Delft University of Technology

Master Thesis

Compressed convolutional neural networks for sewer inspection

---

Author:
Christoph Brendan Determan

Supervisors:
Dr.ir. J.G. Langeveld (TU Delft)
Dr.ir. S. van Nederveen (TU Delft)
Dr. R. Taormina (TU Delft)

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Construction Management and Engineering at the faculty of Civil Engineering and Geosciences

Delft University of Technology

May 14, 2024

Abstract

Christoph Brendan Determan

Compressed convolutional neural networks for sewer inspection

The inspection of extensive and hard-to-access sewer systems is a challenging and expensive task. As these networks age and need to comply with stricter health and environmental regulations, the demand for effective inspection solutions has increased. The introduction of technologies like CCTV (closed-circuit television) and SSET (sewer scanner and evaluation technology) marked the initial automation steps in sewer inspections. Initially, images from these technologies were manually analyzed for defects, but over time, computer vision techniques have emerged as a highly promising method for automating image processing. However, these advanced computer vision methods are computationally intensive and typically rely on cloud-based architectures, which can be costly and sometimes impractical due to energy or communication limitations. A proposed solution is to shift the computational processes from the cloud to edge computing, which can address issues related to latency and scalability.

The Sewer-ML dataset, sourced from sewer inspection videos by Danish water utilities, comprises 1.3 million images annotated across 18 defect categories. This extensive multi-label dataset serves as a foundation for training and evaluating machine learning models within sewer system management. Model performance is evaluated using the $F2_{CIW}$ (class importance weight) score, which emphasizes recall and defect severity to ensure the accurate detection of critical defects, and the $F1_{Normal}$ score, which measures the model's accuracy in identifying instances without defects, essential for efficient resource management.

The ResNet-101 and TResNet-L models, trained on the Sewer-ML dataset, underwent various compression methods including quantization, layer fusion, and pruning. Quantization was applied only to the ResNet-101, reducing its $F2_{CIW}$ and $F1_{Normal}$ scores by 2.52% and 0.72% respectively, while significantly boosting inference speed by up to 95% on standard platforms and 174.50% on L4 GPUs. Layer fusion was also implemented, further enhancing inference efficiency. Additionally, iterative pruning was performed, showing that while the TResNet-L could maintain performance up to an 80% pruning rate, there was a noticeable initial drop in performance for both models.

The quantized ResNet-101, both the one with and without layer fusion, even improve compared to the standard model in regards to correctly identifying the highest CIW defect class present in pipes deemed defective in the validation dataset. This

model behaviour is positive because the priority of a sewer asset manager is the discovery of the defect that carry the highest risk with them if not treated in time. This improved efficiency in defect recognition helps in optimizing repair schedules and resource allocation, thus reducing operational costs as well.

Acknowledgements

I would to thank my supervisors for the support they have given me during this process. I extend my gratitude to Dr. ir. J.G. Langeveld for his insightful comments on the practical implications of my work, Dr. ir. S. van Nederveen for your guidance at the very beginning of my thesis and throughout the master via several courses and I want to thank my bi-weekly supervisor Dr. R. Taormina. Your enthusiasm for this field of study is contagious and I truly believe you are the ideal person to lead the AI transformation within the Faculty of Civil Engineering and Geosciences. Thank you for welcoming attitude and your invaluable expertise.

A big thank you to all my friends from 'civiel': Abdul, Ali, Brahm, Ilias, Marc, Marvin, Onno, Ruben and Shaniel. Would not have been able to do it without you boys.

And lastly I want to thank my dear family. Thank you for all those years of support, I am more grateful than I can put into words. Roman and Julian, thank you always. Mom and dad, I love you. Thank you so much for everything you have done for me and this one is for you.

Contents

Abstract iii

Acknowledgements v

1 Introduction 1
- 1.1 Sewer inspection . . . 1
- 1.2 Problem statement . . . 1
- 1.3 Research questions . . . 3
- 1.4 Delimitations . . . 3
- 1.5 Needs-analysis . . . 4
- 1.6 Methodology & thesis outline . . . 4

2 Sewer asset management 7
- 2.1 Sewer systems . . . 7
- 2.2 Sewer inspection . . . 8
  - 2.2.1 Current sewer inspection practice . . . 8
  - 2.2.2 Research developments . . . 11

3 Introduction to convolutional neural networks 13
- 3.1 Deep Learning . . . 13
- 3.2 Convolutional Neural Networks . . . 14
  - 3.2.1 Data representation in CNNs . . . 15
  - 3.2.2 Convolutional layer . . . 15
  - 3.2.3 Activation layer . . . 19
  - 3.2.4 Pooling layer . . . 19
  - 3.2.5 Fully connected layer . . . 20

4 Compression techniques 21
- 4.1 Quantization . . . 21
  - 4.1.1 Uniform quantization . . . 22
  - 4.1.2 Non-uniform quantization . . . 22
  - 4.1.3 Mixed precision quantization . . . 22
  - 4.1.4 Static quantization . . . 23
  - 4.1.5 Dynamic quantization . . . 23
  - 4.1.6 Quantization Aware Training (QAT) . . . 23
- 4.2 Pruning . . . 23
  - 4.2.1 Structured pruning . . . 23
  - 4.2.2 Unstructured pruning . . . 24
  - 4.2.3 Pruning strategies . . . 24
- 4.3 Knowledge distillation . . . 25

