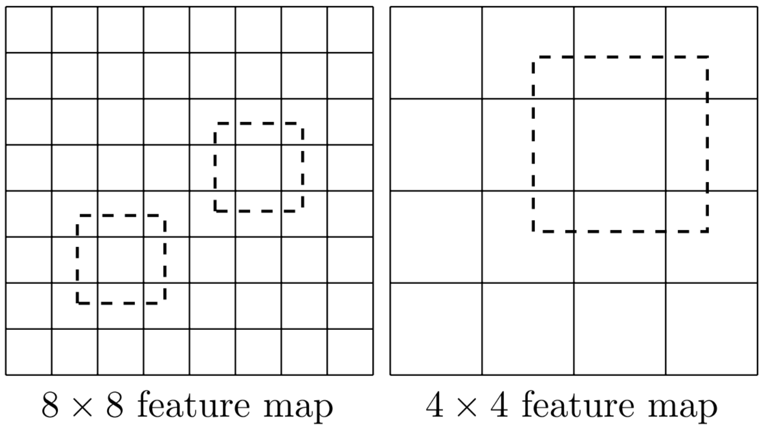

Figure 3.1: Example of the distribution of prior boxes over feature maps.

dataset being used. Extracting low-level intermediate activations allows the capture of finer features, like edges and textures, which are useful for precisely locating objects. Extracting high-level intermediate activations captures more abstract and complex concepts, such as the overall context of the image. The design of prior box dimensions is customized and depends on the size of objects within the images of the dataset being used. Typically, to enhance application performance, extra convolutional layers are added to the base network to improve the extraction of high-level information. The scale, denoted as s, is defined as the ratio of the size between the considered feature map and the original image. For the design of prior boxes, the following formula is used:

$$s_i = s_{min} + \frac{s_{max} - s_{min}}{m-1} \cdot i \tag{3.1}$$

Where s_i is the scale of the prior box belonging to the i-th feature map, s_{min} and s_{max} are the minimum and maximum scales of the prior boxes, and m is the total number of feature maps from which information is extracted for object recognition. The dimension of the prior boxes was slightly readapted to achieve better performance.

```
SSDBoxSizes = collections.namedtuple('SSDBoxSizes', ['min', 'max'])
SSDSpec = collections.namedtuple('SSDSpec', ['feature_map_size', '
    shrinkage', 'box_sizes', 'aspect_ratios'])
specs = [
    SSDSpec(64,8, SSDBoxSizes(15, 41) ,[2]),
    SSDSpec(32,16, SSDBoxSizes(41, 92) ,[2, 3]),
    SSDSpec(16, 32, SSDBoxSizes(92, 166), [2, 3]),
    SSDSpec(8, 64, SSDBoxSizes(166, 239), [2, 3]),
    SSDSpec(4, 128, SSDBoxSizes(239, 313), [2, 3]),
    SSDSpec(2, 256, SSDBoxSizes(313, 460), [2, 3]),
    SSDSpec(1, 512, SSDBoxSizes(460, 534), [2, 3])]
```

It can also be demonstrated that using default boxes along the image borders as shown in Figure 3.2 improves the network's performance metrics.

The training phase is preceded by an augmentation stage, where a randomly selected subset of the entire dataset undergoes transformations such as cropping, expansion and normalization, as illustrated in Figure 3.3.

During the training phase, the algorithm assigns the prior boxes to a specific class and reduces the relative distance (location) between the selected ground truth bounding boxes and the predicted ones by shifting them within the feature maps. Regression and classification losses are respectively used to calculate the position and class of the identified object.

After the assignment phase, most of the prior boxes turn out to be negative. To address this imbalance, only a portion of the total negative bounding boxes is considered during training, in a 1:3 ratio. The prediction phase involves the use of

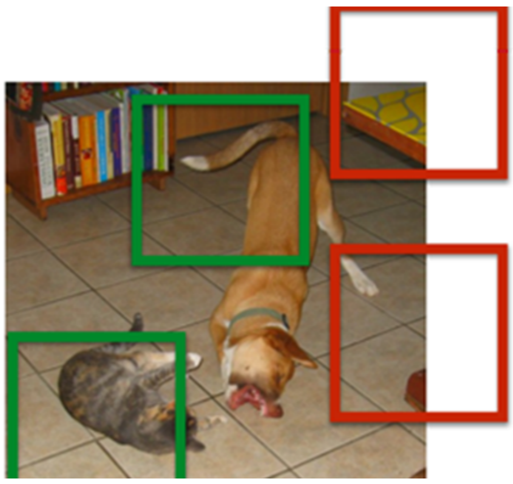

Figure 3.2: Prior bounding boxes along image borders

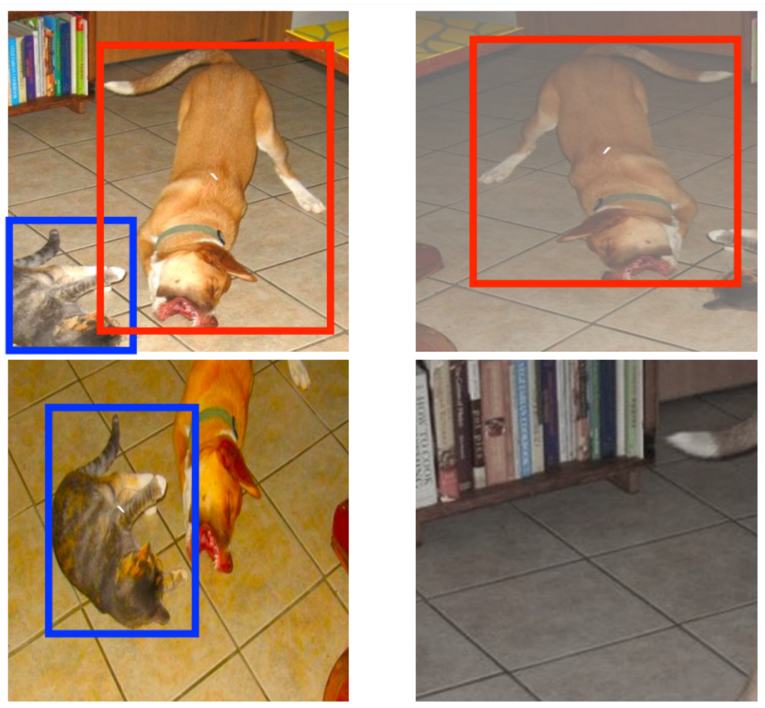

Figure 3.3: Random cropping, expansion and mirror transformation for data augmentation of images.

classification and regression headers, which compute the probability of a detected object belonging or not to a class and the position of the predicted bounding boxes. To avoid having multiple bounding boxes for the same object, the algorithm

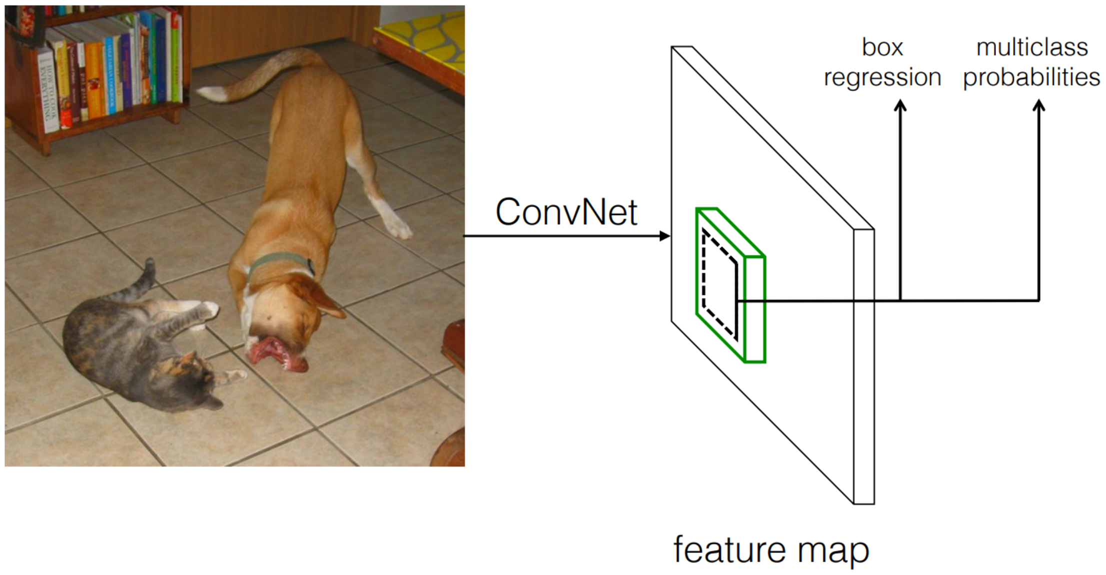

Figure 3.4: Regression and classification loss.

discards redundant boxes using the non-maximum suppression technique. The metrics used to evaluate the network are:

- Precision: $\frac{TP}{TP+FP}$ Metric useful for evaluating how the network performs with false positives, meaning objects erroneously detected.
- Recall: $\frac{TP}{TP+FN}$ Metric useful for assessing the model's ability to identify all objects of interest without overlooking any (avoiding false negatives).
- F1-score: $\frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision}+\text{Recall}}$ Trade-off between precision and recall.

Since the application requires recognizing images with small objects, the size of the prior boxes increases linearly along the extraction layers starting from 15 pixels. The layers from which the feature maps are extracted were chosen to obtain both low and high-level features. High-level features are extracted in the terminal part of the network, particularly in extra layers. In this case, the performance metrics of MobileNetV2 and VGG16 were analyzed and compared.

3.2 MobileNetV2 network

MobileNetV2 network is widely used in object detection contexts and embedded applications. It is composed by the repetition of inverted residual modules. These involve the use of separable convolutions, which are similar to standard convolutions but allow for a significant reduction in parameters number and, consequently, on-chip area [22]. A pre-trained MobileNetV2 network model is used in [23], enabling users to employ the pre-trained weights to initialize the backbone and classification and regression headers using Xavier initialization. Here are reported the network initial specifications:

```
image_size = 512
image_mean = np.array([127, 127, 127])  # RGB layout
image_std = 128.0
iou_threshold = 0.45
center_variance = 0.1
size_variance = 0.2
num_classes = 2
```

Training shown in Figure 3.5 was performed using a multistep scheduler, with a learning rate of 10^{-3} for the first 20 epochs, 10^{-4} for the next ten, and finally 10^{-5} for another ten epochs. The training was conducted with a batch size of 16, momentum of 0.9, and weight decay of $5 \cdot 10^{-4}$.

Results from the performance evaluation metrics are depicted in Figure 3.6, Figure

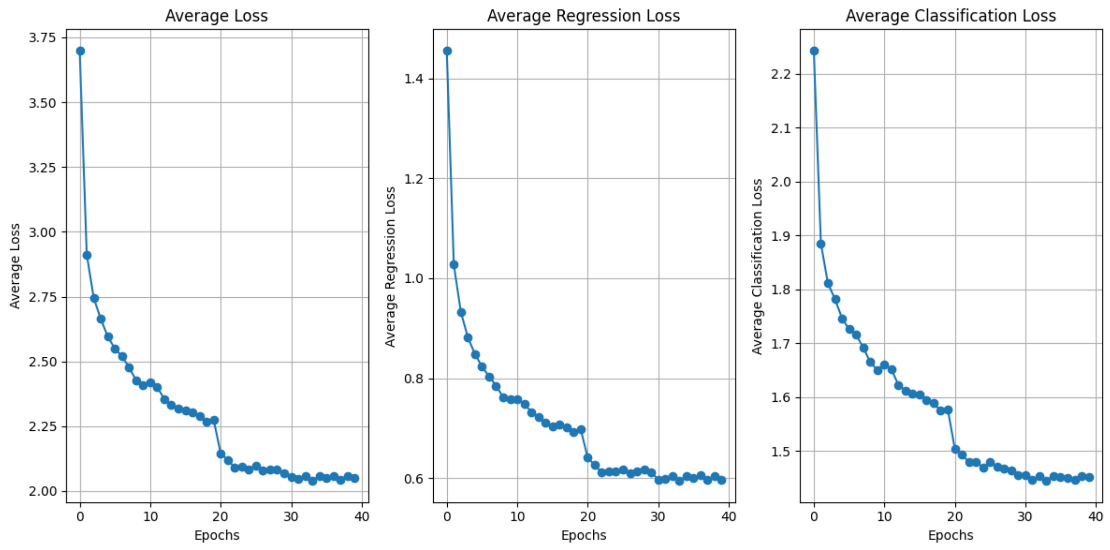

Figure 3.5: Average loss, regression loss and classification loss over epochs with MobileNetV2 backbone

3.7 and Figure 3.8. Precision and recall have values approximately equal to 76% and 68%, respectively, in the first test, 58% and 59% in the second test and 73% and 66% in the third test. The network thus exhibits excellent performance in recognizing this type of objects.

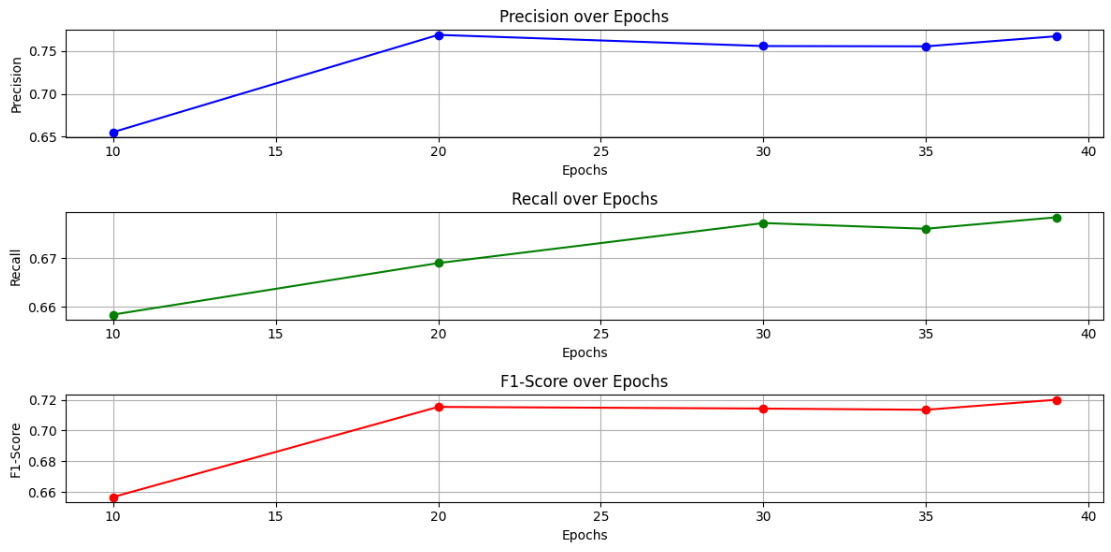

Figure 3.6: Precision, recall and F1-score over epochs using MobileNetV2 backbone: Test1

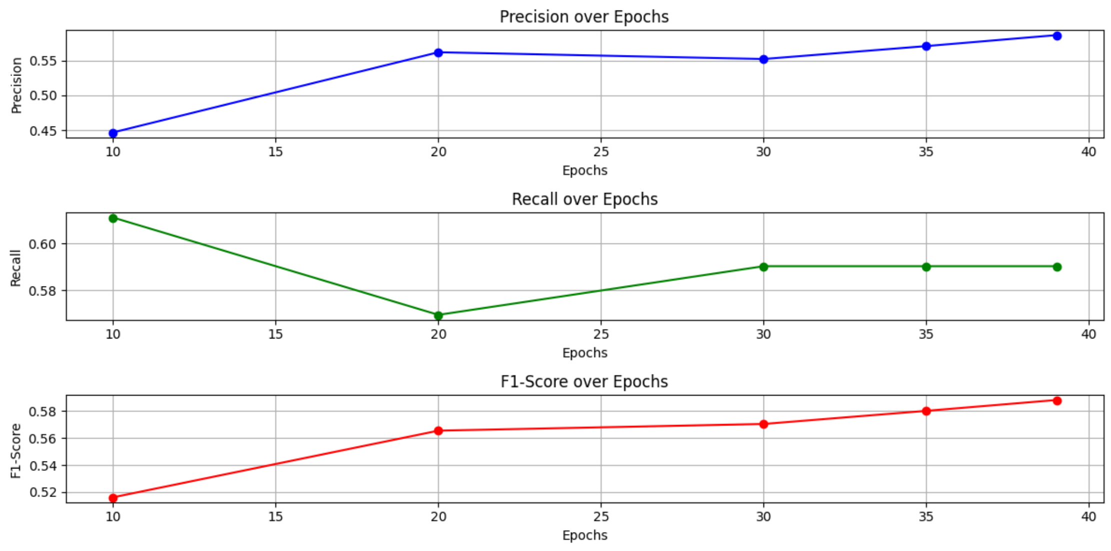

Figure 3.7: Precision, recall and F1-score over epochs using MobileNetV2 backbone: Test2

3.3 VGG16 network

VGG16 network is employed in lots of applications, particularly in tasks involving image classification and object recognition. As in MobileNetV2, extra layers

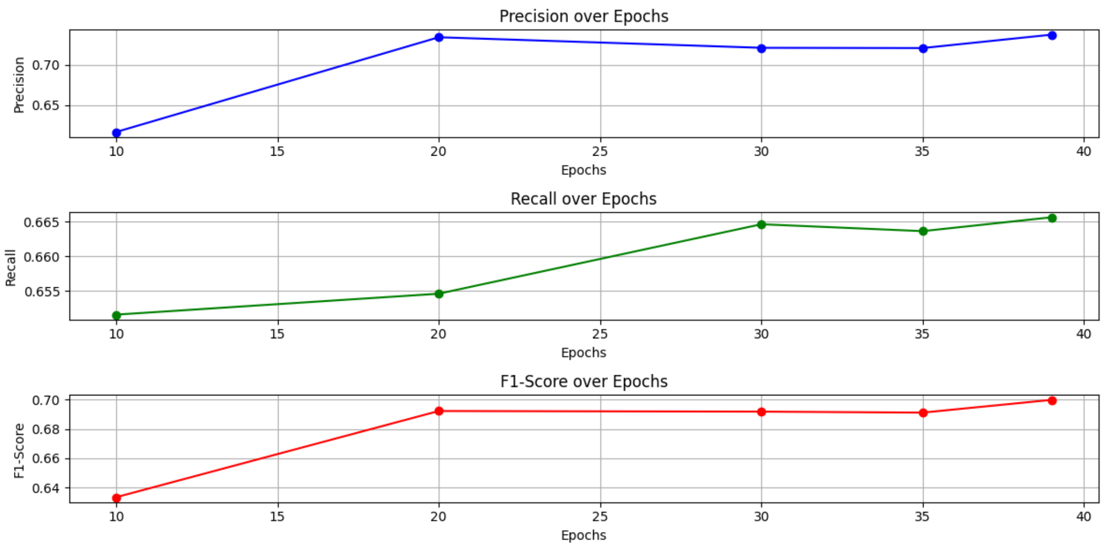

Figure 3.8: Precision, recall and F1-score over epochs using MobileNetV2 backbone: Test3

are introduced to the network to extract features at different levels, including convolutional layers, batch normalization layers, ReLU activations and max-pooling layers to reduce image dimensions. Below are reported the network specifications:

```
image_size = 512
image_mean = np.array([123, 117, 104])  # RGB layout
image_std = 1.0
iou_threshold = 0.45
center_variance = 0.1
size_variance = 0.2
num_classes = 2
```