5 Sewer-ML: data, metrics and benchmarks 27

5.1 Data collection . . . 27
5.2 Performance metrics . . . 27
5.2.1 Precision . . . 29
5.2.2 Recall . . . 29
5.2.3 F1 Score . . . 29
5.2.4 Accuracy . . . 29
5.2.5 F2 score . . . 29
5.3 Applied models and benchmarks . . . 30
5.3.1 Sewer defect classification models . . . 30
5.3.2 Sewer-ML benchmarks . . . 31

6 Methodology **35**
6.1 ResNet-101 model architecture . . . 35
6.2 Static quantization . . . 36
6.3 Layer fusion . . . 36
6.4 Iterative pruning . . . 37
6.5 Iterative pruning in combination with layer fusion and static quantization . . . 37
6.6 Work order . . . 37

7 Results **39**
7.1 ResNet-101 . . . 39
7.1.1 Static quantization . . . 39
7.1.2 Layer fusion . . . 41
7.1.3 Iterative pruning . . . 42
7.1.4 Iterative pruning in combination with layer fusion and static quantization . . . 43
7.2 TResNet-L . . . 46
7.2.1 Iterative pruning . . . 46
7.3 Practical implications . . . 47

8 Discussion **49**
8.1 Compressed models . . . 49
8.2 Future studies . . . 49

9 Conclusion **51**
9.1 Implementation . . . 51
9.2 Performance . . . 51
9.3 Usability . . . 52
9.4 Recommendations . . . 52

A Static quantization: preparation of the ResNet-101 architecture **55**

B Static quantization ResNet-101 **61**

C Layer fusion ResNet-101 **69**

D ResNet-101 iterative pruning **77**

E Performance metrics ResNet-101 models **85**

F Predicted class count **87**

G Performance metrics TResNet-L models **89**

Bibliography **91**

List of Figures

1.1 Pipeline methodology distribution throughout the years. (Haurum and Moeslund, 2020) . . . 4
1.2 Methodology flowchart of this thesis. . . . 5

2.1 Difference in the sewer inspection process between the NEN 3399 and NEN-EN 13508-2 standards. (Stichting RIONED, 2020) . . . 11
2.2 Images from within sewer pipes displaying a diversity in visual characteristics (Haurum and Moeslund, 2020). . . . 12

3.1 Venn diagram displaying ML as a subsets of AI and DL as a subset of ML (Robins, 2023). . . . 13
3.2 Similarities between a biological and an artificial neuron (Han, 2023). . . 14
3.3 A MLP Artifical Neural Network with an input layer, two so-called hidden layers and an output layer (GfG, 2023). . . . 15
3.4 A convolutional operation entails taking the dot product of the kernel with selected input patch (Goodfellow et al., 2016). . . . 16
3.5 Different filters can emphasize different aspects of the input image (Géron, 2017). . . . 17
3.6 A complete convolution (Dumoulin and Visin, 2016). . . . 18
3.7 Three types of activation functions (Han, 2023). . . . 19
3.8 Max and average pooling (Han, 2023). . . . 19
3.9 Fully connected layer (Unzueta, 2022). . . . 20

4.1 Example of the effect quantization has on an image (Weksler, 2021). . . 21
4.2 Uniform (left) vs non-uniform quantization (right). Real values from a continuous range (r) are mapped to discrete, lower-precision values to a quantized domain (Q). In uniform quantization, the spacing between these quantized values is consistent, while in non-uniform quantization, the spacing can differ. This reflects the variable quantization levels. (Gholami et al., 2022). . . . 22
4.3 Unstructured vs structured pruning (Neuralmagic and Neuralmagic, 2023). . . . 24

5.1 Class codes with description and the corresponding class-importance weights (Haurum and Moeslund, 2021). . . . 28
5.2 Distribution of images deemed normal or defective between the training, validation and test dataset splits. (Haurum and Moeslund, 2021). . 28

6.1 Residual block example (He et al., 2015). . . . 35

7.1 Quantized model $F2_{CIW}$ score for each model based on the size of the calibration dataset. . . . 39

7.2 Quantized model $F1_{Normal}$ score for each model based on the size of the calibration dataset. 40
7.3 Layer fusion and quantized model $F2_{CIW}$ score for each model based on the size of the calibration dataset. 41
7.4 Layer fusion and quantized model $F1_{Normal}$ score for each model based on the size of the calibration dataset. 42
7.5 Pruned ResNet-101 $F2_{CIW}$ score for each pruning rate. 42
7.6 Pruned ResNet-101 $F1_{Normal}$ score for each pruning rate. 44
7.7 30% pruned, layer fusion and quantized model $F2_{CIW}$ score for each model based on the size of the calibration dataset. 45
7.8 30% pruned, layer fusion and quantized model $F1_{Normal}$ score for each model based on the size of the calibration dataset. 45
7.9 TResNet-L $F2_{CIW}$ score for each score for each pruning rate. 46
7.10 TResNet-L $F1_{Normal}$ score for each score for each pruning rate. 47