During convolution steps inside the network it is essential to introduce batch normalization layers to avoid gradient vanishing problems and assure a better convergence of the loss function. No pre-trained network was used to initialize the backbone parameters. Training shown in Figure 3.9 was executed using a multistep scheduler, with a learning rate set at 10^{-3} for the initial 50 epochs, 10^{-4} for the subsequent ten, and finally, 10^{-5} for another ten epochs. The training configuration included a batch size of 8, momentum of 0.9 and weight decay of $5 \cdot 10^{-4}$. Given its increased depth compared to MobilenetV2, the VGG16 network entails more convolutional steps before extracting feature maps crucial for classifying and locating ships in images. For this reason, VGG16 network performance is better than the previous backbone. The precision and recall values, as depicted in

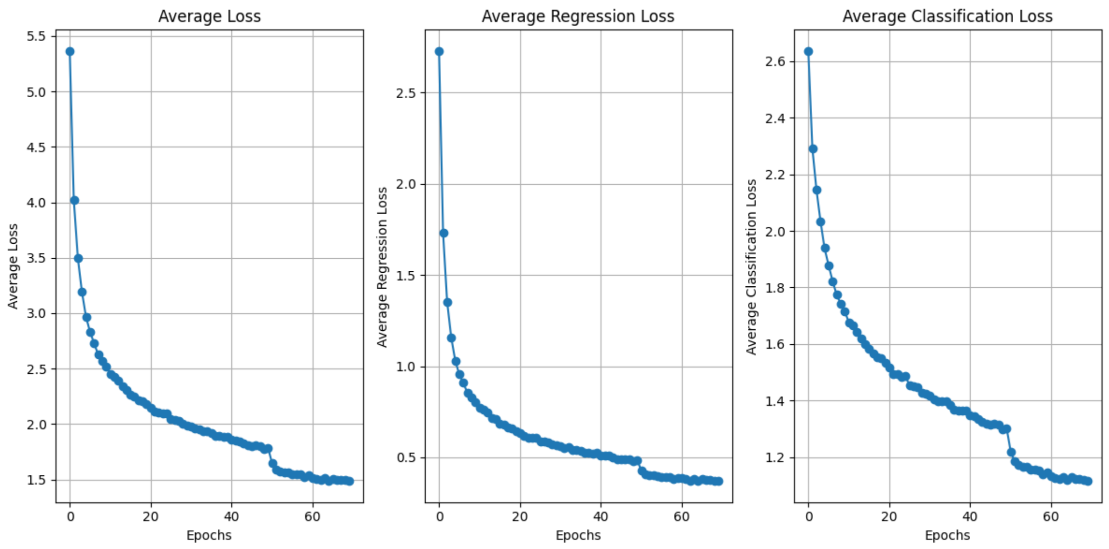

Figure 3.9: Average loss, regression loss and classification loss over epochs with VGG16 backbone

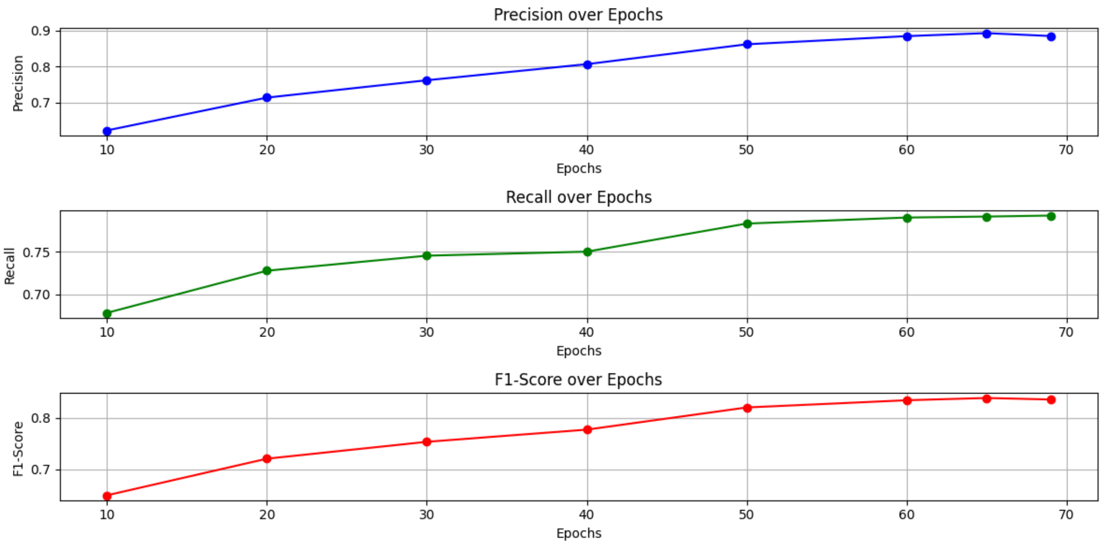

Figure 3.10: Precision, recall and F1-score over epochs using VGG16 backbone: Test1

Figure 3.10, Figure 3.11 and Figure 3.12, are approximately 88% and 79% for the first test, 76% and 59% for the second test and 87% and 76% for the third test, respectively.

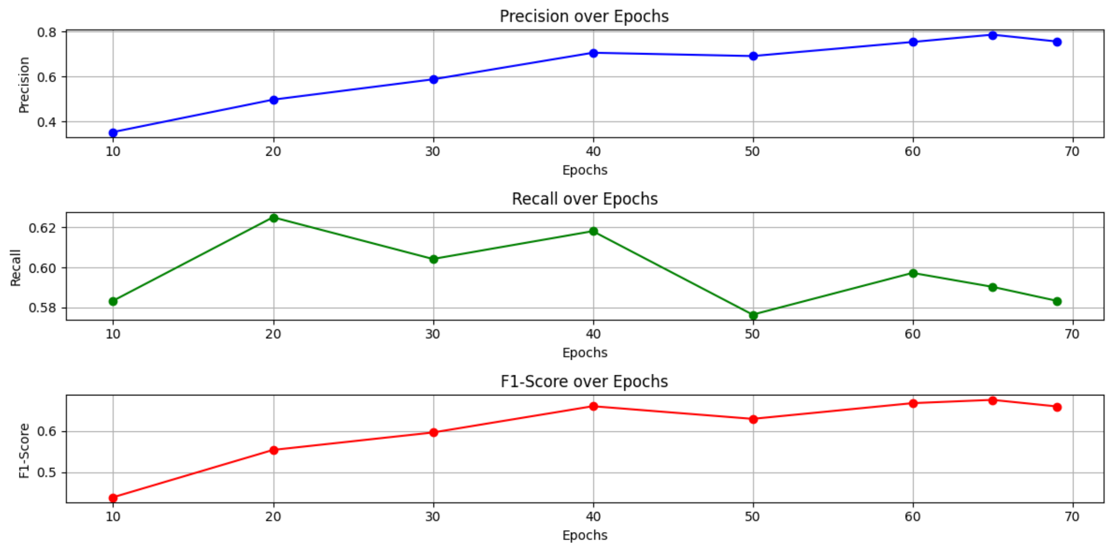

Figure 3.11: Precision, recall and F1-score over epochs using VGG16 backbone: Test2

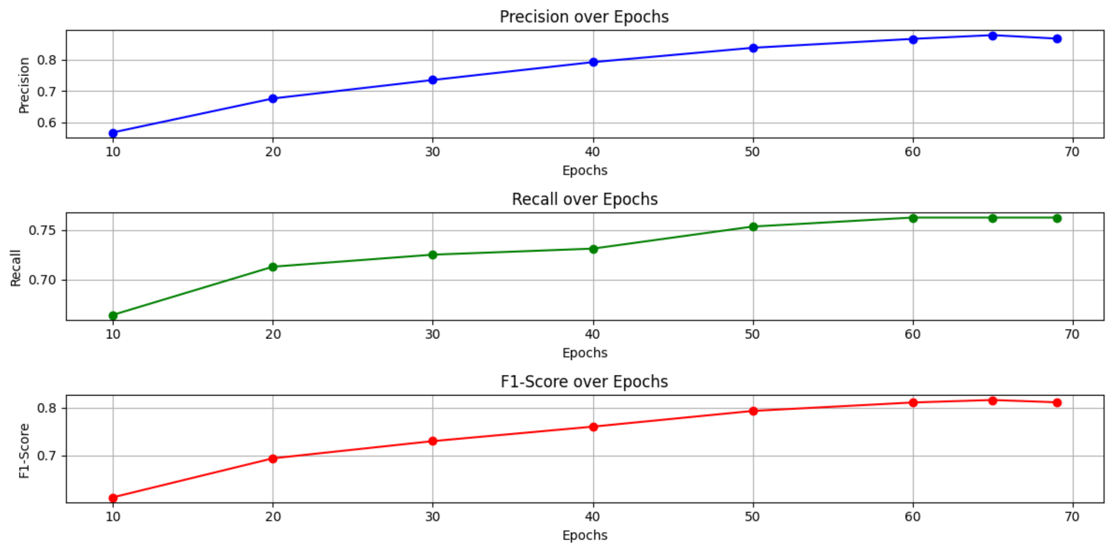

Figure 3.12: Precision, recall and F1-score over epochs using VGG16 backbone: Test3

3.4 Quantization techniques

To save memory as much as possible, it is crucial to reduce the number of bits used to represent parameters and activations. This process must be executed accurately to avoid significant degradation in network performance. Lighter models offer the following benefits:

- Smaller storage (Flash memory) size.
- Smaller download size, resulting in less time and download bandwidth.
- Less memory (RAM) usage, allowing more memory for other parts of an application, potentially improving overall performance.

Various quantization techniques can be employed to reduce the binary representation:

- PTQ (Post-training quantization): it reduces the memory size of weights and activations after training. As quantization is performed after training, application performance may experience a more or less drastic decline depending on the specific model used. It is divided into subgroups based on the desired datatype, assuming starting with FP32 precision:
  - FP16 PTQ: weights trained using FP32 are cast to FP16; activations computed using FP16; minimal accuracy loss; 2x smaller size.
  - Dynamic-Range PTQ: weights converted and stored as INT8 (requires computing $scale_{factor}$ and $zero_{point}$). Input activations converted on the fly from FP32 to INT8. Output activations stored as FP32. The model executes operations that mix integer and float computation when available; otherwise falls back to FP32 operation.
  - Full Integer PTQ: it requires calibration to convert everything to INT8 (both activations and weights). The model executes with integer operations only and conversion fails if the model has unsupported operations.
  - Full Integer PTQ with FP32 fallback: similar to Full Integer PTQ but allows falling back to FP32 operation when integer operations are not available.
- QAT (Quantization-aware training): layers used during training are replaced with quantized layers. Quantization error is considered during training, resulting in less accuracy reduction compared to PTQ. However, this type of quantization is challenging to implement in tools used for completing application synthesis.

The chosen quantization type for the case study is QAT. Quantization was performed using the Brevitas framework, which facilitates the straightforward replacement of standard layers with quantized layers. This framework supports various functionalities and produces quantized layers as output obtained from the convolution of quantized layers. A correct quantized tensor is obtained using the formulation:

$$quantized_{value} = \frac{value}{scale_{factor} + zero_{point}} \tag{3.2}$$

In this case, assuming $zero_{point} = 0$ since the distribution range of weights and activations is symmetric. The $scale_{factor}$ is a conversion factor that manages the transition between high-precision floating point representation during training and low-precision quantized representation during inference. It allows the model to perform operations with quantized values while maintaining relative precision. The value of $scale_{factor}$ is the same for the entire considered quantized tensor and is equal to:

$$scale_{factor} = \frac{1}{Max_{activation}} \tag{3.3}$$

For this reason, as activations have different values depending on the considered layer, each activation will have its $scale_{factor}$. Quantization was only applied to the layers belonging to the application's backbone, leaving the classification and regression headers unchanged.

3.5 Quantized MobileNetV2 network

The structure of MobileNetV2 network has been modified by replacing the convolution layers and ReLU6 activations with quantized convolution layers and quantized ReLU activations. The $scale_{factor}$ used is an 8 bit fixed point datatype for weights and input and output activations, while biases are quantized to 16 bits. This choice provides a good balance between accuracy and memory usage. The computation of the $scale_{factor}$ is automatically done with Brevitas, which allows the tensor to be returned with quantized formatting. The training shown in Figure 3.13 was performed using a multistep scheduler, with a learning rate of 10^{-3} for the first 20 epochs, 10^{-4} for the next ten, and finally 10^{-5} for another ten epochs. The training was conducted with a batch size of 16, momentum equal to 0.9, and weight decay of $5 \cdot 10^{-4}$. The metrics of the network remain more or less the same once it has been quantized. The precision and recall values, as reported in Figure 3.14, Figure 3.15 and Figure 3.16, are equal to 68% and 58% for the first test, 44% and 48% for the second test and 64% and 57% for the third test, respectively. The total number of parameters and the memory space occupied by the backbone are calculated using the formula:

$$\text{Memory parameter} = \frac{\#\text{PARAM(weights)} \times 1B}{2^{20}} + \frac{\#\text{PARAM(biases)} \times 2B}{2^{20}} \tag{3.4}$$

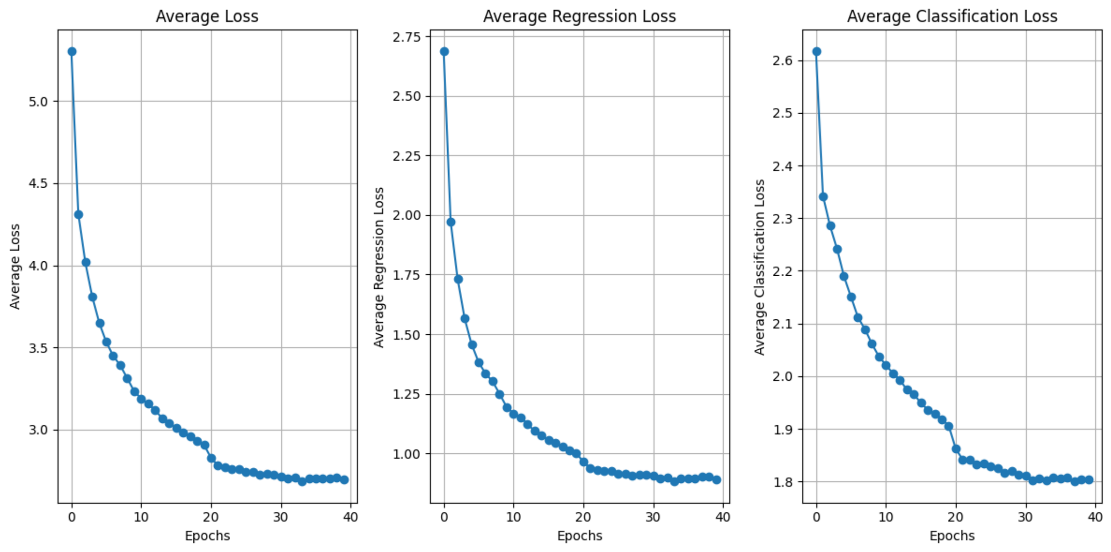

Figure 3.13: Average loss, regression loss and classification loss over epochs with quantized MobileNetV2 backbone

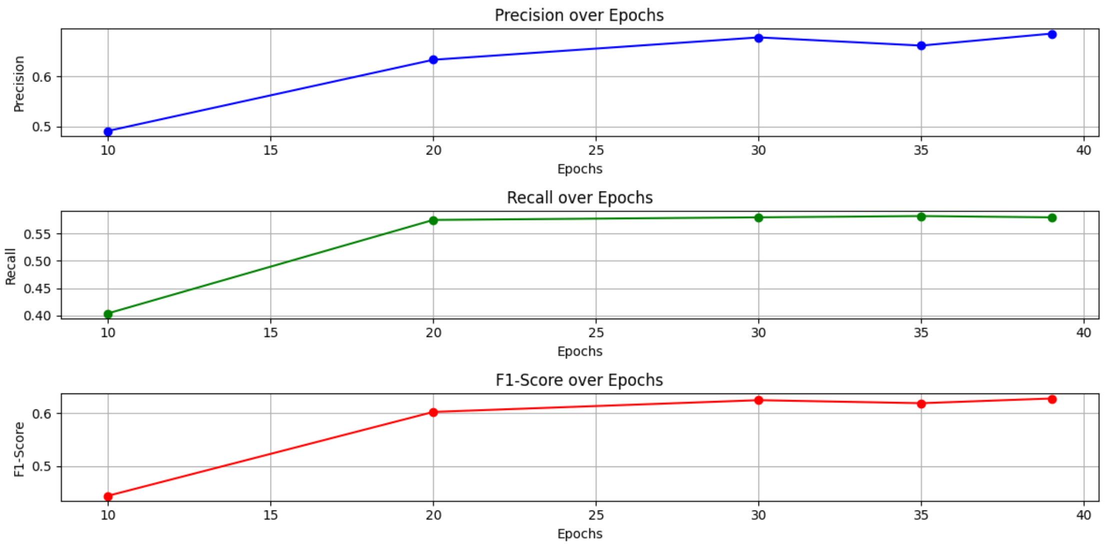

Figure 3.14: Precision, recall and F1-score over epochs after quantization using MobileNetV2 backbone: Test1