List of Tables

2.1 Codes and descriptions according to NEN-EN 13508-2. (NEN-EN 13508-2, 2021) . . . 9
2.2 Comparison of Inspection Standards. (Stichting RIONED, 2020) 10

5.1 Performance metrics for each model on the validation and test sets (Haurum and Moeslund, 2021). . . . 32

7.1 Inference speed, throughput on CPU and L4 GPU, and size of each model with percentage change in throughput relative to Standard ResNet-101. . . . 41
7.2 Count of images with correctly identified highest CIW defect and recall scores for various model configurations, based on the ground truth total of 61,365 defective images. . . . 47

E.1 Overall $F2_{CIW}$ and $F1_{Normal}$ scores for all variants of the ResNet-101 model. . . . 85
E.2 Class $F2$ scores for all variants of the ResNet-101 model. . . . 85
E.3 Class precision scores for all variants of the ResNet-101 model. . . . 85
E.4 Class recall scores for all variants of the ResNet-101 model. . . . 85

F.2 Number of images with no defects predicted by all variants of the ResNet-101 model. . . . 87
F.1 Model prediction counts and percentage changes relative to the standard ResNet-101 model. . . . 88

G.1 Model prediction counts and percentage changes relative to the standard TResNet-L model. . . . 89
G.2 Number of images with no defects predicted by selected variants of the TResNet-L model. . . . 89
G.3 Per-class $F2$ scores for the TResNet-L model variants. . . . 90
G.4 Per-class precision scores for the TResNet-L model variants. . . . 90
G.5 Per-class recall scores for the TResNet-L model variants. . . . 90

Dedicated to my mother and my father

Chapter 1

Introduction

1.1 Sewer inspection

The significance of sewer systems in the modern world cannot be overstated. It can even be said that sewer systems are foundational to the development of a modern society, since it is a prerequisite for harboring public health, urban functionality and the sustainability of the environment. The role of sewer systems is becoming even more important as cities grow and societies get more interconnected. Therefore, these seemingly invisible networks demand thorough maintenance to ensure present and future functionality.

The combined Dutch municipal sewage expenditure in 2022 was €1.8 billion (*Geld - Riool en raad* 2023). This money, collected by the Dutch municipalities via a sewage tax, is used to maintain and repair all parts of the sewer system in the Netherlands. Proper maintenance of the sewer system demands it to be timed correctly, for delayed maintenance can result in more significant damage within the sewer system. As a result, specific parts of the sewer piping might be damaged too severely to be repaired, necessitating a complete replacement of said part. This, in turn, causes a higher repair cost. It is estimated that it would cost 87 billion euros to completely replace the current sewer system in the Netherlands (Stichting RIONED, n.d.[a]).

In conclusion, proper asset management is crucial to maintaining the uninterrupted operation of the sewer system, while also keeping maintenance costs at a minimum.

1.2 Problem statement

The inspection of a vast and difficult-to-access sewer pipeline system is a challenging and costly endeavor. An effective response to this challenge has become increasingly sought after as many networks approach the end of their designed lifetimes and must comply with stricter health and environmental regulations (Moradi and Zayed, 2017). Initially, the shift from traditional manual inspections to the use of CCTV (closed-circuit television) and SSET (sewer scanner and evaluation technology) marked the first step towards automating this process. Initially, images captured by these technologies were manually checked for defects, but later, the use of computer vision techniques emerged as the most promising method to automate image processing (Haurum and Moeslund, 2020).

Despite these advancements, state-of-the-art computer vision techniques entail high computational costs, typically requiring a cloud-centric architecture for deployment (Zaidi et al., 2022). This not only increases financial costs but also poses feasibility challenges in situations constrained by energy and communication limitations. The solution to this problem has been the migration of computational tasks from the cloud to the edge, addressing issues of latency and scalability effectively (Chen and Ran, 2019).

Edge computing facilitates data processing close to the data source—in this case, the sewer system itself—significantly reducing latency as data no longer needs to be transmitted to a distant cloud server. This approach alleviates scalability problems caused by network congestion as more devices connect and interact. However, deploying computationally intensive algorithms like those used in computer vision on small, edge devices requires innovations such as model compression techniques to decrease both computational complexity and storage demands.

Traditional sewer inspection methods, particularly those involving closed-circuit television (CCTV), are increasingly seen as inadequate due to their time-consuming nature, high costs, and the subjective interpretations required by human inspectors (Xu et al., 2022). These methods also pose safety risks to workers and often lead to inconsistent and unreliable data regarding sewer conditions. In contrast, the introduction of sewer floating capsule robots equipped with advanced computer vision technologies offers a more effective solution (Xu et al., 2022). These robots automate the inspection process, providing rapid and accurate assessments. This shift not only mitigates the limitations associated with manual inspections, but also significantly improves the efficiency and safety of sewer maintenance operations.

Mounce et al. (2021) notes that the emergence of autonomous robotics in sewer networks is revolutionizing how these infrastructures are maintained. Advanced robots, equipped with new sensing approaches like in-pipe robotics, are becoming integral to water companies as they transition to smarter, more proactive practices. This is part of the broader movement towards smart water networks, where robotic autonomous systems (RAS), that are spread out through the sewer system, make continuous assessment of pipe condition and operational performance possible, allowing for a shift from reactive to proactive maintenance strategies (Mounce et al., 2021).

The development of a crawling robot capable of inspecting long distances within narrow sewer pipes is another innovative approach (Tanaka et al., 2014). These robots address the accessibility challenges posed by narrow pipes that traditional methods fail to inspect effectively, thus enhancing safety and efficiency in sewer inspections. Additionally, the deployment of legged robots for the autonomous inspection of concrete deterioration showcases the potential for robotic systems to perform tasks traditionally done by human inspectors, even under challenging conditions (Kolvenbach et al., 2020). Equipped with sensors that allow them to tactically assess structural integrity, these robots can navigate through sewers, feeling the surface roughness to evaluate concrete conditions where visual inspection fails. Using advanced sensing technologies like sonar and LIDAR allows these robotic systems to gather detailed data, which is important for understanding the condition of sewer infrastructures. This ongoing data collection is essential for scheduling maintenance, which helps prolong the life of these important infrastructure elements.

In conclusion, as sewer systems around the world age and the demands for efficient, cost-effective maintenance increase, the role of technology, particularly the use of edge computing and robotics in sewer inspection, becomes increasingly crucial. These technologies not only promise to reduce costs and enhance efficiency, but also improve the safety conditions under which these inspections are carried out. Therefore, applying existing methods to compress neural network models specifically for sewer defect inspection on edge devices is crucial for enhancing the inspection processes and represents a significant advancement in the field.

1.3 Research questions

The main research question that arises from the problem statement is as follows:

"Can model compression methods be exploited to facilitate computer vision tasks at the edge for sewer system defect detection?"

The first two sub-questions are formulated to answer the main research question. The third sub-question aims to identify how usable the model is in the context of Dutch sewer asset management and whether specific improvements could perhaps be made. The three sub-questions are:

1. How are the compression techniques implemented into the original models?
2. How do the compressed models perform compared to the uncompressed models?
3. How usable are the compressed models for sewer asset management?

1.4 Delimitations

In order to adapt computer vision algorithms to such an extent that it is possible to be run on an edge device and uphold significant accuracy in regards to effectively identifying sewer system damages, it is necessary to establish several key aspects that contribute to realizing this practical application. These aspects are: computer vision algorithms, model compression techniques and defining the hardware limitations of the hypothetical edge device.

A plethora of algorithms exist that can be used for classifying purposes in the context of sewer inspection. When looking at Figure 1.1, an increase in popularity of the use of deep learning algorithms can be observed. Convolutional neural networks (CNNs) hold the biggest share of the applied deep learning methods (Haurum and Moeslund, 2020). Therefore, the focus of this thesis will be on the use of the CNN architecture as a means to deploy computer vision for sewer system inspection.

When looking at model compression techniques, this thesis will be limited to the research of effective application of the following three techniques: pruning, knowledge distillation and quantization. These techniques have proven to be popular model compression choices in recent years (Li et al., 2023) (Choudhary et al., 2020).

The dataset with which the models are trained, is the Sewer-ML dataset created by Haurum and Moeslund (2021). This extensive, multi-label dataset has 1.3 million images, which have been labelled by professional sewer inspectors over a period of nine years (Haurum and Moeslund, 2021). A more in depth analysis of this dataset will follow in a later Chapter.

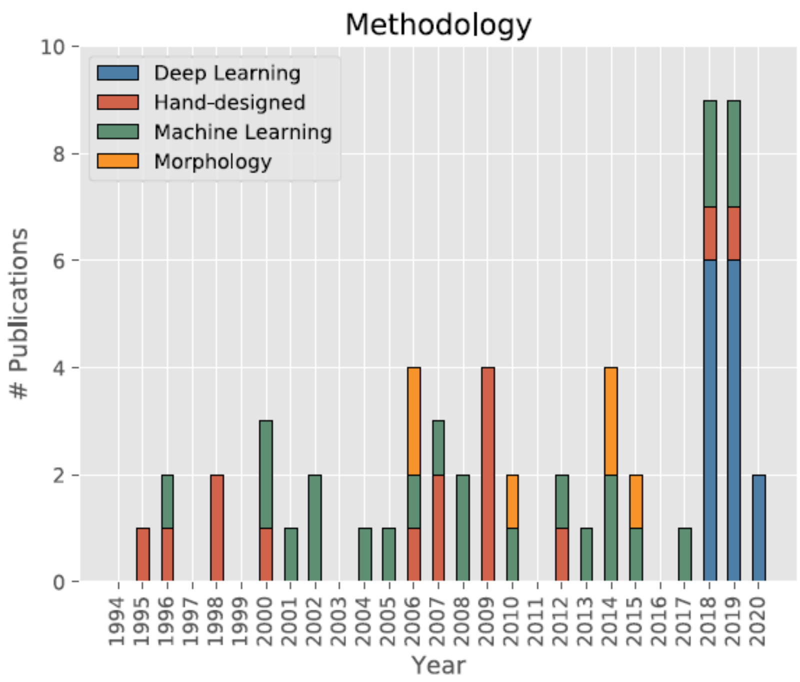

FIGURE 1.1: Pipeline methodology distribution throughout the years. (Haurum and Moeslund, 2020)