Depending on the type of platform used, parameters will be stored in FPGA's BRAMs or in off-chip memory. For this type of application, FPGAs have much larger on-chip memory than standard ones, so it is often unnecessary to resort to external memory. The total number of parameters used for the backbone is

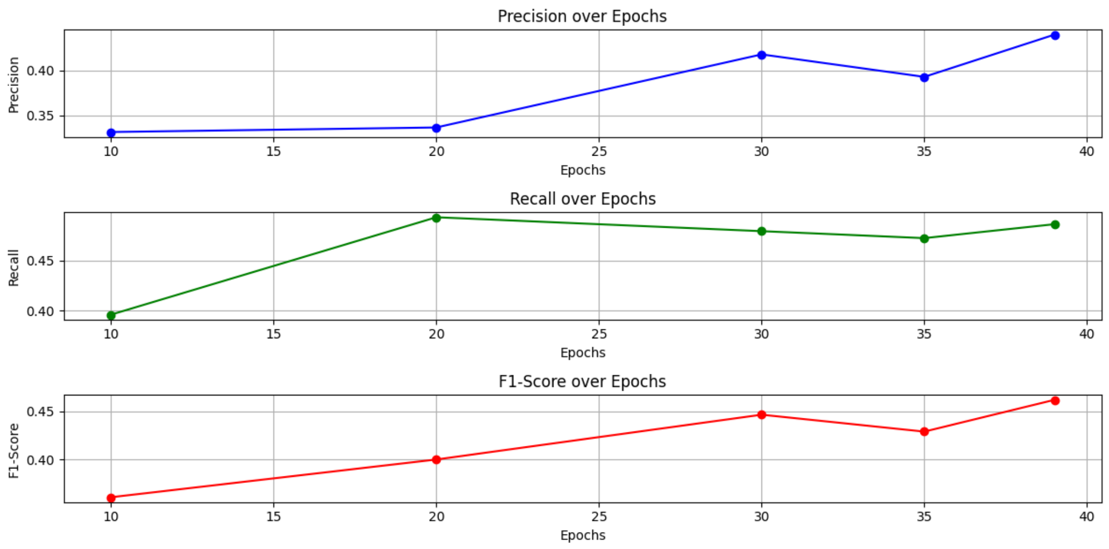

Figure 3.15: Precision, recall and F1-score over epochs after quantization using MobileNetV2 backbone: Test2

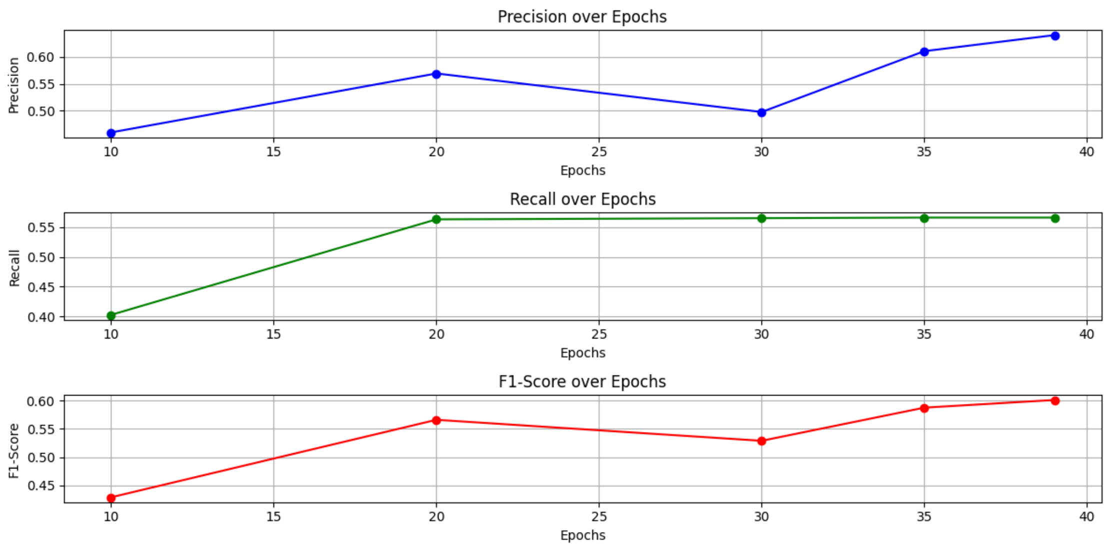

Figure 3.16: Precision, recall and F1-score over epochs after quantization using MobileNetV2 backbone: Test3

respectively 2,857,483 weights and 19,296 biases. The total memory occupied by the backbone, using the Equation 3.4, amounts to 2.76 MB. In terms of computational costs, the number of multiplication and accumulation operations required by the network to produce outputs can be calculated. Specifically, considering only convolution operations, assuming that batch normalization layers are merged with convolution layers to preserve resources in terms of latency and operations executed, the total count of computations in the case of the MobileNetV2 network amounts to 1,747,217,728 MAC operations.

3.6 Quantized VGG16 network

The network structure has been modified by replacing the convolution layers, ReLU activations and max-pooling layers with quantized convolution layers, quantized ReLU activations and quantized max-pooling layers. The data type chosen for both weights and biases matches that of the previous backbone. The calculation of the $scale_{factor}$ is automatically done with Brevitas, which allows the tensor to be returned with quantized formatting. As depicted in Figure 3.17, the training was performed using a multistep scheduler, with a learning rate of 10^{-3} for the first 50 epochs, 10^{-4} for the next ten, and finally 10^{-5} for another ten epochs. The training was conducted with a batch size of 8, momentum equal to 0.9, and weight decay equal to $5 \cdot 10^{-4}$. After the quantization, the performance metrics remain

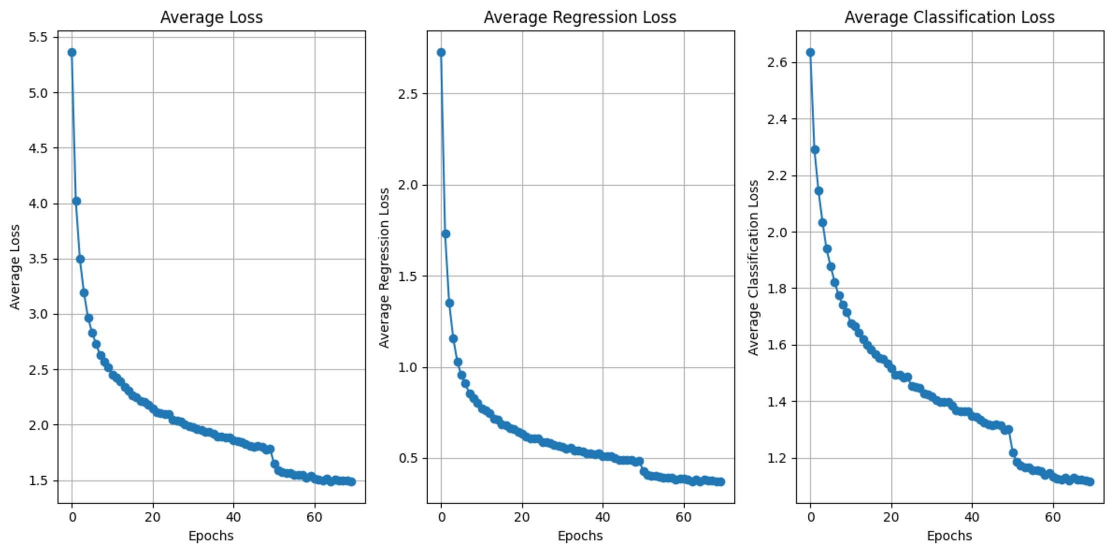

Figure 3.17: Average loss, regression loss and classification loss over epochs with quantized VGG16 backbone

more or less the same. As appears in Figure 3.18, Figure 3.19 and Figure 3.20, precision and recall have values approximately equal to 86% and 75%, respectively, in the first test, 73% and 59% in the second test and 84% and 73% in the third test. For the total computation of network parameters, the VGG16 backbone comprises 23,496,512 weights and 12,800 biases. This results in a memory footprint

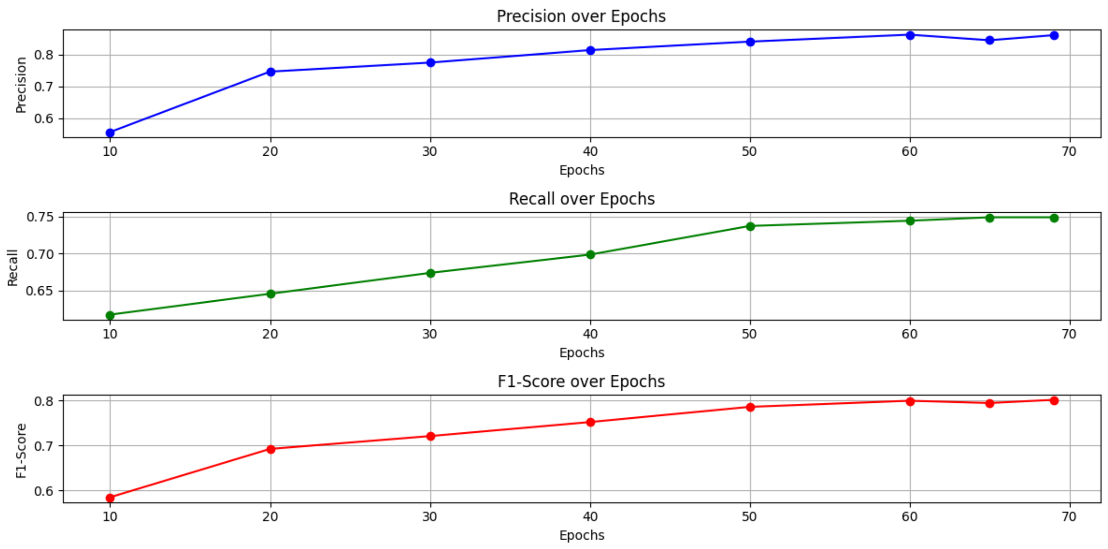

Figure 3.18: Precision, recall and F1-score over epochs after quantization using VGG16 backbone: Test1

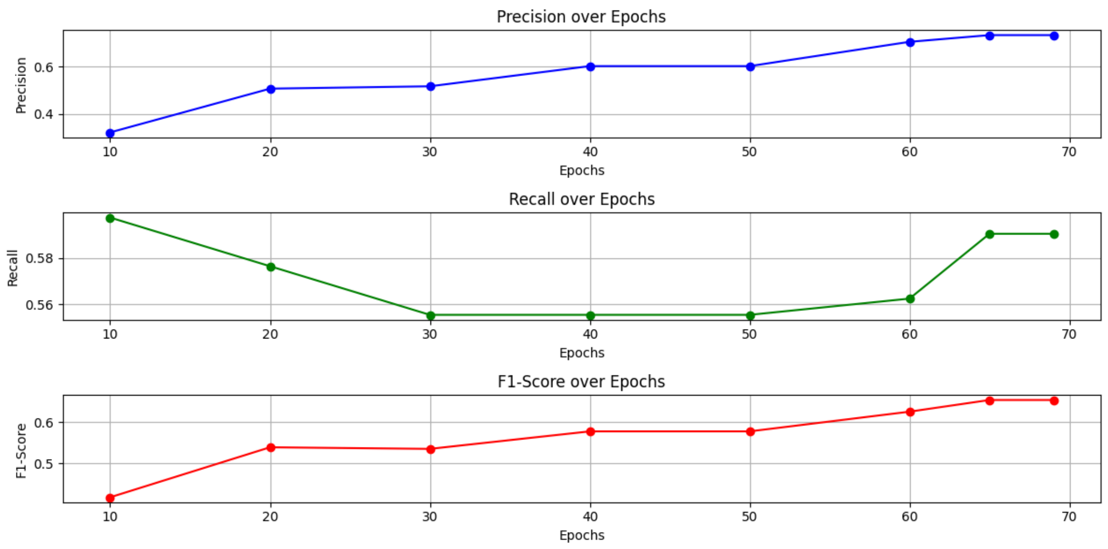

Figure 3.19: Precision, recall and F1-score over epochs after quantization using VGG16 backbone: Test2

of 22.4 MB. Dataflow accelerators are available to handle networks of this size for such applications. If the platform cannot support the network, it would necessitate the use of off-chip memory for storing both activations and parameters. The total number of MAC (Multiply-Accumulate) operations performed in this case is

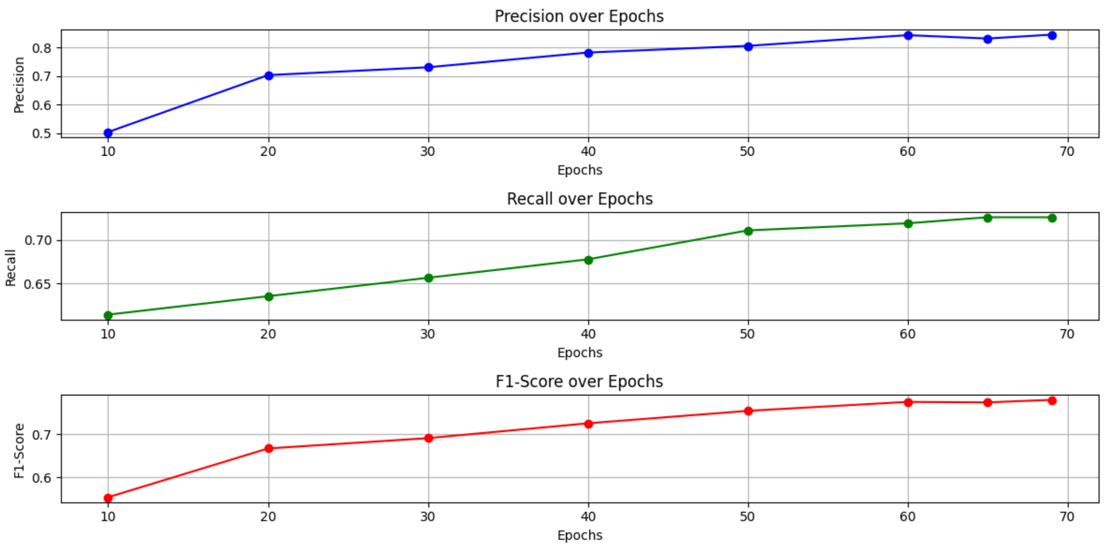

Figure 3.20: Precision, recall and F1-score over epochs after quantization using VGG16 backbone: Test3

87,541,314,560.

3.7 Output predictions

In this section, the results after an example of inference are displayed using images. The predicted bounding boxes are marked in red, while the ground truth bounding boxes are marked in blue. Only output examples related to the inference of the MobileNetV2 network are depicted in the following images. In Figure 3.21 the presence of wrong ground truth bounding boxes is noticeable, impacting the network's performance. This scenario is challenging for prediction, marked by the occurrence of a false positive. The existence of inaccurate ground truth boxes within the white circle is evident also in Figure 3.22. In this case, the image proves to be a difficult instance of recognition, as there is a scenario with objects similar to ships. The previously highlighted error affects prediction in Figure 3.23. The application cannot recognize ships whose dimensions are below approximately 10 pixels (Figure 3.24). Large amounts of ships are correctly predicted, as shown in Figure 3.25 and Figure 3.26. There are some false positive predictions (Figure 3.27, Figure 3.28) in environments tough to discern. A similar discussion can be made for the VGG16 network, which proves to be more accurate in the localization and recognition of ships.