1.5 Needs-analysis

The needs-analysis for this thesis involves understanding the current challenges in sewer system defect detection, how the applied computer vision models work and how to implement compression techniques. A standard computer vision algorithm uses a neural network that is trained with labelled data (e.g. an image of a sewer pipe with a crack in the wall) to classify an unlabelled image, therefore a high quality labelled dataset to possibly train or correct the models is also needed. Another aspect of the needs-analysis that must be looked at are the user demands. In practice this means that the performance of the conceived computer vision algorithm must be compared to that of current sewer inspection performance numbers. Caradot et al. (2018) found that the probability of correctly assessing a sewer pipe in poor condition is approximately 80%. In this study, assessments by sewer system professionals were analysed. This implies that the 80% is representative of human inspection prowess. A part of the needs-analysis for the yet to be developed algorithm is, therefore, to be able to correctly assess a sewer pipe in poor condition in at least 80% of the cases.

1.6 Methodology & thesis outline

Chapter two will first start with an in-depth overview of sewer system asset management practices and the type of damages that occur. A comprehensive review will provide insights into the challenges faced by asset managers and the importance of timely maintenance for the longevity and functionality of sewer systems. Chapter three will explain the technical aspects of the CNN and the rationale behind using CNNs for image-based detection. Chapter four the technical aspects of the three model compression methods that will be explored within this thesis.

In Chapter five the Sewer-ML dataset will be introduced. The architectures of the custom sewer models that were trained on this dataset will be explained and the benchmark performances will be presented. In Chapter 6 the applied model compression techniques will be discussed, followed by the yielded results. The two remaining Chapters are the discussion and conclusion. Figure 1.2 displays the flowchart for the conceptual framework of this thesis.

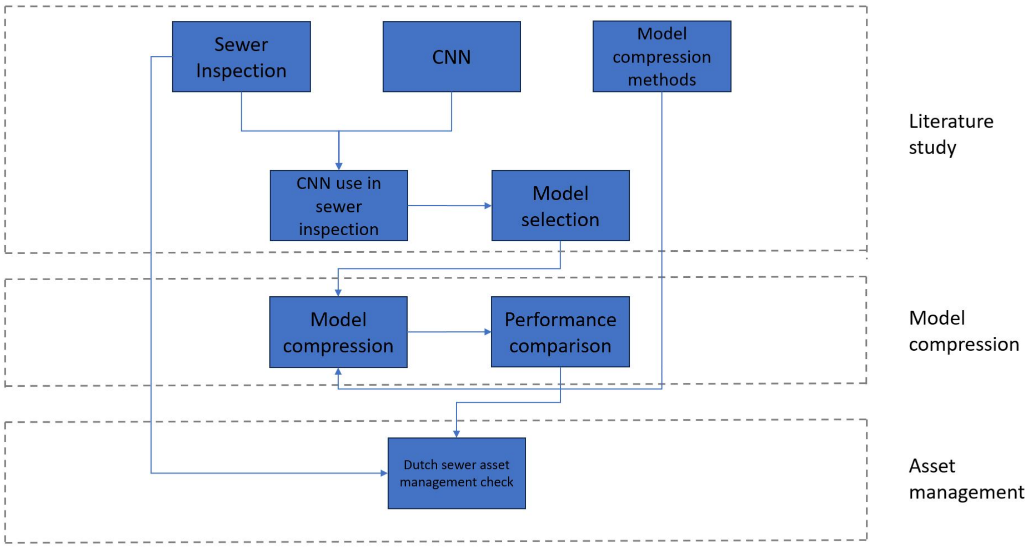

FIGURE 1.2: Methodology flowchart of this thesis.

Chapter 2

Sewer asset management

2.1 Sewer systems

Sewer systems are critical parts of urban water infrastructure, because these systems are responsible for the collection and transport of wastewater and stormwater from residential, commercial, and industrial sources to treatment facilities (Hahn et al., 2002). This water is transported to water treatment facilities, which are designed to remove pollutants and contaminants from the wastewater before safely discharging it into natural water bodies. Therefore, sewer systems are essential for upholding proper sanitation, mitigating the transmission of waterborne diseases and ensuring the responsible handling of wastewater in urban areas.

Because of the critical importance of sewer systems, significant effort is demanded to maintain these systems to guarantee functionality. This results in sewer systems being one of the most expensive infrastructures to maintain (Wirahadikusumah et al., 2001). In Europe, the sewer assets have a estimated collective value of €2 trillion. If a replacement rate of once every 100 years is considered, the annual repair costs will be a grand total of €20 billion (Langeveld and Clemens, 2015). Another estimate states that 50% of the sewer system construction budget is used for repair works (Du et al., 2019).

As of January 1, 2012, USEPA (2016) stated that the total capital requirements for wastewater and stormwater treatment and collection across the United States of America, for projects that will be completed between 2012 and 2017, will amount to $271 billion. This encompasses the following expenses

- Capital needs for publicly owned wastewater pipes and treatment facilities ($197.8 billion).
- Correction of combined sewer overflow issues ($48.0 billion).
- Stormwater management ($19.2 billion).
- The treatment and distribution of recycled water ($6.1 billion).

In recent times, increased urbanization and excessive sewer system connections surpassing their original design capacity to the main sewer pipeline have caused complications in regards to sewer system management. Deprecated wastewater collection systems give rise to various problems, including structural collapse, corroded concrete, fractured tiles, and blockages. The cost of solving these issues can be high (WWAP, 2017).

Therefore, as these underground assets continue to age, concerns will arise regarding their sustained performance and the potential risks of future failures. To guarantee the functionality of this aging infrastructure, it is imperative that strategies are developed that concentrate maintenance efforts on the network components where they can yield the most significant impact. On top of that, growing user and political demands, coupled with stricter environmental regulations, are contributing to the necessity for a more sustainable and holistic management approach to the maintenance of these systems (Fenner, 2000).

In the Netherlands, extensive sewer system development took place between 1950 and 1970 (van Riel, 2016). The average lifetime of sewer pipes is estimated to be between 50 and 90 years, which means that many of the sewer pipes in the Netherlands are at or approaching the end of their respective lifespans. Therefore, both the money spend on repairment and replacement, and the need for effective sewer asset management increases (van Riel et al., 2012).

Dutch municipalities have the following legally mandated responsibilities with regard to water management (Stichting RIONED, n.d.[b]):

- Transportation of wastewater from buildings to purification plants.
- Collecting rainwater and directing it into the ground or surface water, but only if the land or building owner is unable to do so independently. Currently, many municipalities still manage the runoff from gutters and gardens. However, due to climate change, cost considerations and urban and rural growth, this may not always be feasible.
- Implementing measures in response to recurring problems caused by either excessively high or low groundwater levels.

The municipality funds all three water management related tasks through the sewage tax. This tax is levied on all residents and businesses within the municipality.

However, the importance of sewer asset management extends beyond just protecting functionality. It also includes a comprehensive strategy for monitoring and optimizing sewer assets to maximize their lifespan and minimize both operational costs and the environmental impact. In this era of aging infrastructure and increased urbanization, an effective sewer asset management program is essential to safeguard public health, protect natural resources and, by doing so, safeguard the livability of cities and towns.

2.2 Sewer inspection

In order to improve sewer asset management, it is important to firstly define the current reality of sewer inspection, and secondly the direction that improvement developments are taking.

2.2.1 Current sewer inspection practice

In spite of the ongoing shift towards adopting a proactive sewer asset management approach, managers are currently making decisions with inadequate justifications. The methods being employed predominantly rely on intuition (van Riel et al., 2012) and entail high uncertainty (Elachachi et al., 2006). This is the case for both the strategies and the data that are used.

Caradot et al. (2018) researched the likelihood of either underestimating, overestimating, or accurately estimating the true condition of a pipe through visual inspection. This approach relies on the examination of two separate inspections of the same sewer pipes and has undergone testing using a comprehensive data set from Braunschweig, Germany. The structural condition of the examined pipes has been assessed by using an altered version of the French classification methodology RERAU. This methodology assigns a grade ranging from 1 to 4 to sewer pipes, with 4 indicating the poorest condition. It was found that the probability of correctly assessing a sewer pipe in poor condition 4 is nearly 80%, resulting in a corresponding probability of approximately 20% for overestimating the pipe's condition. Generally, the probability of overestimating the condition of a pipe (false negative, FN) tends to be higher than underestimating its condition (false positive, FP). Specifically, for pipes in poor condition, the probability of a false negative is 20%, while for pipes in good condition, the probability of a false positive is 15%.

Dirksen et al. (2013) found that the likelihood of an inspector failing to detect the existence of a defect (FN) is considerably higher than the likelihood of reporting a defect that is not actually present (FP). The occurrence of a false positive is within the range of a few percent, while the probability of a false negative is approximately 25%. Furthermore, upon analyzing sewer inspector examination data using the EN 13508-2 standard, it was revealed that the probability of an inaccurate observation (in terms of defect recognition and/or description) for all defects exceeded 50%.

The EN 13508-2 standard (*Investigation and assessment of drain and sewer systems outside buildings - Part 2: Visual inspection coding system*) is the European standard that states how the visual inspection of sewer pipes should be performed. A coding system is deployed to denote all the types of sewer pipe defects , which are shown in Table 2.1. The EN 13508-2 standard that is adopted in the Netherlands and published by the Dutch Standardization Institute NEN, is called the NEN-EN 13508-2. The only difference between the old NEN 3399 and the current NEN-EN 13508-2 is the addition of the BBH class of vermin.