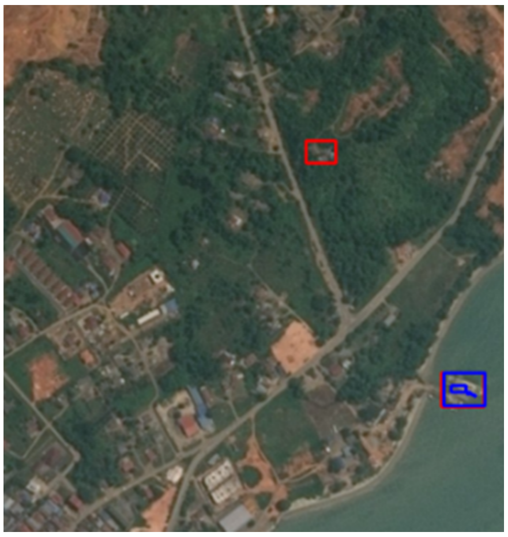

Figure 3.21: Prediction 1

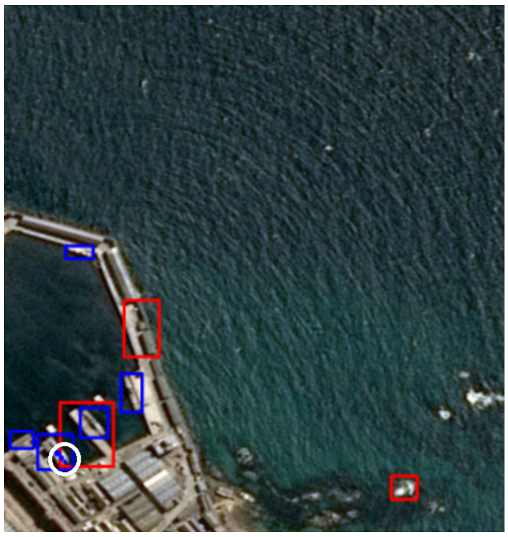

Figure 3.22: Prediction 2

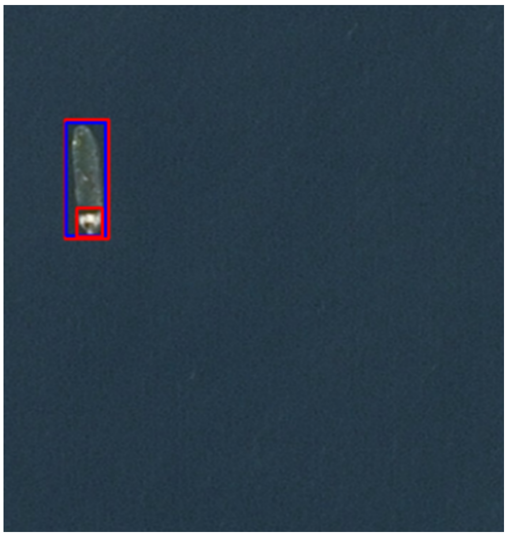

Figure 3.23: Prediction 3

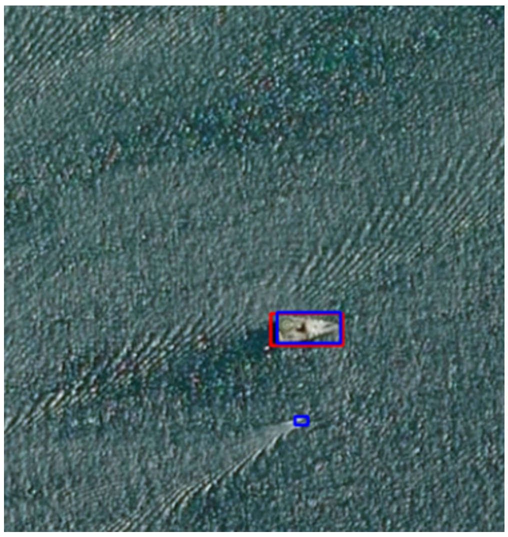

Figure 3.24: Prediction 4

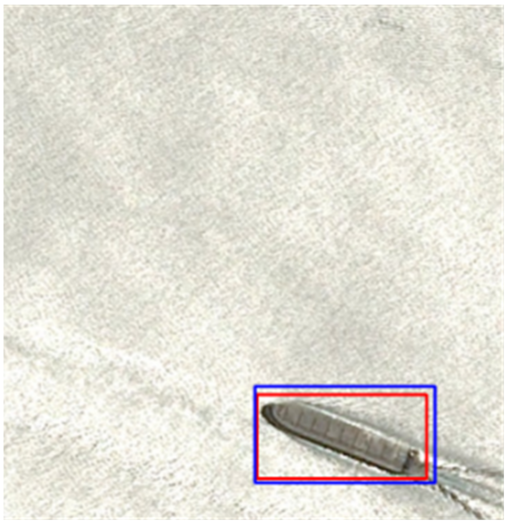

Figure 3.25: Prediction 5

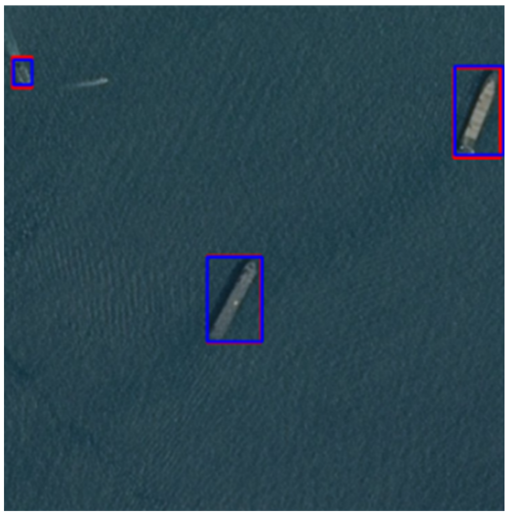

Figure 3.26: Prediction 6

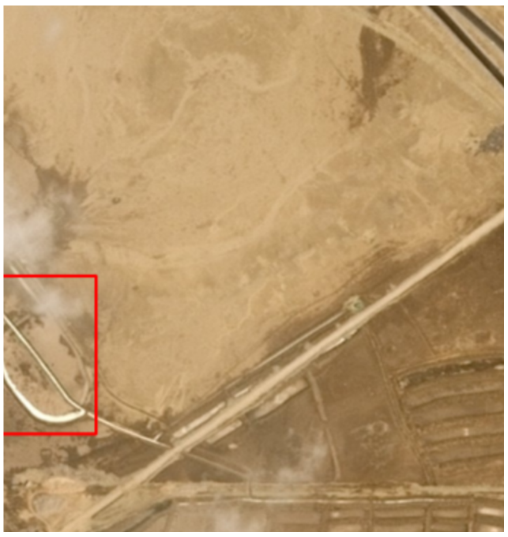

Figure 3.27: Prediction 7

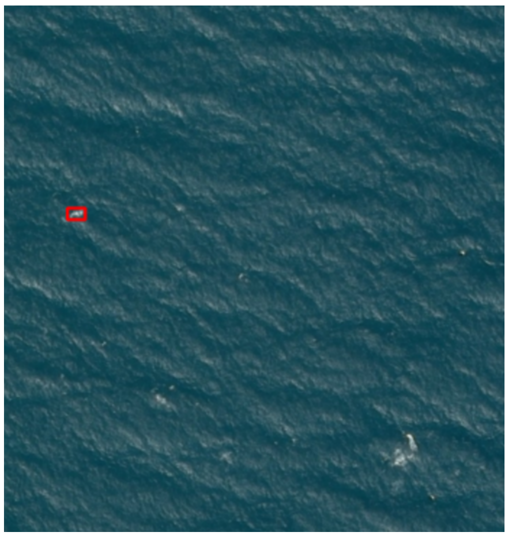

Figure 3.28: Prediction 8

3.8 Network without training augmentation

Training augmentation is essential to ensure optimal performance for the network under challenging conditions. In an application utilizing an onboard camera for image detection, the ability to recognize objects in low-light and low-sharpness conditions is indispensable. To assess the impact of training augmentation on the backbone, the network was retrained, and its performance was re-evaluated. The results on the test dataset are better without training augmentation, but in this type of application, it is still necessary to consider it to ensure that objects that are too small or not well illuminated are visible (such images are not included in the test dataset). Considering the performance metrics of precision and recall, we obtain for the first test values of 81% and 62%, for the second test 50% and 41%, and for the third test 76% and 59% for the MobileNetV2 network. Regarding the VGG16 network, the performance values are for the first test 88% and 79%, for the second test 75% and 68% and for the third test 87% and 76%.

Chapter 4

Error injection technique

As previously mentioned, the influence of radiation-induced bit-flip errors within FPGA platforms on intermediate activations and subsequent catastrophic performance degradation is the focus of this case study. Error injection has been specifically applied to intermediate activations, assuming that weights are safeguarded by error correction code (ECC) mechanisms within the FPGA memory allocations. Considering the structure of a traditional DNN execution pipeline, when a device is exposed to a radioactive particle resulting in a bit-flip error within the line buffers responsible for computing intermediate outputs, the intensity of the particles or their impact across multiple memory regions may lead to error accumulation within an intermediate activation. This is because, during each clock cycle, the pixels within the intermediate activations stored in the input window registers are shifted by one position to execute a new convolution operation. Consequently, an error occurring in a line buffer, if sustained over numerous convolution operations, can affect multiple pixels within the activation. Therefore, it is crucial to assess not only the impact of a single bit-flip error (often negligible) but also the potential accumulation of bit-flip errors across multiple pixels. To execute error injection, PyTorch Hook functions were employed. These functions offer convenient access to internal neural network modules during both forward and backward propagation phases, commonly utilized for debugging, gradient analysis or dynamic modification of network behavior. Following a methodology similar to that detailed in [24], Hook functions were used to manipulate intermediate activations and introduce bit-flip errors. Bit-flip errors were injected into each layer of each backbone, with a variable percentage ranging from 20% to 80% for each bit. The choice of the pixel to be affected occurs randomly: the pixel is incremented by a value equal to that of the bit affected by radiation. Referring to Equation 3.2 and 3.3, it is possible to modify the value contained within a pixel of an intermediate activation. This simulates the presence of a bit-flip, transitioning from the initial value of 0 to 1. The opposite case has not been addressed, as both networks use the ReLU activation function, which eliminates the presence of errors in case of negative values. Upon obtaining the complete model, the ensuing output metrics were thoroughly evaluated to gauge the impact of bit-flip errors on performance. The metrics were evaluated exclusively on the test dataset containing at least one ship in each image (Test1). The degradation of performance in the output will depend on the depth of the considered layer, the number of activation pixels it contains and whether the layer is utilized to extract and generate the final outputs. The deeper the layer, the more likely the error will propagate through the convolutions and generate a negative output. It is feasible to illustrate, employing tables, the response of the network under conditions of accumulating these kinds of errors at the intermediate activation level. In this scenario, the metric to consider is recall, as injecting such errors alters the network's behavior, resulting in a failure to recognize target objects and leading to an increase in false negatives. When too many errors are injected into a layer, the network, being unaccustomed to processing such intermediate activations, will generate significantly fewer predictions. Consequently, the number of false positives will be notably reduced, resulting in an improvement in precision values in certain cases.

4.1 MobileNetV2 error injection technique

It is feasible to assess how the performance of the network degrades or not for each layer following the error injection. In MobileNetV2 the extraction layers used are the second output within the inverted residual module of the sixth, thirteenth, eighteenth layers and the additional extra layers added to the main architecture. In this case, the Hook function was introduced at the end of the output of the inverted residual module, before the final batch normalization layer. It is possible to evaluate, bit by bit, how the network's performance degrades on a single layer in this regard.

LAYER 0 Layer 0 is the first layer encountered within the MobileNetV2 and consists of a single convolutional layer. Starting from a size of 512x512, the image is reduced to half its original dimensions with an increase in the number of output channels. It is the layer with the highest number of pixels within the network, amounting to 2,097,152 units. Consequently, the number of pixels where the error is injected will also be higher. For this reason, the probability of the network's performance being degraded increases, as can be observed from Table 4.1.

LAYER 1-2-3-4-5 Layers 1, 2, 3, 4 and 5 are inverted residual layers. They feature a significantly higher number of pixels within the activation compared to the final layers. As mentioned earlier, extracting features from these layers allows

Bit	Metrics	20% bit-flip injection	40% bit-flip injection	60% bit-flip injection	80% bit-flip injection
0	Precision	0.68854	0.68333	0.68917	0.68251
	Recall	0.58068	0.57950	0.57714	0.57479
	F1-score	0.63003	0.62715	0.62820	0.62404
1	Precision	0.68892	0.69546	0.69857	0.68882
	Recall	0.57126	0.57832	0.57597	0.57361
	F1-score	0.62459	0.63151	0.63137	0.62596
2	Precision	0.69640	0.70144	0.70246	0.70639
	Recall	0.57008	0.57008	0.57008	0.57243
	F1-score	0.62694	0.62897	0.62938	0.63240
3	Precision	0.71176	0.71879	0.71343	0.71299
	Recall	0.57008	0.56301	0.56301	0.55594
	F1-score	0.63309	0.63143	0.62936	0.62475
4	Precision	0.74877	0.75043	0.76672	0.76490
	Recall	0.54063	0.50647	0.52650	0.52885
	F1-score	0.62790	0.60478	0.62430	0.62534
5	Precision	0.77299	0.77973	0.79954	0.77446
	Recall	0.46525	0.41696	0.41342	0.42873
	F1-score	0.58088	0.54336	0.54503	0.55193
6	Precision	0.34393	0.79333	0.78467	0.80952
	Recall	0.34393	0.28032	0.25323	0.26030
	F1-score	0.48105	0.41427	0.38290	0.39393
7	Precision	0.81909	0.83193	0.83544	0.85507
	Recall	0.19199	0.11660	0.07773	0.06949
	F1-score	0.31106	0.20454	0.14224	0.12854

Table 4.1: Performance metrics for different percentages of bit-flip injection within layer 0 for MobileNetV2 backbone

us to obtain low-level features. Based on the number of pixels within the layer, we will have more or less catastrophic consequences on the output of the network. In this case, layer 2, as shown in Table 4.2, exhibits the least internal resilience among all, resulting in a catastrophic output starting from a modification of bit 5.

LAYER 6 Layer 6 is one of the input feature extraction layers used to determine the size and location of the predicted bounding boxes. Starting from bit 6, the network's inference is altered to the extent that it drastically reduces the network's performance, as shown in Table 4.3.

Bit	Metrics	20% bit-flip injection	40% bit-flip injection	60% bit-flip injection	80% bit-flip injection
0	Precision	0.68802	0.69219	0.69503	0.68144
	Recall	0.58186	0.57479	0.57714	0.57950
	F1-score	0.63050	0.62805	0.63063	0.62635
1	Precision	0.69943	0.69774	0.68627	0.68679
	Recall	0.58186	0.57714	0.57597	0.57832
	F1-score	0.63455	0.62699	0.62652	0.63110
2	Precision	0.70724	0.70930	0.72106	0.70562
	Recall	0.57479	0.57479	0.57243	0.57597
	F1-score	0.63417	0.63500	0.63821	0.63424
3	Precision	0.72291	0.73990	0.75040	0.74068
	Recall	0.55005	0.53945	0.54534	0.53828
	F1-score	0.62474	0.62397	0.63165	0.62346
4	Precision	0.77019	0.78336	0.76878	0.76806
	Recall	0.50530	0.47703	0.46996	0.47585
	F1-score	0.61024	0.59297	0.58333	0.58763
5	Precision	0.78830	0.68600	0.78902	0.82131
	Recall	0.33333	0.23674	0.22025	0.30859
	F1-score	0.46854	0.35201	0.34438	0.44863
6	Precision	0.78571	0.00117	0.00117	0.00235
	Recall	0.03886	0.00117	0.00117	0.00235
	F1-score	0.07407	0.00235	0.00235	0.00470
7	Precision	0.0	0.5	1.0	1.0
	Recall	0.0	0.00117	0.00117	0.00235
	F1-score	0.0	0.00235	0.00235	0.00470

Table 4.2: Performance metrics for different percentages of bit-flip injection within layer 2 for MobileNetV2 backbone

LAYER 7-8-9-10-11-12 These layers facilitate the processing of the image to extract middle-level features. The image size is further reduced while increasing the number of channels. The earlier portion of the backbone remains unaffected by errors, ensuring that some ships will still be recognized. Layer 7 exhibits lower resilience within the series, starting to degrade the network's performance from bit 5 (Table 4.4).

LAYER 13 Layer 13, similar to layer 6, is employed for feature extraction within the network and the computation of its outputs. More catastrophic consequences arise from bit-flips affecting the most significant bit, shown in Table 4.5.

Bit	Metrics	20% bit-flip injection	40% bit-flip injection	60% bit-flip injection	80% bit-flip injection
0	Precision	0.69154	0.69014	0.68575	0.69252
	Recall	0.57832	0.57714	0.57832	0.57832
	F1-score	0.62989	0.62860	0.62747	0.63029
1	Precision	0.68715	0.69393	0.68917	0.69757
	Recall	0.57950	0.57950	0.57714	0.57597
	F1-score	0.62875	0.63157	0.62820	0.63096
2	Precision	0.69154	0.69559	0.69971	0.69884
	Recall	0.57832	0.57597	0.57361	0.57126
	F1-score	0.62989	0.63015	0.63042	0.62864
3	Precision	0.69491	0.70348	0.71279	0.73076
	Recall	0.57950	0.57008	0.56419	0.55948
	F1-score	0.63198	0.62979	0.62984	0.63375
4	Precision	0.70114	0.73219	0.74367	0.75953
	Recall	0.57479	0.55712	0.55359	0.53945
	F1-score	0.63171	0.63277	0.63470	0.63085
5	Precision	0.72515	0.75945	0.78269	0.82105
	Recall	0.55005	0.52061	0.47938	0.45936
	F1-score	0.62558	0.61774	0.59459	0.58912
6	Precision	0.70567	0.73096	0.69122	0.54166
	Recall	0.46878	0.33922	0.23203	0.13780
	F1-score	0.56334	0.46339	0.34744	0.21971
7	Precision	0.23183	0.03170	0.00757	0.00124
	Recall	0.15783	0.01531	0.00235	0.00117
	F1-score	0.18780	0.02065	0.00359	0.00120

Table 4.3: Performance metrics for different percentages of bit-flip injection within layer 6 for MobileNetV2 backbone

LAYER 14-15-16-17 These types of layers are characterized by a high number of output channels. The information within the convolutions becomes increasingly high-level and is useful for recognizing the context in which the image is located. The network does not undergo catastrophic deteriorations when affected by bit-flip errors, as shown in Table 4.6 for layer 15.

LAYER 18 This is the last layer extracted from the backbone. It is characterized by a very high number of output channels. It is important to recognize high-level features. As for layers 14-15-16-17, there are no catastrophic consequences on the output, regardless of the type of bit affected. Results are represented in Table 4.7

Bit	Metrics	20% bit-flip injection	40% bit-flip injection	60% bit-flip injection	80% bit-flip injection
0	Precision	0.68845	0.69165	0.68767	0.69057
	Recall	0.58303	0.57597	0.57832	0.57832
	F1-score	0.63137	0.62853	0.62827	0.62948
1	Precision	0.69067	0.69295	0.69067	0.69503
	Recall	0.57597	0.57950	0.57597	0.57714
	F1-score	0.62813	0.63117	0.62813	0.63063
2	Precision	0.68980	0.69800	0.69797	0.70544
	Recall	0.57361	0.57714	0.56890	0.56419
	F1-score	0.62636	0.63185	0.62686	0.62696
3	Precision	0.69186	0.70764	0.72100	0.72785
	Recall	0.57126	0.55594	0.54181	0.53239
	F1-score	0.62580	0.62269	0.61869	0.61496
4	Precision	0.72429	0.71944	0.67689	0.63209
	Recall	0.54770	0.49234	0.38987	0.30153
	F1-score	0.62374	0.58461	0.49476	0.40829
5	Precision	0.68686	0.49333	0.46666	0.44791
	Recall	0.40047	0.17432	0.15665	0.15194
	F1-score	0.50595	0.25761	0.23456	0.22691
6	Precision	0.46428	0.46043	0.46043	0.46043
	Recall	0.15312	0.15076	0.15076	0.15076
	F1-score	0.23029	0.22715	0.22715	0.22715
7	Precision	0.46043	0.44755	0.45390	0.45714
	Recall	0.15076	0.15076	0.15076	0.15076
	F1-score	0.22715	0.22555	0.22634	0.22674

Table 4.4: Performance metrics for different percentages of bit-flip injection within layer 7 for MobileNetV2 backbone

EXTRA LAYERS Extra layers do not significantly affect the network's output due to bit-flip errors, given the chosen prior boxes scale and dataset. Bit-flip injection of extra layer 1 is depicted in Table 4.8.