Code	**Description**	**Code**	**Description**
BAA	Deformation	BBA	Roots
BAB	Cracks	BBB	Attached deposit
BAC	Fracture/collapse	BBC	Settled deposit
BAD	Defective brickwork or masonry	BBD	Ingress of soil
BAE	Missing mortar	BBE	Other obstacle
BAF	Surface damage	BBF	Infiltration
BAG	Intruding inlet	BBG	Exfiltration
BAH	Defective connection	BBH	Vermin
BAI	Intruding sealing material		
BAJ	Displaced joint		
BAK	Defective lining		
BAL	Defective repair		
BAM	Weld error		
BAN	Porous pipe		
BAO	Soil visible due to defect		
BAP	Void visible due to defect		

TABLE 2.1: Codes and descriptions according to NEN-EN 13508-2. (NEN-EN 13508-2, 2021)

Currently, sewer inspection is performed via the following steps:

1. On-site collection of CCTV images by a trained inspector. Mostly done by navigating a remotely controlled vehicle equipped with a movable camera through the sewer pipes.
2. The inspector thoroughly registers his observations and provides this information to the responsible administrator of the sewer system.
3. The administrator provides a detailed defect description, involving characterization, quantification of its magnitude and identification of its location.

In recent times the norm regulated process of inspecting a sewer system has undergone substantial changes. Even when the NEN 3399 was still the used standard, changes were applied by issuing an updated version. The NEN 3399:2004 was revoked and the NEN 3399:2015 was issued. This updated NEN 3399 standard was a simplified classification methodology, focusing on whether certain aspects were observed rather than noting their extent. The intention was to make the inspection process easier for inspectors, reduce the likelihood of errors and cut costs. Unfortunately, the simplification did not yield positive results. Municipalities requested contractors to inspect and record data according to NEN 3399:2015. In practice, inspectors documented only the classes, lacking the detailed observations in line with the European standard. The simplified classification system provided inadequate information for effective management, leading most sewerage managers to revert to the outdated 2004 standard (Stichting RIONED, 2020).

Alarmed by these challenges, a group of municipalities took action. In 2017, the Waste Water Engineering standards committee, RIONED Foundation, and stakeholders extensively assessed and discussed the situation. Recognizing the need for revision, a decision was made to adopt NEN-EN 13508-2 from 2011 for sewer inspections in the Netherlands, effective from 2020. Consequently, NEN 3399 was revoked and is no longer in use. Table 2.2 shows the difference between the different standards throughout recent times and Figure 2.1 shows the change in process order and responsibility for both the inspector and administrator.

	Up to 2014	**2015 - 2019**	**From 2020**
Standard	NEN 3399:2004	NEN 3399:2015	NEN-EN 13508-2+A1:2011
What is Recorded?	Global classes	Limited number of global classes	Measured or estimated values (details)
Standard Range	Pipes only	Pipes and manholes	Pipes and manholes
Exchange Format	SUF-RIB 2.1	RibX	RibX 1.3.2
Who Inspects?	Inspector	Inspector	Inspector
Who Classifies?	Inspector	Inspector	Administrator (with the help of software)
Contract Formation	Unambiguous	Disorganized	Unambiguous

TABLE 2.2: Comparison of Inspection Standards. (Stichting RIONED, 2020)

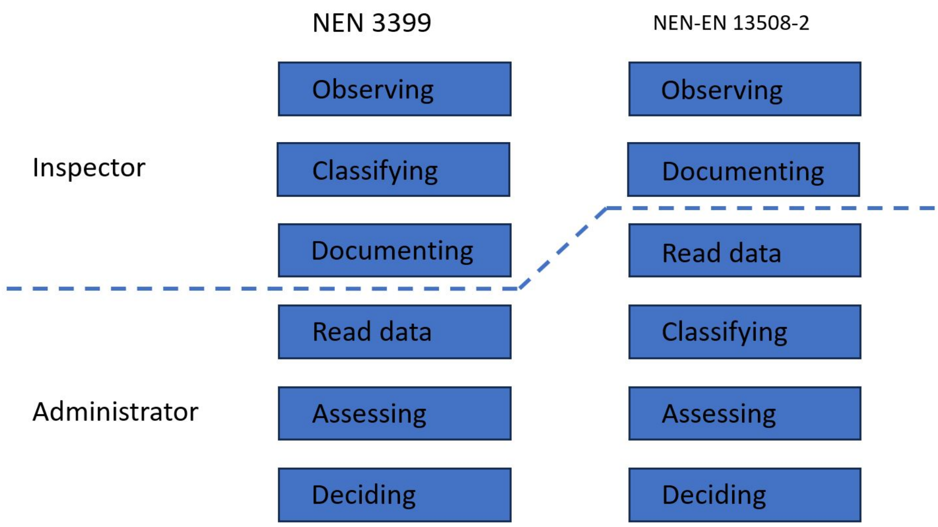

FIGURE 2.1: Difference in the sewer inspection process between the NEN 3399 and NEN-EN 13508-2 standards. (Stichting RIONED, 2020)

The task of inspecting a sewer can be demanding since inspectors are required to observe a video feed for an extended period of time. Inspectors are prone to inaccuracies under such conditions. Moreover, the diversity in visual appearance within sewer pipes adds an additional layer of complexity to the task (Haurum and Moeslund, 2021). An example of this can be seen in Figure 2.2. Because of these challenges, over the last three decades, industry and academia have extensively investigated the field of automated sewer inspection, involving the development of diverse robot platforms and specialized algorithms (Haurum and Moeslund, 2020).