4.2 VGG16 error injection technique

The VGG16 consists of 16 layers of convolutions and pooling, followed by three fully connected layers. Convolutions use 3x3 kernels with a stride of 1, while pooling employs 2x2 windows with a stride of 2. The initial convolutional layers are simple, but as the network progresses, convolutions become deeper and more

Bit	Metrics	20% bit-flip injection	40% bit-flip injection	60% bit-flip injection	80% bit-flip injection
0	Precision	0.68472	0.68333	0.68428	0.68611
	Recall	0.58068	0.57950	0.57950	0.58186
	F1-score	0.62842	0.62715	0.62755	0.62970
1	Precision	0.68421	0.68282	0.68333	0.68377
	Recall	0.58186	0.58068	0.57950	0.58068
	F1-score	0.62889	0.62762	0.62715	0.62802
2	Precision	0.68428	0.68282	0.68802	0.68384
	Recall	0.57950	0.58068	0.58186	0.57832
	F1-score	0.62755	0.62762	0.63050	0.62667
3	Precision	0.68333	0.68741	0.67812	0.67349
	Recall	0.58539	0.58539	0.58068	0.58068
	F1-score	0.62715	0.63231	0.62563	0.62365
4	Precision	0.68275	0.66088	0.64690	0.64993
	Recall	0.58303	0.58303	0.57832	0.57950
	F1-score	0.62897	0.61952	0.61069	0.61270
5	Precision	0.61209	0.51115	0.54566	0.61679
	Recall	0.57243	0.56654	0.56301	0.55359
	F1-score	0.59160	0.53743	0.55420	0.58348
6	Precision	0.27964	0.30123	0.40546	0.54765
	Recall	0.55005	0.51590	0.45465	0.39929
	F1-score	0.37078	0.38037	0.42865	0.46185
7	Precision	0.24485	0.28031	0.38421	0.41065
	Recall	0.40636	0.20965	0.17196	0.15429
	F1-score	0.30558	0.23989	0.23759	0.22431

Table 4.5: Performance metrics for different percentages of bit-flip injection within layer 13 for MobileNetV2 backbone

complex. Within the network, feature can be extracted at low, medium, or high levels depending on the dataset available. In the specific case of the SSD algorithm and the considered network, extracting information from deeper layers involves using more parameters for the classification and regression headers necessary to compute the network's outputs. This is because the final classification and regression headers are responsible for placing prior boxes within the extraction feature maps. Extracting features from very low-level layers would involve considering feature maps with a high number of pixels and thus a large number of prior boxes to consider. This would result in a significant overhead during training and inference. However, considering lower-level feature maps still allows us to achieve a good level

Bit	Metrics	20% bit-flip injection	40% bit-flip injection	60% bit-flip injection	80% bit-flip injection
0	Precision	0.68523	0.68523	0.68523	0.68619
	Recall	0.57950	0.57950	0.57950	0.57950
	F1-score	0.62795	0.62795	0.62795	0.62835
1	Precision	0.68238	0.68619	0.68523	0.68333
	Recall	0.57950	0.57950	0.57950	0.57950
	F1-score	0.62675	0.62835	0.62795	0.62715
2	Precision	0.68472	0.68282	0.68333	0.68428
	Recall	0.58068	0.58068	0.57950	0.57950
	F1-score	0.62842	0.62762	0.62715	0.62755
3	Precision	0.68377	0.68758	0.58068	0.67812
	Recall	0.58068	0.58068	0.58068	0.58068
	F1-score	0.62802	0.62962	0.62642	0.62563
4	Precision	0.68619	0.67582	0.66847	0.65211
	Recall	0.57950	0.57950	0.58186	0.58068
	F1-score	0.62835	0.62396	0.62216	0.61433
5	Precision	0.67582	0.63849	0.56597	0.46022
	Recall	0.57950	0.57832	0.57597	0.57243
	F1-score	0.62396	0.60692	0.57092	0.51023
6	Precision	0.60902	0.61558	0.63766	0.63418
	Recall	0.57243	0.54888	0.52650	0.49823
	F1-score	0.59016	0.58032	0.57677	0.55804
7	Precision	0.67069	0.66720	0.66720	0.66720
	Recall	0.52296	0.48881	0.48881	0.48881
	F1-score	0.58769	0.56424	0.56424	0.56424

Table 4.6: Performance metrics for different percentages of bit-flip injection within layer 15 for MobileNetV2 backbone

of performance while saving computational resources as much as possible. Below is a comprehensive analysis of how injecting errors into a layer generates catastrophic behavior in the output.

LAYER 0 The first layer of the network results from a convolution operation followed by the application of a ReLU activation function. The initial image size of 512x512 is reduced to its half. Additionally, it initiates an increase in the number of channels, making the network denser gradually. An error at this level can potentially lead to complete degradation of the network, especially in the case of a bit-flip in the most significant bit (MSB). Results are illustrated in Table 4.9.

Bit	Metrics	20% bit-flip injection	40% bit-flip injection	60% bit-flip injection	80% bit-flip injection
0	Precision	0.68333	0.68282	0.68282	0.68093
	Recall	0.57950	0.58068	0.58068	0.58068
	F1-score	0.62715	0.62762	0.62762	0.62682
1	Precision	0.68282	0.68093	0.67906	0.67534
	Recall	0.58068	0.58068	0.58068	0.58068
	F1-score	0.62762	0.62682	0.62603	0.62444
2	Precision	0.68093	0.58068	0.67578	0.65866
	Recall	0.58068	0.58068	0.58186	0.58186
	F1-score	0.62682	0.62642	0.62531	0.61788
3	Precision	0.67626	0.66756	0.58186	0.67582
	Recall	0.58068	0.58186	0.58186	0.57950
	F1-score	0.62484	0.62177	0.58186	0.62396
4	Precision	0.66397	0.49007	0.18326	0.07967
	Recall	0.58186	0.58186	0.58303	0.58303
	F1-score	0.62021	0.53204	0.27887	0.14018
5	Precision	0.58760	0.47439	0.45211	0.55455
	Recall	0.58068	0.57832	0.57832	0.58068
	F1-score	0.58412	0.52123	0.50749	0.56731
6	Precision	0.59754	0.63219	0.67760	0.69354
	Recall	0.57361	0.56890	0.55948	0.55712
	F1-score	0.58533	0.59888	0.61290	0.61789
7	Precision	0.64032	0.67633	0.66930	0.65560
	Recall	0.55359	0.52179	0.49823	0.48881
	F1-score	0.59380	0.58909	0.57123	0.56005

Table 4.7: Performance metrics for different percentages of bit-flip injection within layer 18 for MobileNetV2 backbone

LAYER 1-2-3-4-5 These layers contain low-level information internally. The number of pixels contained within these layers is very high, so injecting a high percentage of bit-flip errors can greatly reduce the network's performance, especially for the first two most significant bits. The layer that experiences the most significant performance drop in output is indeed the fourth layer, which has the highest number of pixels, amounting to 8,388,608. Information about error injection is illustrated in Table 4.10.

LAYER 6-7-8-9-10-11 These layers contain medium-low-level information. The number of pixels gradually decreases due to a reduction in the image dimensions,

Bit	Metrics	20% bit-flip injection	40% bit-flip injection	60% bit-flip injection	80% bit-flip injection
0	Precision	0.68523	0.68523	0.68523	0.68523
	Recall	0.57950	0.57950	0.57950	0.57950
	F1-score	0.62795	0.62795	0.62795	0.62795
1	Precision	0.68523	0.68523	0.68523	0.68523
	Recall	0.57950	0.57950	0.57950	0.57950
	F1-score	0.62795	0.62795	0.62795	0.62795
2	Precision	0.68523	0.68428	0.68523	0.68523
	Recall	0.57950	0.57950	0.57950	0.57950
	F1-score	0.62795	0.62795	0.62755	0.62795
3	Precision	0.68523	0.68523	0.68523	0.68523
	Recall	0.57950	0.57950	0.57950	0.57950
	F1-score	0.62795	0.62795	0.62795	0.62795
4	Precision	0.68523	0.68523	0.68523	0.68523
	Recall	0.57950	0.57950	0.57950	0.57950
	F1-score	0.62795	0.62795	0.62795	0.62795
5	Precision	0.68523	0.68523	0.68523	0.68523
	Recall	0.57950	0.57950	0.57950	0.57950
	F1-score	0.62795	0.62795	0.62795	0.62795
6	Precision	0.68523	0.68523	0.68523	0.68523
	Recall	0.57950	0.57950	0.57950	0.57950
	F1-score	0.62795	0.62795	0.62795	0.62795
7	Precision	0.68523	0.68523	0.68523	0.68523
	Recall	0.57950	0.57950	0.57950	0.57950
	F1-score	0.62795	0.62795	0.62795	0.62795

Table 4.8: Performance metrics for different percentages of bit-flip injection within layer extra 1 for MobileNetV2 backbone

but with a consequent increase in the number of channels. As we approach the extraction layers, injecting an error at this point means there is a higher probability of generating an error in the output. For this reason, the internal resilience of these intermediate activations is significantly reduced, to the point that even bit-flips of bit 4 can have catastrophic consequences on the network's output. Particularly, the least resilient layers are layer 8 and layer 10, depicted in Table 4.11 and Table 4.11.

LAYER 12 Layer 12 is the first layer used for feature map extraction. For this reason, an error injection starting from bit 3 can result in a significant reduction in

Bit	Metrics	20% bit-flip injection	40% bit-flip injection	60% bit-flip injection	80% bit-flip injection
0	Precision	0.86314	0.86043	0.86314	0.86314
	Recall	0.75029	0.74793	0.74676	0.74676
	F1-score	0.80277	0.80025	0.79999	0.79999
1	Precision	0.85986	0.86160	0.85888	0.85850
	Recall	0.74440	0.74793	0.74558	0.74322
	F1-score	0.79797	0.80075	0.79823	0.79671
2	Precision	0.85869	0.86103	0.85948	0.86401
	Recall	0.74440	0.74440	0.74204	0.74087
	F1-score	0.79747	0.79848	0.79646	0.79771
3	Precision	0.85986	0.86301	0.87168	0.87430
	Recall	0.74440	0.74204	0.73616	0.73733
	F1-score	0.79797	0.79797	0.79821	0.8
4	Precision	0.86501	0.87114	0.87060	0.87926
	Recall	0.73969	0.73262	0.72909	0.72909
	F1-score	0.79746	0.79590	0.79358	0.79716
5	Precision	0.87119	0.87517	0.88629	0.88888
	Recall	0.74087	0.72673	0.71613	0.69729
	F1-score	0.80076	0.79407	0.79218	0.78151
6	Precision	0.87411	0.89681	0.90016	0.90243
	Recall	0.72791	0.69611	0.65842	0.61012
	F1-score	0.79434	0.78381	0.76054	0.72803
7	Precision	0.89767	0.91013	0.89970	0.98333
	Recall	0.68197	0.56065	0.35924	0.13898
	F1-score	0.77510	0.69387	0.51346	0.24355

Table 4.9: Performance metrics for different percentages of bit-flip injection within layer 0 for VGG16 backbone

network performance. Results are displayed in Table 4.13

LAYER 13-14-15-16-17 These layers are crucial for extracting high-level features. Injecting an error at this point will not affect the deeper backbone of the network, so the ship detection up to layer 12 will not be impacted. Performance only degrades slightly on these layers, mainly for layers 14 and 16 due to errors in the last two most significant bits. Table 4.14 shows error injection for layer 14.

LAYER 18 This layer is the second feature map used for feature extraction in the output. Similarly, considering the first two most significant bits, the network's

Bit	Metrics	20% bit-flip injection	40% bit-flip injection	60% bit-flip injection	80% bit-flip injection
0	Precision	0.86178	0.86178	0.86024	0.85964
	Recall	0.74911	0.74911	0.74676	0.75029
	F1-score	0.80151	0.80151	0.79949	0.80125
1	Precision	0.85772	0.86062	0.85733	0.85752
	Recall	0.74558	0.74911	0.75029	0.75147
	F1-score	0.79773	0.80100	0.80025	0.80100
2	Precision	0.85444	0.85234	0.85140	0.84574
	Recall	0.74676	0.74793	0.74911	0.74911
	F1-score	0.79698	0.79673	0.79699	0.79450
3	Precision	0.85234	0.83552	0.82490	0.82558
	Recall	0.74793	0.74793	0.74911	0.75265
	F1-score	0.79673	0.78931	0.78518	0.78743
4	Precision	0.83464	0.81241	0.81419	0.81712
	Recall	0.74911	0.73969	0.74322	0.74204
	F1-score	0.78957	0.77435	0.77709	0.77777
5	Precision	0.83679	0.83207	0.83619	0.82968
	Recall	0.71260	0.64782	0.63133	0.62544
	F1-score	0.76972	0.72847	0.71946	0.71323
6	Precision	0.88235	0.91701	0.89444	0.89673
	Recall	0.45936	0.26030	0.18963	0.19434
	F1-score	0.60418	0.40550	0.31292	0.31945
7	Precision	1.0	0.0	0.0	0.0
	Recall	0.00117	0.0	0.0	0.0
	F1-score	0.00235	0.0	0.0	0.0

Table 4.10: Performance metrics for different percentages of bit-flip injection within layer 4 for VGG16 backbone

performance degrades significantly (Table 4.15).

LAYER EXTRA An error in this part of the network does not significantly degrade the network's output. Table 4.16 illustrates performance metrics for VGG16 backbone after bit-flip injection for layer extra 0.

4.3 Conclusions and future works

The discussion highlights how the presence of bit-flip errors can lead to catastrophic performance degradation in a DNN-based object recognition application. Single

Bit	Metrics	20% bit-flip injection	40% bit-flip injection	60% bit-flip injection	80% bit-flip injection
0	Precision	0.85967	0.86538	0.86301	0.86556
	Recall	0.74322	0.74204	0.74204	0.74322
	F1-score	0.79722	0.79898	0.79797	0.79974
1	Precision	0.86084	0.86556	0.86419	0.86164
	Recall	0.74322	0.74322	0.74204	0.74087
	F1-score	0.79772	0.79974	0.79847	0.79670
2	Precision	0.86084	0.86331	0.87023	0.86638
	Recall	0.74322	0.72909	0.72673	0.72555
	F1-score	0.79772	0.79054	0.79204	0.78974
3	Precision	0.87696	0.89805	0.89969	0.90186
	Recall	0.72202	0.70553	0.68669	0.68197
	F1-score	0.79198	0.79023	0.77889	0.77665
4	Precision	0.940740	0.939571	0.93849	0.93849
	Recall	0.59835	0.56772	0.55712	0.55712
	F1-score	0.73146	0.70778	0.69918	0.69918
5	Precision	0.96359	0.99203	0.99029	0.99074
	Recall	0.46760	0.29328	0.24028	0.252061
	F1-score	0.62965	0.45272	0.38672	0.40187
6	Precision	0.97368	1.0	1.0	1.0
	Recall	0.08716	0.00588	0.00353	0.00353
	F1-score	0.15999	0.01170	0.00704	0.00704
7	Precision	0.0	0.0	0.0	0.0
	Recall	0.0	0.0	0.0	0.0
	F1-score	0.0	0.0	0.0	0.0

Table 4.11: Performance metrics for different percentages of bit-flip injection within layer 8 for VGG16 backbone

bit-flip errors can cause severe issues not only in user memory but also in the CRAM, altering the platform's logic and potentially resulting in critical consequences for the entire system. The current best solutions for configuration memory include redundancy modules (TMR), error recognition and correction modules (ECC or SEC-DED) and partial static or dynamic reconfiguration. However, these methodologies come with a high overhead in terms of on-chip area occupation, making it beneficial to implement sustainable solutions for extreme applications of this nature. It would be interesting to consider this aspect for future work. As described in the preceding chapters, the study allows for a comparison of the characteristics of two different backbones, assessing which one, in terms of memory

Bit	Metrics	20% bit-flip injection	40% bit-flip injection	60% bit-flip injection	80% bit-flip injection
0	Precision	0.86084	0.86998	0.88135	0.88825
	Recall	0.74322	0.74087	0.73498	0.73027
	F1-score	0.79772	0.80025	0.80154	0.80155
1	Precision	0.86221	0.86842	0.87746	0.88904
	Recall	0.74440	0.73851	0.73380	0.72673
	F1-score	0.79898	0.79821	0.79923	0.79974
2	Precision	0.87257	0.89306	0.90694	0.91527
	Recall	0.74204	0.72791	0.72320	0.71260
	F1-score	0.80203	0.80207	0.80471	0.80132
3	Precision	0.88825	0.91376	0.93174	0.94256
	Recall	0.73027	0.71142	0.69140	0.65724
	F1-score	0.80155	0.8	0.79377	0.77446
4	Precision	0.91666	0.93781	0.95229	0.95739
	Recall	0.71260	0.65724	0.58775	0.50294
	F1-score	0.80185	0.77285	0.72687	0.65945
5	Precision	0.94097	0.96163	0.95104	1.0
	Recall	0.65724	0.47232	0.16018	0.01884
	F1-score	0.77392	0.63349	0.27419	0.03699
6	Precision	0.95433	0.76923	0.00117	0.0
	Recall	0.49234	0.02355	0.00117	0.0
	F1-score	0.64957	0.04571	0.00235	0.0
7	Precision	0.85714	0.0	0.0	0.0
	Recall	0.02826	0.0	0.0	0.0
	F1-score	0.05473	0.0	0.0	0.0

Table 4.12: Performance metrics for different percentages of bit-flip injection within layer 10 for VGG16 backbone

saving and performance degradation, is better suited for this type of application. To precisely evaluate which network undergoes the greatest performance degradation, it is necessary to delve into the specifics of the application and understand which layers for each backbone are most at risk of generating a significant degradation in performance metrics. Regarding the MobileNetV2 network, the first extraction layer is the sixth one. In case of bit-flip error accumulation in the initial layers, the network's performance is completely degraded, considering both bit 6 and bit 7. If bit-flip errors accumulate after the sixth layer, the network can still recognize some objects in the images. In this case, due to the application's design, the recall does not exceed values lower than 15% (a reduction of recall by approximately

Bit	Metrics	20% bit-flip injection	40% bit-flip injection	60% bit-flip injection	80% bit-flip injection
0	Precision	0.87552	0.88904	0.72084	0.90433
	Recall	0.73733	0.72673	0.72084	0.71260
	F1-score	0.80051	0.79974	0.80052	0.79710
1	Precision	0.89177	0.90433	0.92187	0.93537
	Recall	0.72791	0.71260	0.69493	0.68197
	F1-score	0.80155	0.79710	0.79247	0.78882
2	Precision	0.90854	0.93821	0.94290	0.94821
	Recall	0.71378	0.67962	0.64193	0.62544
	F1-score	0.79947	0.78825	0.76384	0.75372
3	Precision	0.93944	0.95660	0.96653	0.96680
	Recall	0.67608	0.62308	0.57832	0.54888
	F1-score	0.78630	0.75463	0.72365	0.70022
4	Precision	0.95825	0.97881	0.97959	0.98165
	Recall	0.62190	0.54416	0.50883	0.50412
	F1-score	0.75428	0.69947	0.66976	0.66614
5	Precision	0.98218	0.98971	0.98694	0.98941
	Recall	0.51943	0.45347	0.44522	0.44051
	F1-score	0.67950	0.62197	0.61363	0.60961
6	Precision	1.0	1.0	1.0	1.0
	Recall	0.34864	0.10836	0.07302	0.09893
	F1-score	0.51703	0.19553	0.13611	0.18006
7	Precision	0.0	0.0	0.0	0.0
	Recall	0.0	0.0	0.0	0.0
	F1-score	0.0	0.0	0.0	0.0

Table 4.13: Performance metrics for different percentages of bit-flip injection within layer 12 for VGG16 backbone

74% of the initial metrics). The second extraction layer is the thirteenth, so in case of bit-flip error accumulation of the first two most significant bits in the subsequent layers, the number of recognized objects will increase, reaching recall values of around 49% (a reduction of recall by approximately 15% compared to initial values). Finally, once the last layer within the backbone is extracted (layer 18), the performance remains very similar to the initial values, even in case of error accumulation in the most significant bit. On the other hand, the VGG16 network, having numerous convolutional layers and being deeper, allows for learning more complex features. This inevitably leads to better distinguishing one object from another and addressing more complex problems. Deeper networks also tend to

Bit	Metrics	20% bit-flip injection	40% bit-flip injection	60% bit-flip injection	80% bit-flip injection
0	Precision	0.86160	0.86178	0.86178	0.86178
	Recall	0.74793	0.74911	0.74911	0.74911
	F1-score	0.80075	0.80151	0.80151	0.80151
1	Precision	0.86277	0.86493	0.86493	0.86867
	Recall	0.74793	0.74676	0.74676	0.74793
	F1-score	0.80126	0.80151	0.80151	0.80379
2	Precision	0.86258	0.86748	0.87004	0.87260
	Recall	0.74676	0.74793	0.74911	0.75029
	F1-score	0.80050	0.80328	0.80506	0.80683
3	Precision	0.86885	0.87260	0.87362	0.87845
	Recall	0.74911	0.75029	0.74911	0.74911
	F1-score	0.80455	0.80683	0.80659	0.80864
4	Precision	0.87140	0.87966	0.87988	0.88494
	Recall	0.75029	0.74911	0.74204	0.73380
	F1-score	0.80632	0.80916	0.80511	0.80231
5	Precision	0.88311	0.88260	0.88252	0.87519
	Recall	0.72084	0.73498	0.70789	0.66077
	F1-score	0.79377	0.80205	0.78562	0.75302
6	Precision	0.88169	0.87883	0.85792	0.82471
	Recall	0.73733	0.67491	0.55477	0.43227
	F1-score	0.80307	0.76349	0.67381	0.56723
7	Precision	0.87856	0.81384	0.75925	0.75776
	Recall	0.69022	0.40164	0.28975	0.28739
	F1-score	0.77308	0.53785	0.41943	0.41673

Table 4.14: Performance metrics for different percentages of bit-flip injection within layer 14 for VGG16 backbone

generalize better on unseen data compared to shallower networks. This is due to their ability to learn more generic data representations that can be applied to a wider variety of examples. Digging deeper into the network's performance, the network completely degrades up to layer 12, in some cases even for error injections in the last three most significant bits. In case of errors in subsequent feature maps, the network achieves recall values of 29% for the MSB (a reduction of the metric by approximately 61% of the initial metrics). The last extraction layer of the backbone then allows further performance improvements, reaching values similar to those expected in case of errors in the extra layers. From this analysis, a comprehensive understanding of how the backbones behave in case of bit-flip error accumulation

Bit	Metrics	20% bit-flip injection	40% bit-flip injection	60% bit-flip injection	80% bit-flip injection
0	Precision	0.86160	0.86376	0.86612	0.86867
	Recall	0.74793	0.74676	0.74676	0.74793
	F1-score	0.80075	0.80101	0.80202	0.80379
1	Precision	0.86376	0.86867	0.86867	0.86986
	Recall	0.74676	0.74793	0.74793	0.74793
	F1-score	0.80101	0.80379	0.80379	0.80430
2	Precision	0.86867	0.86986	0.87242	0.87327
	Recall	0.74793	0.74793	0.74911	0.74676
	F1-score	0.80379	0.80430	0.80608	0.80507
3	Precision	0.86867	0.87310	0.87430	0.875
	Recall	0.74793	0.74558	0.73733	0.73380
	F1-score	0.80379	0.80432	0.73733	0.79820
4	Precision	0.87707	0.87377	0.87392	0.70082
	Recall	0.74793	0.73380	0.71849	0.70082
	F1-score	0.80737	0.79769	0.78862	0.77828
5	Precision	0.87711	0.70082	0.87366	0.86816
	Recall	0.73144	0.70082	0.67608	0.63604
	F1-score	0.79768	0.77828	0.76228	0.73419
6	Precision	0.87536	0.86837	0.85370	0.82751
	Recall	0.70318	0.63722	0.54299	0.44640
	F1-score	0.77988	0.73505	0.66378	0.57995
7	Precision	0.86858	0.82826	0.36749	0.77714
	Recall	0.63839	0.44876	0.36749	0.32037
	F1-score	0.73591	0.58212	0.50363	0.45371

Table 4.15: Performance metrics for different percentages of bit-flip injection within layer 18 for VGG16 backbone

can be obtained. The MobileNetV2 network maintains a high level of performance despite having a much lower number of parameters compared to VGG16 (x12 smaller). Additionally, the computational cost of the MobileNetV2 network in terms of MAC operations is about x50 lower than the VGG16 one. Therefore, the choice of which network to use can be based on the required performance, the available memory resources and the computational costs. However, resilience for both backbones against this type of error can be improved by using techniques such as fault-aware training, combined with ECC, TMR, and partial reconfiguration techniques (scrubbing can also be performed for BRAMs, not only for configuration memory). Selective TMR applications for layers most sensitive to these errors also

Bit	Metrics	20% bit-flip injection	40% bit-flip injection	60% bit-flip injection	80% bit-flip injection
0	Precision	0.86178	0.86178	0.86178	0.86178
	Recall	0.74911	0.74911	0.74911	0.74911
	F1-score	0.80151	0.80151	0.80151	0.80151
1	Precision	0.86178	0.86178	0.86178	0.86295
	Recall	0.74911	0.74911	0.74911	0.74911
	F1-score	0.80151	0.80151	0.80151	0.80151
2	Precision	0.86178	0.86295	0.86295	0.86295
	Recall	0.74911	0.74911	0.74911	0.74911
	F1-score	0.80151	0.80201	0.80201	0.80201
3	Precision	0.86295	0.86295	0.86295	0.86295
	Recall	0.74911	0.74911	0.74911	0.74911
	F1-score	0.80201	0.80201	0.80201	0.80201
4	Precision	0.86295	0.86295	0.86295	0.86376
	Recall	0.74911	0.74911	0.74911	0.74676
	F1-score	0.80201	0.80201	0.80201	0.80101
5	Precision	0.86295	0.86376	0.86282	0.86225
	Recall	0.74911	0.74676	0.74087	0.73733
	F1-score	0.80201	0.80101	0.79721	0.79492
6	Precision	0.86413	0.86225	0.86033	0.85795
	Recall	0.74911	0.73733	0.72555	0.71142
	F1-score	0.80252	0.79492	0.78722	0.77784
7	Precision	0.86206	0.85775	0.85631	0.85419
	Recall	0.73616	0.71024	0.69493	0.68315
	F1-score	0.79415	0.77706	0.76723	0.75916

Table 4.16: Performance metrics for different percentages of bit-flip injection within layer extra 0 for VGG16 backbone

appear interesting, as they can help limit bit-flip errors for the most sensitive layers. The development of resilience techniques is also left as future work.

Appendix A

Training augmentation transforms

```
def intersect(box_a, box_b):
    max_xy = np.minimum(box_a[:, 2:], box_b[2:])
    min_xy = np.maximum(box_a[:, :2], box_b[:2])
    inter = np.clip((max_xy - min_xy), a_min=0, a_max=np.inf)
    return inter[:, 0] * inter[:, 1]

def jaccard_numpy(box_a, box_b):
    inter = intersect(box_a, box_b)
    area_a = ((box_a[:, 2]-box_a[:, 0]) *
              (box_a[:, 3]-box_a[:, 1]))  # [A,B]
    area_b = ((box_b[2]-box_b[0]) *
              (box_b[3]-box_b[1]))  # [A,B]
    union = area_a + area_b - inter
    return inter / union  # [A,B]
```

```
class Custom_lambda_transform(object):
    def __init__(self,std):
        self.std=std

    def __call__(self, image, boxes=None, labels=None):
        image /= self.std
        return image.astype(np.float32), boxes, labels
```

```
class ConvertFromInts(object):
    def __call__(self, image, boxes=None, labels=None):
        return image.astype(np.float32), boxes, labels
```

```
class SubtractMeans(object):
    def __init__(self, mean):
        self.mean = np.array(mean, dtype=np.float32)

    def __call__(self, image, boxes=None, labels=None):
        image = image.astype(np.float32)
        image -= self.mean
        return image.astype(np.float32), boxes, labels
```

```
class ToAbsoluteCoords(object):
    def __call__(self, image, boxes=None, labels=None):
        height, width, channels = image.shape
        boxes[:, 0] *= width
        boxes[:, 2] *= width
        boxes[:, 1] *= height
        boxes[:, 3] *= height

        return image, boxes, labels

class ToPercentCoords(object):
    def __call__(self, image, boxes=None, labels=None):
        height, width, channels = image.shape
        boxes[:, 0] /= width
        boxes[:, 2] /= width
        boxes[:, 1] /= height
        boxes[:, 3] /= height

        return image, boxes, labels
```

```
class Resize(object):
    def __init__(self, size=image_size):
        self.size = size

    def __call__(self, image, boxes=None, labels=None):
        image = cv2.resize(image, (self.size,
                                  self.size))
        return image, boxes, labels
```

```
class RandomSaturation(object):
    def __init__(self, lower=0.5, upper=1.5):
        self.lower = lower
        self.upper = upper
        assert self.upper >= self.lower, "contrast upper must be >=
  lower."
        assert self.lower >= 0, "contrast lower must be non-negative
  ."

    def __call__(self, image, boxes=None, labels=None):
        if random.randint(2):
            image[:, :, 1] *= random.uniform(self.lower, self.upper)

        return image, boxes, labels
```

```
class RandomHue(object):
    def __init__(self, delta=18.0):
        assert delta >= 0.0 and delta <= 360.0
        self.delta = delta

    def __call__(self, image, boxes=None, labels=None):
        if random.randint(2):
            image[:, :, 0] += random.uniform(-self.delta, self.delta)
            image[:, :, 0][image[:, :, 0] > 360.0] -= 360.0
            image[:, :, 0][image[:, :, 0] < 0.0] += 360.0
        return image, boxes, labels
```

```
class RandomLightingNoise(object):
    def __init__(self):
        self.perms = ((0, 1, 2), (0, 2, 1),
                      (1, 0, 2), (1, 2, 0),
                      (2, 0, 1), (2, 1, 0))

    def __call__(self, image, boxes=None, labels=None):
        if random.randint(2):
            swap = self.perms[random.randint(len(self.perms))]
            shuffle = SwapChannels(swap)  # shuffle channels
            image = shuffle(image)
        return image, boxes, labels
```

```python
class ConvertColor(object):
    def __init__(self, current, transform):
        self.transform = transform
        self.current = current

    def __call__(self, image, boxes=None, labels=None):
        if self.current == 'BGR' and self.transform == 'HSV':
            image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
        elif self.current == 'RGB' and self.transform == 'HSV':
            image = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)
        elif self.current == 'BGR' and self.transform == 'RGB':
            image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        elif self.current == 'HSV' and self.transform == 'BGR':
            image = cv2.cvtColor(image, cv2.COLOR_HSV2BGR)
        elif self.current == 'HSV' and self.transform == "RGB":
            image = cv2.cvtColor(image, cv2.COLOR_HSV2RGB)
        else:
            raise NotImplementedError
        return image, boxes, labels
```

```python
class RandomContrast(object):
    def __init__(self, lower=0.5, upper=1.5):
        self.lower = lower
        self.upper = upper
        assert self.upper >= self.lower, "contrast upper must be >=
    lower."
        assert self.lower >= 0, "contrast lower must be non-negative
    ."

    # expects float image
    def __call__(self, image, boxes=None, labels=None):
        if random.randint(2):
            alpha = random.uniform(self.lower, self.upper)
            image *= alpha
        return image, boxes, labels
```

```python
class RandomBrightness(object):
    def __init__(self, delta=32):
        assert delta >= 0.0
        assert delta <= 255.0
        self.delta = delta

    def __call__(self, image, boxes=None, labels=None):
        if random.randint(2):
            delta = random.uniform(-self.delta, self.delta)
            image += delta
        return image, boxes, labels
```

```
class ToCV2Image(object):
    def __call__(self, tensor, boxes=None, labels=None):
        return tensor.cpu().numpy().astype(np.float32).transpose((1,
    2, 0)), boxes, labels
```

```
class ToTensor(object):
    def __call__(self, cvimage, boxes=None, labels=None):
        return torch.from_numpy(cvimage.astype(np.float32)).permute
    (2, 0, 1), boxes, labels
```

```
class RandomSampleCrop(object):

    def __init__(self):
        self.sample_options = (
            # using entire original input image
            None,
            # sample a patch s.t. MIN jaccard w/ obj in
    .1,.3,.4,.7,.9
            (0.1, None),
            (0.3, None),
            (0.7, None),
            (0.9, None),
            # randomly sample a patch
            (None, None),
        )
    def __call__(self, image, boxes=None, labels=None):
        height, width, _ = image.shape
        while True:
            # randomly choose a mode
            np.warnings.filterwarnings('ignore', category=np.
    VisibleDeprecationWarning)
            mode = random.choice(self.sample_options)
            if mode is None:# or not boxes.any():
                return image, boxes, labels

            min_iou, max_iou = mode
            if min_iou is None:
                min_iou = float('-inf')
            if max_iou is None:
```

```
            max_iou = float('inf')

        # max trails (50)
        for _ in range(50):
            current_image = image
            w = random.uniform(0.3 * width, width)
            h = random.uniform(0.3 * height, height)
            # aspect ratio constraint b/t .5 & 2
            if h / w < 0.5 or h / w > 2:
                continue
            left = random.uniform(width - w)
            top = random.uniform(height - h)
            # convert to integer rect x1,y1,x2,y2
            rect = np.array([int(left), int(top), int(left+w), int(top+h)])
            # calculate IoU (jaccard overlap) b/t the cropped and gt boxes
            overlap = jaccard_numpy(boxes, rect)
            # is min and max overlap constraint satisfied? if not try again
            if overlap.min() < min_iou and max_iou < overlap.max():
                continue
            # cut the crop from the image
            current_image = current_image[rect[1]:rect[3], rect[0]:rect[2],:]
            # keep overlap with gt box IF center in sampled patch
            centers = (boxes[:, :2] + boxes[:, 2:]) / 2.0
            # mask in all gt boxes that above and to the left of centers
            m1 = (rect[0] < centers[:, 0]) * (rect[1] < centers[:, 1])
            # mask in all gt boxes that under and to the right of centers
            m2 = (rect[2] > centers[:, 0]) * (rect[3] > centers[:, 1])
            # mask in that both m1 and m2 are true
            mask = m1 * m2
            # have any valid boxes? try again if not
            if not mask.any():
                continue
            # take only matching gt boxes
            current_boxes = boxes[mask, :].copy()
            # take only matching gt labels
            current_labels = labels[mask]
            # should we use the box left and top corner or the crop's
            current_boxes[:, :2] = np.maximum(current_boxes[:,:2], rect[:2])
```

```
                # adjust to crop (by substracting crop's left,top)
                current_boxes[:, :2] -= rect[:2]
                current_boxes[:, 2:] = np.minimum(current_boxes[:, 2:],
                                                  rect[2:])
                # adjust to crop (by substracting crop's left,top)
                current_boxes[:, 2:] -= rect[:2]
                return current_image, current_boxes, current_labels
```

```
class Expand(object):
    def __init__(self, mean):
        self.mean = mean

    def __call__(self, image, boxes, labels):
        if random.randint(2):# or not boxes.any():
            return image, boxes, labels

        height, width, depth = image.shape
        ratio = random.uniform(1, 4)
        left = random.uniform(0, width*ratio - width)
        top = random.uniform(0, height*ratio - height)
        expand_image = np.zeros(
            (int(height*ratio), int(width*ratio), depth),
            dtype=image.dtype)
        expand_image[:, :, :] = self.mean
        expand_image[int(top):int(top + height),
                     int(left):int(left + width)] = image
        image = expand_image
        boxes = boxes.copy()
        boxes[:, :2] += (int(left), int(top))
        boxes[:, 2:] += (int(left), int(top))

        return image, boxes, labels
```

```
class RandomMirror(object):
    def __call__(self, image, boxes, classes):
        _, width, _ = image.shape
        if random.randint(2):#or not boxes.any():
            image = image[:, ::-1]
            boxes = boxes.copy()
            boxes[:, 0::2] = width - boxes[:, 2::-2]
        return image, boxes, classes
```

```
class SwapChannels(object):
    """Transforms a tensorized image by swapping the channels in the
   order
      specified in the swap tuple.
    Args:
        swaps (int triple): final order of channels
            eg: (2, 1, 0)
    """

    def __init__(self, swaps):
        self.swaps = swaps

    def __call__(self, image):
        """
        Args:
            image (Tensor): image tensor to be transformed
        Return:
            a tensor with channels swapped according to swap
        """
        # if torch.is_tensor(image):
        #     image = image.data.cpu().numpy()
        # else:
        #     image = np.array(image)
        image = image[:, :, self.swaps]
        return image
```

```
class PhotometricDistort(object):
    def __init__(self):
        self.pd = [
            RandomContrast(),  # RGB
            ConvertColor(current="RGB", transform='HSV'),  # HSV
            RandomSaturation(),  # HSV
            RandomHue(),  # HSV
            ConvertColor(current='HSV', transform='RGB'),  # RGB
            RandomContrast()  # RGB
        ]
        self.rand_brightness = RandomBrightness()
        self.rand_light_noise = RandomLightingNoise()

    def __call__(self, image, boxes, labels):
        im = image.copy()
        im, boxes, labels = self.rand_brightness(im, boxes, labels)
        if random.randint(2):# or not boxes.any():
            distort = Compose(self.pd[:-1])
        else:
            distort = Compose(self.pd[1:])
        im, boxes, labels = distort(im, boxes, labels)
        return self.rand_light_noise(im, boxes, labels)
```

Code implementation: [23].

Appendix B

MobileNetV2

Inverted residual module and MobileNetV2 definition:

```
def conv_bn(inp, oup, stride):
    return nn.Sequential(
        nn.Conv2d(inp, oup, 3, stride, 1, bias=False),
        nn.BatchNorm2d(oup),
        nn.ReLU6(inplace=True)
    )
```

```
def conv_1x1_bn(inp, oup):
    return nn.Sequential(
        nn.Conv2d(inp, oup, 1, 1, 0, bias=False),
        nn.BatchNorm2d(oup),
        nn.ReLU6(inplace=True)
    )
```

```
class InvertedResidual(nn.Module):
    def __init__(self, inp, oup, stride, expand_ratio):
        super(InvertedResidual, self).__init__()
        self.stride = stride
        assert stride in [1, 2]

        hidden_dim = round(inp * expand_ratio)
        self.use_res_connect = self.stride == 1 and inp == oup

        if expand_ratio == 1:
            self.conv = nn.Sequential(
                # dw
                nn.Conv2d(hidden_dim, hidden_dim, 3, stride, 1,
    groups=hidden_dim, bias=False),
                nn.BatchNorm2d(hidden_dim),
                nn.ReLU6(inplace=True),
                # pw-linear
                nn.Conv2d(hidden_dim, oup, 1, 1, 0, bias=False),
                nn.BatchNorm2d(oup),
            )
        else:
            self.conv = nn.Sequential(
                # pw
                nn.Conv2d(inp, hidden_dim, 1, 1, 0, bias=False),
                nn.BatchNorm2d(hidden_dim),
                nn.ReLU6(inplace=True),
                # dw
                nn.Conv2d(hidden_dim, hidden_dim, 3, stride, 1,
    groups=hidden_dim, bias=False),
                nn.BatchNorm2d(hidden_dim),
                nn.ReLU6(inplace=True),
                # pw-linear
                nn.Conv2d(hidden_dim, oup, 1, 1, 0, bias=False),
                nn.BatchNorm2d(oup),
            )

    def forward(self, x):
        if self.use_res_connect:
            return x + self.conv(x)
        else:
            return self.conv(x)
```

```
class MobileNetV2(nn.Module):
    def __init__(self, n_class=2, input_size=512, width_mult=1.0):
        super(MobileNetV2, self).__init__()
        block = InvertedResidual
        min_depth = 16
        input_channel = 32
        last_channel = 1280
        interverted_residual_setting = [
            # t, c, n, s
            [1, 16, 1, 1],
            [6, 24, 2, 2],
            [6, 32, 3, 2],
            [6, 64, 4, 2],
            [6, 96, 3, 1],
            [6, 160, 3, 2],
            [6, 320, 1, 1],
        ]

        # building first layer
        assert input_size % 32 == 0
        input_channel = int(input_channel * width_mult) if width_mult >= 1.0 else input_channel
        self.last_channel = int(last_channel * width_mult) if width_mult > 1.0 else last_channel
        self.features = [conv_bn(3, input_channel, 2)]
        # building inverted residual blocks
        for t, c, n, s in interverted_residual_setting:
            output_channel = max(int(c * width_mult), min_depth)
            for i in range(n):
                if i == 0:
                    self.features.append(block(input_channel, output_channel, s, expand_ratio=t))
                else:
                    self.features.append(block(input_channel, output_channel, 1, expand_ratio=t))
                input_channel = output_channel
        # building last several layers
        self.features.append(conv_1x1_bn(input_channel, self.last_channel))
        # make it nn.Sequential
        self.features = nn.Sequential(*self.features)

        # building classifier
        self.classifier = nn.Sequential(
            nn.Dropout(0.2),
            nn.Linear(self.last_channel, n_class),
        )

        self._initialize_weights()

    def forward(self, x):
        x = self.features(x)
        x = x.mean(3).mean(2)
        x = self.classifier(x)
        return x

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_(0, math.sqrt(2. / n))
                if m.bias is not None:
                    m.bias.data.zero_()
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()
            elif isinstance(m, nn.Linear):
                n = m.weight.size(1)
                m.weight.data.normal_(0, 0.01)
                m.bias.data.zero_()
```

Separable convolution and model generation:

```
def SeperableConv2d(in_channels, out_channels, kernel_size=1, stride
    =1, padding=0, onnx_compatible=False):
    """Replace Conv2d with a depthwise Conv2d and Pointwise Conv2d.
    """
    ReLU = nn.ReLU if onnx_compatible else nn.ReLU6
    return Sequential(
        Conv2d(in_channels=in_channels, out_channels=in_channels,
    kernel_size=kernel_size,
               groups=in_channels, stride=stride, padding=padding),
        BatchNorm2d(in_channels),
        ReLU(),
        Conv2d(in_channels=in_channels, out_channels=out_channels,
    kernel_size=1),
    )
```

```
def create_mobilenetv2_ssd_lite(num_classes, width_mult=1.0,
    use_batch_norm=True, onnx_compatible=False, is_test=False):
    base_net = MobileNetV2(width_mult=width_mult).features
    source_layer_indexes = [
        GraphPath(7, 'conv' , 3),
        GraphPath(14, 'conv', 3), 19,
    ]
    extras = ModuleList([
        InvertedResidual(1280, 512, stride=2, expand_ratio=0.2),
        InvertedResidual(512, 256, stride=2, expand_ratio=0.25),
        InvertedResidual(256, 256, stride=2, expand_ratio=0.5),
        InvertedResidual(256, 64, stride=2, expand_ratio=0.25)
    ])
    regression_headers = ModuleList([
        SeperableConv2d(in_channels=round(192 * width_mult),
    out_channels=4 * 4,kernel_size=3, padding=1, onnx_compatible=False
    ),
        SeperableConv2d(in_channels=round(576 * width_mult),
    out_channels=6 * 4,
                        kernel_size=3, padding=1, onnx_compatible=
    False),
        SeperableConv2d(in_channels=1280, out_channels=6 * 4,
    kernel_size=3, padding=1, onnx_compatible=False),
        SeperableConv2d(in_channels=512, out_channels=6 * 4, kernel_size=3, padding=1, onnx_compatible=False),
        SeperableConv2d(in_channels=256, out_channels=6 * 4, kernel_size=3, padding=1, onnx_compatible=False),
        SeperableConv2d(in_channels=256, out_channels=6 * 4, kernel_size=3, padding=1, onnx_compatible=False),
        Conv2d(in_channels=64, out_channels=6 * 4, kernel_size=1),
    ])
    classification_headers = ModuleList([
        SeperableConv2d(in_channels=round(192 * width_mult), out_channels=4 * num_classes,kernel_size=3, padding=1),
        SeperableConv2d(in_channels=round(576 * width_mult), out_channels=6 * num_classes, kernel_size=3, padding=1),
        SeperableConv2d(in_channels=1280, out_channels=6 * num_classes, kernel_size=3, padding=1),
        SeperableConv2d(in_channels=512, out_channels=6 * num_classes, kernel_size=3, padding=1),
        SeperableConv2d(in_channels=256, out_channels=6 * num_classes, kernel_size=3, padding=1),
        SeperableConv2d(in_channels=256, out_channels=6 * num_classes, kernel_size=3, padding=1),
        Conv2d(in_channels=64, out_channels=6 * num_classes, kernel_size=1),
    ])

    return SSD(num_classes, base_net, source_layer_indexes,
               extras, classification_headers, regression_headers, center_variance,size_variance,is_test=is_test)
```

Appendix C

Quantized MobileNetV2

The application in this case is the same. Only backbone is changed:

```
def quant_conv_bn(inp,oup,stride):
    return nn.Sequential(
        QConv2d(inp, oup, 3, stride, 1, bias=False, weight_quant = Int8WeightPerTensorFixedPoint, bias_quant = Int16Bias, input_quant = Int8ActPerTensorFixedPoint, output_quant = Int8ActPerTensorFixedPoint,return_quant_tensor=True),
        nn.BatchNorm2d(oup),
        QuantReLU(act_quant=CommonUintActQuant,
            bit_width=8,
            per_channel_broadcastable_shape=(1, oup, 1, 1),
            scaling_stats_permute_dims=(1, 0, 2, 3),
            scaling_per_output_channel=False,
            return_quant_tensor=True) )
```

```
def quant_conv_1x1_bn(inp, oup):
    return nn.Sequential(
        QConv2d(inp, oup, 1, 1, 0, bias=False,  weight_quant = Int8WeightPerTensorFixedPoint, bias_quant = Int16Bias, input_quant = Int8ActPerTensorFixedPoint, output_quant = Int8ActPerTensorFixedPoint, return_quant_tensor=True),
        nn.BatchNorm2d(oup),
        QuantReLU(act_quant=CommonUintActQuant,
            bit_width=8,
            per_channel_broadcastable_shape=(1, oup, 1, 1),
            scaling_stats_permute_dims=(1, 0, 2, 3),
            scaling_per_output_channel=False,
            return_quant_tensor=True)
    )
```

```
class InvertedResidual(nn.Module):
    def __init__(self, inp, oup, stride, expand_ratio):
        super(InvertedResidual, self).__init__()
        self.stride = stride
        assert stride in [1, 2]

        hidden_dim = round(inp * expand_ratio)
        self.use_res_connect = self.stride == 1 and inp == oup

        if expand_ratio == 1:
            self.conv = nn.Sequential(
                # dw
                QConv2d(hidden_dim, hidden_dim, 3, stride, 1, groups=hidden_dim, bias=False,  weight_quant = Int8WeightPerTensorFixedPoint, bias_quant = Int16Bias, input_quant = Int8ActPerTensorFixedPoint, output_quant = Int8ActPerTensorFixedPoint, return_quant_tensor=True),
                nn.BatchNorm2d(hidden_dim),
                QuantReLU(act_quant=CommonUintActQuant,
                bit_width=8,
                per_channel_broadcastable_shape=(1, hidden_dim, 1, 1),
                scaling_stats_permute_dims=(1, 0, 2, 3),
                scaling_per_output_channel=False,
                return_quant_tensor=True),
                # pw-linear
                QConv2d(hidden_dim, oup, 1, 1, 0, bias=False, weight_quant = Int8WeightPerTensorFixedPoint, bias_quant = Int16Bias, input_quant = Int8ActPerTensorFixedPoint, output_quant = Int8ActPerTensorFixedPoint,return_quant_tensor=True),
                nn.BatchNorm2d(oup),
            )
        else:
            self.conv = nn.Sequential(
                # pw
                QConv2d(inp, hidden_dim, 1, 1, 0, bias=False, weight_quant = Int8WeightPerTensorFixedPoint, bias_quant = Int16Bias, input_quant = Int8ActPerTensorFixedPoint, output_quant = Int8ActPerTensorFixedPoint, return_quant_tensor=True),
                nn.BatchNorm2d(hidden_dim),
                QuantReLU(act_quant=CommonUintActQuant,
                bit_width=8,
                per_channel_broadcastable_shape=(1, hidden_dim, 1, 1),
                scaling_stats_permute_dims=(1, 0, 2, 3),
                scaling_per_output_channel=False,
                return_quant_tensor=True),
                # dw
                QConv2d(hidden_dim, hidden_dim, 3, stride, 1, groups=hidden_dim, bias=False,  weight_quant = Int8WeightPerTensorFixedPoint, bias_quant = Int16Bias, input_quant = Int8ActPerTensorFixedPoint, output_quant = Int8ActPerTensorFixedPoint, return_quant_tensor=True),
                nn.BatchNorm2d(hidden_dim),
                QuantReLU(act_quant=CommonUintActQuant,
                bit_width=8,
                per_channel_broadcastable_shape=(1, hidden_dim, 1, 1),
                scaling_stats_permute_dims=(1, 0, 2, 3),
                scaling_per_output_channel=False,
                return_quant_tensor=True),
                # pw-linear
                QConv2d(hidden_dim, oup, 1, 1, 0, bias=False, weight_quant = Int8WeightPerTensorFixedPoint, bias_quant = Int16Bias, input_quant = Int8ActPerTensorFixedPoint, output_quant = Int8ActPerTensorFixedPoint, return_quant_tensor=True),
                nn.BatchNorm2d(oup),
            )
    def forward(self, x):
        if self.use_res_connect:
            return x + self.conv(x)
        else:
            return self.conv(x)
```

```
class MobileNetV2(nn.Module):
    def __init__(self, n_class=2, input_size=512, width_mult=1.0):
        super(MobileNetV2, self).__init__()
        block = InvertedResidual
        min_depth = 16
        input_channel = 32
        last_channel = 1280
        interverted_residual_setting = [
            # t, c, n, s
            [1, 16, 1, 1],
            [6, 24, 2, 2],
            [6, 32, 3, 2],
            [6, 64, 4, 2],
            [6, 96, 3, 1],
            [6, 160, 3, 2],
            [6, 320, 1, 1],
        ]

        # building first layer
        assert input_size % 32 == 0
        input_channel = int(input_channel * width_mult) if width_mult >= 1.0 else input_channel
        self.last_channel = int(last_channel * width_mult) if width_mult > 1.0 else last_channel
        self.features = [quant_conv_bn(3, input_channel, 2)]
        # building inverted residual blocks
        for t, c, n, s in interverted_residual_setting:
            output_channel = max(int(c * width_mult), min_depth)
            for i in range(n):
                if i == 0:
                    self.features.append(block(input_channel, output_channel, s, expand_ratio=t))
                else:
                    self.features.append(block(input_channel, output_channel, 1, expand_ratio=t))
                input_channel = output_channel
        # building last several layers
        self.features.append(quant_conv_1x1_bn(input_channel, self.last_channel))
        # make it nn.Sequential
        self.features = nn.Sequential(*self.features)

        # building classifier
        self.classifier = nn.Sequential(
            nn.Dropout(0.2),
            nn.Linear(self.last_channel, n_class),
        )

        self._initialize_weights()

    def forward(self, x):
        x = self.features(x)
        x = x.mean(3).mean(2)
        x = self.classifier(x)
        return x

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, QConv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_(0, math.sqrt(2. / n))
                if m.bias is not None:
                    m.bias.data.zero_()
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()
            elif isinstance(m, nn.Linear):
                n = m.weight.size(1)
                m.weight.data.normal_(0, 0.01)
                m.bias.data.zero_()
```

Appendix D

VGG16

VGG16 model architecture:

```
def vgg(cfg, batch_norm=True):
    layers = []
    in_channels = 3
    for v in cfg:
        if v == 'M':
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
        elif v == 'C':
            layers += [nn.MaxPool2d(kernel_size=2, stride=2,
    ceil_mode=True)]
        else:
            conv2d = nn.Conv2d(in_channels, v, kernel_size=3, padding
    =1)
            if batch_norm:
                layers += [conv2d, nn.BatchNorm2d(v), nn.ReLU(inplace
    =True)]
            else:
                layers += [conv2d, nn.ReLU(inplace=True)]
            in_channels = v
    pool5 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
    conv6 = nn.Conv2d(512, 1024, kernel_size=3, padding=6, dilation
    =6)
    conv7 = nn.Conv2d(1024, 1024, kernel_size=1)
    layers += [pool5, conv6,
               nn.ReLU(inplace=True), conv7, nn.ReLU(inplace=True)]
    return layers
```

```
def create_vgg_ssd(num_classes, is_test=False):
    vgg_config = [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'C', 512, 512, 512, 'M',
                  512, 512, 512]
    base_net = nn.ModuleList(vgg(vgg_config))
    #_initialize_weights(base_net)
    source_layer_indexes = [
        33,
        len(base_net),
    ]
    extras = nn.ModuleList([
        nn.Sequential(
            nn.Conv2d(in_channels=1024, out_channels=256, kernel_size=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=256, out_channels=512, kernel_size=3, stride=2, padding=1),
            nn.ReLU()
        ),
        nn.Sequential(
            nn.Conv2d(in_channels=512, out_channels=128, kernel_size=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, stride=2, padding=1),
            nn.ReLU()
        ),
        nn.Sequential(
            nn.Conv2d(in_channels=256, out_channels=128, kernel_size=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3,stride=2, padding=1),
            nn.ReLU()
        ),
        nn.Sequential(
            nn.Conv2d(in_channels=256, out_channels=128, kernel_size=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3,stride=2,padding=1),
            nn.ReLU()
        ),
        nn.Sequential(
            nn.Conv2d(in_channels=256, out_channels=128, kernel_size=1),
            nn.ReLU(),
            nn.Conv2d(in_channels=128, out_channels=256, kernel_size=4, padding=1),
            nn.ReLU()
        )
    ])

    regression_headers = nn.ModuleList([
        nn.Conv2d(in_channels=512, out_channels=4 * 4, kernel_size=3, padding=1),
        nn.Conv2d(in_channels=1024, out_channels=6 * 4, kernel_size=3, padding=1),
        nn.Conv2d(in_channels=512, out_channels=6 * 4, kernel_size=3, padding=1),
        nn.Conv2d(in_channels=256, out_channels=6 * 4, kernel_size=3, padding=1),
        nn.Conv2d(in_channels=256, out_channels=6 * 4, kernel_size=3, padding=1),
        nn.Conv2d(in_channels=256, out_channels=6 * 4, kernel_size=3, padding=1),
        nn.Conv2d(in_channels=256, out_channels=6 * 4, kernel_size=1), # TODO: change to kernel_size=1, padding=0?
    ])

    classification_headers = nn.ModuleList([
        nn.Conv2d(in_channels=512, out_channels=4 * num_classes, kernel_size=3, padding=1),
        nn.Conv2d(in_channels=1024, out_channels=6 * num_classes, kernel_size=3, padding=1),
        nn.Conv2d(in_channels=512, out_channels=6 * num_classes, kernel_size=3, padding=1),
        nn.Conv2d(in_channels=256, out_channels=6 * num_classes, kernel_size=3, padding=1),
        nn.Conv2d(in_channels=256, out_channels=6 * num_classes, kernel_size=3, padding=1),
        nn.Conv2d(in_channels=256, out_channels=6 * num_classes, kernel_size=3, padding=1),
        nn.Conv2d(in_channels=256, out_channels=6 * num_classes, kernel_size=1), # TODO: change to kernel_size=1, padding=0?
    ])

    return SSD(num_classes, base_net, source_layer_indexes,
               extras, classification_headers, regression_headers, center_variance,size_variance,is_test=is_test)
```

Appendix E

Quantized VGG16

```
def vgg(cfg, batch_norm=True):
    layers = []
    in_channels = 3
    for v in cfg:
        if v == 'M':
            layers += [QuantMaxPool2d(kernel_size=2, stride=2,
    return_quant_tensor=True)]
        elif v == 'C':
            layers += [QuantMaxPool2d(kernel_size=2, stride=2,
    ceil_mode=True,return_quant_tensor=True)]
        else:
            conv2d = QConv2d(in_channels, v, kernel_size= 3, padding=
     1, weight_quant = Int8WeightPerTensorFixedPoint, bias_quant =
    Int16Bias, input_quant = Int8ActPerTensorFixedPoint, output_quant
    = Int8ActPerTensorFixedPoint,return_quant_tensor=True)
            #conv2d = nn.Conv2d(in_channels, v, kernel_size=3,
    padding=1)
            if batch_norm:
                layers += [conv2d, nn.BatchNorm2d(v), QuantReLU(
    act_quant=CommonUintActQuant, bit_width=8,
    per_channel_broadcastable_shape=(1, v, 1, 1),
    scaling_stats_permute_dims=(1, 0, 2, 3),
    scaling_per_output_channel=False, return_quant_tensor=True,
    inplace=True)]
                #layers += [conv2d, nn.BatchNorm2d(v), nn.ReLU(
    inplace=True)]
            else:
                layers += [conv2d, QuantReLU(act_quant=CommonUintActQuant, bit_width=8, per_channel_broadcastable_shape=(1, v, 1, 1), scaling_stats_permute_dims=(1, 0, 2, 3), scaling_per_output_channel=False, return_quant_tensor=True, inplace=True)]
                #layers += [conv2d, nn.ReLU(inplace=True)]
            in_channels = v
    pool5 = QuantMaxPool2d(kernel_size=3, stride=1, padding=1, return_quant_tensor=True)
    conv6 = QConv2d(512, 1024, kernel_size= 3, padding= 6, dilation=6, weight_quant = Int8WeightPerTensorFixedPoint, bias_quant = Int16Bias, input_quant = Int8ActPerTensorFixedPoint, output_quant = Int8ActPerTensorFixedPoint, return_quant_tensor=True)
    conv7 = QConv2d(1024, 1024, kernel_size= 1, weight_quant = Int8WeightPerTensorFixedPoint, bias_quant = Int16Bias, input_quant = Int8ActPerTensorFixedPoint, output_quant = Int8ActPerTensorFixedPoint, return_quant_tensor=True)
    layers += [pool5, conv6, QuantReLU(act_quant=CommonUintActQuant, bit_width=8, per_channel_broadcastable_shape=(1, 1024, 1, 1), scaling_stats_permute_dims=(1, 0, 2, 3), scaling_per_output_channel=False, return_quant_tensor=True, inplace=True), conv7, QuantReLU(act_quant=CommonUintActQuant, bit_width=8, per_channel_broadcastable_shape=(1, 1024, 1, 1), scaling_stats_permute_dims=(1, 0, 2, 3), scaling_per_output_channel=False, return_quant_tensor=True, inplace=True)]
    return layers
```

```
def create_vgg_ssd(num_classes, is_test=False):
    vgg_config = [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'C', 512, 512, 512, 'M',
                  512, 512, 512]
    base_net = nn.ModuleList(vgg(vgg_config))

    source_layer_indexes = [
        33,
        len(base_net),
    ]
    extras = nn.ModuleList([
        nn.Sequential(
            QConv2d(1024, 256, kernel_size= 1, weight_quant = Int8WeightPerTensorFixedPoint, bias_quant = Int16Bias, input_quant = Int8ActPerTensorFixedPoint, output_quant = Int8ActPerTensorFixedPoint),
            QuantReLU(act_quant=CommonUintActQuant, bit_width=8, per_channel_broadcastable_shape=(1, 256, 1, 1), scaling_stats_permute_dims=(1, 0, 2, 3), scaling_per_output_channel=False),
            QConv2d(256, 512, kernel_size=3, stride=2, padding=1, weight_quant = Int8WeightPerTensorFixedPoint, bias_quant = Int16Bias, input_quant = Int8ActPerTensorFixedPoint, output_quant = Int8ActPerTensorFixedPoint),
            QuantReLU(act_quant=CommonUintActQuant, bit_width=8, per_channel_broadcastable_shape=(1, 512, 1, 1), scaling_stats_permute_dims=(1, 0, 2, 3), scaling_per_output_channel=False, return_quant_tensor=True)
        ),
        nn.Sequential(
            QConv2d(512, 128, kernel_size= 1, weight_quant = Int8WeightPerTensorFixedPoint, bias_quant = Int16Bias, input_quant = Int8ActPerTensorFixedPoint, output_quant = Int8ActPerTensorFixedPoint),
            QuantReLU(act_quant=CommonUintActQuant, bit_width=8, per_channel_broadcastable_shape=(1, 128, 1, 1), scaling_stats_permute_dims=(1, 0, 2, 3), scaling_per_output_channel=False),
            QConv2d(128, 256, kernel_size=3, stride=2, padding=1, weight_quant = Int8WeightPerTensorFixedPoint, bias_quant = Int16Bias, input_quant = Int8ActPerTensorFixedPoint, output_quant = Int8ActPerTensorFixedPoint),
            QuantReLU(act_quant=CommonUintActQuant, bit_width=8, per_channel_broadcastable_shape=(1, 256, 1, 1), scaling_stats_permute_dims=(1, 0, 2, 3), scaling_per_output_channel=False, return_quant_tensor=True)
        ),
        nn.Sequential(
            QConv2d(256, 128, kernel_size= 1, weight_quant = Int8WeightPerTensorFixedPoint, bias_quant = Int16Bias, input_quant = Int8ActPerTensorFixedPoint, output_quant = Int8ActPerTensorFixedPoint),
            QuantReLU(act_quant=CommonUintActQuant, bit_width=8, per_channel_broadcastable_shape=(1, 128, 1, 1), scaling_stats_permute_dims=(1, 0, 2, 3), scaling_per_output_channel=False),
            QConv2d(128, 256, kernel_size=3, stride=2, padding=1, weight_quant = Int8WeightPerTensorFixedPoint, bias_quant = Int16Bias, input_quant = Int8ActPerTensorFixedPoint, output_quant = Int8ActPerTensorFixedPoint),
            QuantReLU(act_quant=CommonUintActQuant, bit_width=8, per_channel_broadcastable_shape=(1, 256, 1, 1), scaling_stats_permute_dims=(1, 0, 2, 3), scaling_per_output_channel=False, return_quant_tensor=True)
        ),
        nn.Sequential(
            QConv2d(256, 128, kernel_size= 1, weight_quant = Int8WeightPerTensorFixedPoint, bias_quant = Int16Bias, input_quant = Int8ActPerTensorFixedPoint, output_quant = Int8ActPerTensorFixedPoint),
            QuantReLU(act_quant=CommonUintActQuant, bit_width=8, per_channel_broadcastable_shape=(1, 128, 1, 1), scaling_stats_permute_dims=(1, 0, 2, 3), scaling_per_output_channel=False),
            QConv2d(128, 256, kernel_size=3, stride=2, padding=1, weight_quant = Int8WeightPerTensorFixedPoint, bias_quant = Int16Bias, input_quant = Int8ActPerTensorFixedPoint, output_quant = Int8ActPerTensorFixedPoint),
            QuantReLU(act_quant=CommonUintActQuant, bit_width=8, per_channel_broadcastable_shape=(1, 256, 1, 1), scaling_stats_permute_dims=(1, 0, 2, 3), scaling_per_output_channel=False, return_quant_tensor=True)
        ),
        nn.Sequential(
            QConv2d(256, 128, kernel_size= 1, weight_quant = Int8WeightPerTensorFixedPoint, bias_quant = Int16Bias, input_quant = Int8ActPerTensorFixedPoint, output_quant = Int8ActPerTensorFixedPoint),
            QuantReLU(act_quant=CommonUintActQuant, bit_width=8, per_channel_broadcastable_shape=(1, 128, 1, 1), scaling_stats_permute_dims=(1, 0, 2, 3), scaling_per_output_channel=False),
            QConv2d(128, 256, kernel_size=4, padding=1, weight_quant = Int8WeightPerTensorFixedPoint, bias_quant = Int16Bias, input_quant = Int8ActPerTensorFixedPoint, output_quant = Int8ActPerTensorFixedPoint),
            QuantReLU(act_quant=CommonUintActQuant, bit_width=8, per_channel_broadcastable_shape=(1, 256, 1, 1), scaling_stats_permute_dims=(1, 0, 2, 3), scaling_per_output_channel=False, return_quant_tensor=True)
        )
    ])
    regression_headers = nn.ModuleList([
        nn.Conv2d(in_channels=512, out_channels=4 * 4, kernel_size=3, padding=1),
        nn.Conv2d(in_channels=1024, out_channels=6 * 4, kernel_size=3, padding=1),
        nn.Conv2d(in_channels=512, out_channels=6 * 4, kernel_size=3, padding=1),
        nn.Conv2d(in_channels=256, out_channels=6 * 4, kernel_size=3, padding=1),
        nn.Conv2d(in_channels=256, out_channels=6 * 4, kernel_size=3, padding=1),
        nn.Conv2d(in_channels=256, out_channels=6 * 4, kernel_size=3,
padding=1),
        nn.Conv2d(in_channels=256, out_channels=6 * 4, kernel_size=1)
, # TODO: change to kernel_size=1, padding=0?
    ])

    classification_headers = nn.ModuleList([
        nn.Conv2d(in_channels=512, out_channels=4 * num_classes,
kernel_size=3, padding=1),
        nn.Conv2d(in_channels=1024, out_channels=6 * num_classes,
kernel_size=3, padding=1),
        nn.Conv2d(in_channels=512, out_channels=6 * num_classes,
kernel_size=3, padding=1),
        nn.Conv2d(in_channels=256, out_channels=6 * num_classes,
kernel_size=3, padding=1),
        nn.Conv2d(in_channels=256, out_channels=6 * num_classes,
kernel_size=3, padding=1),
        nn.Conv2d(in_channels=256, out_channels=6 * num_classes,
kernel_size=3, padding=1),
        nn.Conv2d(in_channels=256, out_channels=6 * num_classes,
kernel_size=1), # TODO: change to kernel_size=1, padding=0?
    ])

    return SSD(num_classes, base_net, source_layer_indexes,
               extras, classification_headers, regression_headers,
center_variance,size_variance,is_test=is_test)
```

Bibliography

[1] URL: https://llis.nasa.gov/lesson/824 (cit. on p. 3).

[2] R.C. Baumann. «Radiation-induced soft errors in advanced semiconductor technologies». In: *IEEE Transactions on Device and Materials Reliability* 5.3 (2005), pp. 305–316. DOI: 10.1109/TDMR.2005.853449 (cit. on p. 3).

[3] R. N. Raphael, L. E. Seixas, Agord M. Pinto, S. A. Bascopé, L. T. Manera, S. Finco, and S. P. Gimenez. «Overview about radiation–matter interaction mechanisms and mitigation techniques». In: *Proceedings of the 3rd Brazilian Technology Symposium* (Aug. 2018), pp. 223–238. DOI: 10.1007/978-3-319-93112-8_23 (cit. on p. 3).

[4] T.S. Nidhin, Anindya Bhattacharyya, R.P. Behera, T. Jayanthi, and K. Velusamy. «Understanding radiation effects in SRAM-based field programmable gate arrays for implementing instrumentation and control systems of nuclear power plants». In: *Nuclear Engineering and Technology* 49.8 (Dec. 2017), pp. 1589–1599. DOI: 10.1016/j.net.2017.09.002 (cit. on p. 3).

[5] Filippo Minnella. «Protection and characterization of an open source soft core against radiation effects». Apr. 2018. URL: http://webthesis.biblio.polito.it/7536/ (cit. on p. 3).

[6] Filippo Minnella, Teodoro Urso, Mihai T. Lazarescu, and Luciano Lavagno. *Design and Optimization of Residual Neural Network Accelerators for Low-Power FPGAs Using High-Level Synthesis.* 2023. arXiv: 2309.15631 [cs.AR] (cit. on p. 7).

[7] *techniques for radiation effects mitigation in ASICs and FPGAs.* ESA Requirements and Standards Division ESTEC, P.O. Box 299, 2200 AG Noordwijk The Netherlands, 2016 (cit. on p. 9).

[8] Krzysztof Marek Sielewicz. «Mitigation Methods Increasing Radiation Hardness of the FPGA-Based Readout of the ALICE Inner Tracking System». Presented 13 Nov 2018. Warsaw U., 2018. URL: https://cds.cern.ch/record/2643800 (cit. on p. 10).

[9] Paulo R. C. Villa, Rodrigo Travessini, Fabian L. Vargas, and Eduardo A. Bezerra. «Processor checkpoint recovery for transient faults in critical applications». In: *2018 IEEE 19th Latin-American Test Symposium (LATS)*. 2018, pp. 1–6. DOI: `10.1109/LATW.2018.8349674` (cit. on p. 13).

[10] Kai Zhao, Sheng Di, Sihuan Li, Xin Liang, Yujia Zhai, Jieyang Chen, Kaiming Ouyang, Franck Cappello, and Zizhong Chen. «Algorithm-Based Fault Tolerance for Convolutional Neural Networks». In: *CoRR* abs/2003.12203 (2020). arXiv: `2003.12203`. URL: `https://arxiv.org/abs/2003.12203` (cit. on p. 14).

[11] Dionysios Filippas, Nikolaos Margomenos, Nikolaos Mitianoudis, Chrysostomos Nicopoulos, and Giorgos Dimitrakopoulos. «Low-Cost Online Convolution Checksum Checker». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 30.2 (2022), pp. 201–212. DOI: `10.1109/TVLSI.2021.3119511` (cit. on p. 14).

[12] M. Lanuzza, P. Zicari, F. Frustaci, S. Perri, and P. Corsonello. «Exploiting self-reconfiguration capability to improve SRAM-based FPGA robustness in space and avionics applications». In: *ACM Transactions on Reconfigurable Technology and Systems* 4.1 (Dec. 2010), pp. 1–22. DOI: `10.1145/1857927.1857935` (cit. on p. 14).

[13] Muhammad Abdullah Hanif and Muhammad Shafique. «Dependable Deep Learning: Towards Cost-Efficient Resilience of Deep Neural Network Accelerators against Soft Errors and Permanent Faults». In: *2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS)*. 2020, pp. 1–4. DOI: `10.1109/IOLTS50870.2020.9159734` (cit. on p. 17).

[14] Ussama Zahid, Giulio Gambardella, Nicholas J. Fraser, Michaela Blott, and Kees Vissers. *FAT: Training Neural Networks for Reliable Inference Under Hardware Faults*. 2020. arXiv: `2011.05873 [cs.LG]` (cit. on p. 17).

[15] R. T. Syed, M. Ulbricht, K. Piotrowski, and M. Krstic. «Fault Resilience Analysis of Quantized Deep Neural Networks». In: *2021 IEEE 32nd International Conference on Microelectronics (MIEL)*. 2021, pp. 275–279. DOI: `10.1109/MIEL52794.2021.9569094` (cit. on p. 18).

[16] Dongyeob Shin, Wonseok Choi, Jongsun Park, and Swaroop Ghosh. «Sensitivity-Based Error Resilient Techniques With Heterogeneous Multiply–Accumulate Unit for Voltage Scalable Deep Neural Network Accelerators». In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.3 (2019), pp. 520–531. DOI: `10.1109/JETCAS.2019.2933862` (cit. on p. 18).

[17] Florian Geissler, Syed Sha Qutub, Sayanta Roychowdhury, Ali Asgari Khoshouyeh, Yang Peng, Akash Dhamasia, Ralf Graefe, Karthik Pattabiraman, and Michael Paulitsch. «Towards a Safety Case for Hardware Fault Tolerance in Convolutional Neural Networks Using Activation Range Supervision». In: *CoRR* abs/2108.07019 (2021). arXiv: `2108.07019`. URL: `https://arxiv.org/abs/2108.07019` (cit. on p. 19).

[18] Zitao Chen, Guanpeng Li, and Karthik Pattabiraman. «Ranger: Boosting Error Resilience of Deep Neural Networks through Range Restriction». In: *CoRR* abs/2003.13874 (2020). arXiv: `2003.13874`. URL: `https://arxiv.org/abs/2003.13874` (cit. on p. 19).

[19] F. Libano, B. Wilson, J. Anderson, M. J. Wirthlin, C. Cazzaniga, C. Frost, and P. Rech. «Selective Hardening for Neural Networks in FPGAs». In: *IEEE Transactions on Nuclear Science* 66.1 (2019), pp. 216–222. DOI: `10.1109/TNS.2018.2884460` (cit. on p. 20).

[20] URL: `https://www.kaggle.com/c/airbus-ship-detection` (cit. on p. 21).

[21] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. «SSD: Single Shot MultiBox Detector». In: *CoRR* abs/1512.02325 (2015). arXiv: `1512.02325`. URL: `http://arxiv.org/abs/1512.02325` (cit. on p. 22).

[22] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. «Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation». In: *CoRR* abs/1801.04381 (2018). arXiv: `1801.04381`. URL: `http://arxiv.org/abs/1801.04381` (cit. on p. 26).

[23] Qfgaohao. *Qfgaohao/pytorch-SSD: Mobilenetv1, mobilenetv2, VGG based SSD/SSD-lite implementation in pytorch 1.0 / pytorch 0.4. out-of-box support for retraining on open images dataset. ONNX and caffe2 support. experiment ideas like coordconv.* URL: `https://github.com/qfgaohao/pytorch-ssd?tab=readme-ov-file` (cit. on pp. 26, 67).

[24] A. Mahmoud, N. Aggarwal, A. Nobbe, J. R. S. Vicarte, S. V. Adve, C. W. Fletcher, I. Frosio, and S. K. S. Hari. «PyTorchFI: A Runtime Perturbation Tool for DNNs». In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2020, pp. 25–31 (cit. on p. 41).