2.2.2 Research developments

Recent progress predominantly revolves around the use of deep learning models to create sewer inspection systems, with a focus on capitalizing on the potential of data-centric feature representation (Zhao et al., 2022). Yin et al. (2021) performed a review of recent literature and concluded that it is evident that defect detection can be automated through various methods, with deep learning techniques being extensively explored for this purpose. Figure 1.1 shows that in recent years deep learning algorithms have become the most popular method for classification tasks within in the context of sewer inspection and CNNs are the lion's share of applied deep learning methods (Haurum and Moeslund, 2020). This phenomenon can be explained by the fact that CNNs have shown significant potential in the tasks of image classification and object detection (Li et al., 2019). However, as discussed in Chapter one, running a big CNN in a cloud-centric architecture is a computational expensive endeavour. Model compression is proposed to make a computer vision algorithm that is suitable for edge devices.

In the next two Chapters CNNs and the three model compression techniques (quantization, pruning and knowledge distillation) will be explained.

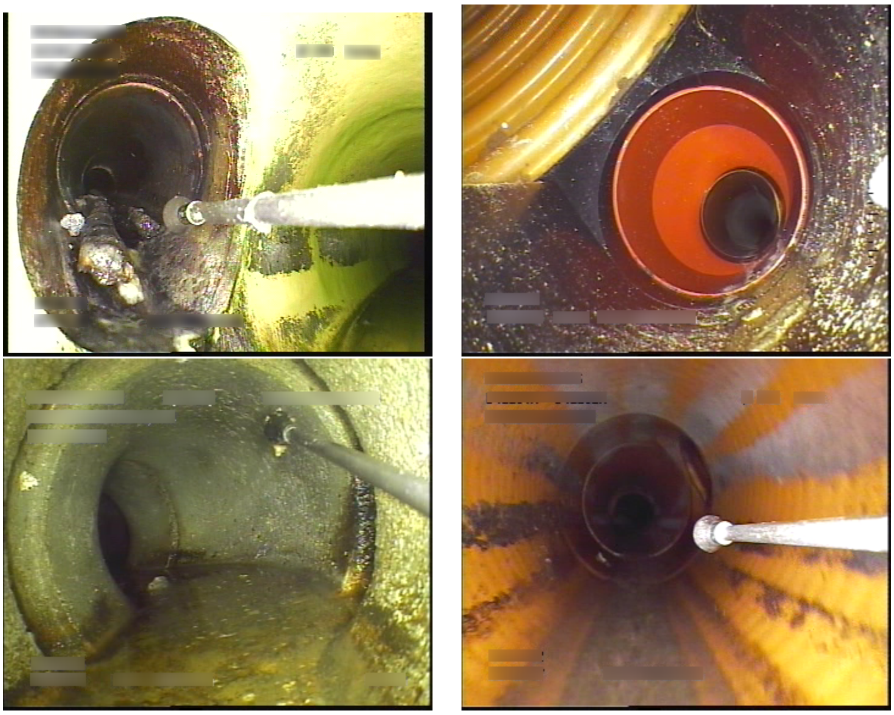

FIGURE 2.2: Images from within sewer pipes displaying a diversity in visual characteristics (Haurum and Moeslund, 2020).

Chapter 3

Introduction to convolutional neural networks

This Chapter will first give a short introduction of deep learning, which is followed up by a more indepth introduction of CNNs.

3.1 Deep Learning

In recent years, the field of Artificial Intelligence (AI) has seen significant advancements, largely driven by developments in Machine Learning (ML) and Deep Learning (DL) techniques. AI, the broader concept, aims to create systems capable of performing tasks that would typically require human intelligence. Within this field, Machine Learning is a subset that focuses on enabling machines to learn from data, make predictions, and improve over time without being explicitly programmed for each task (Jordan and Mitchell, 2015).

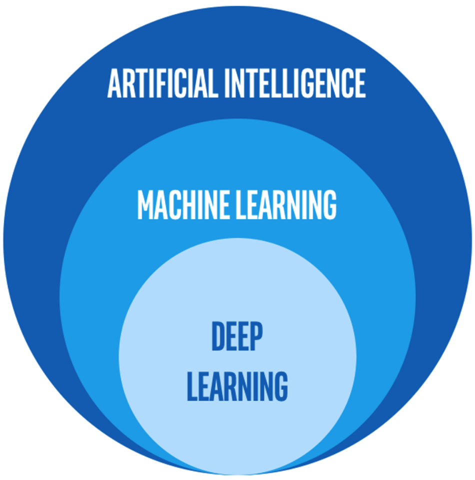

FIGURE 3.1: Venn diagram displaying ML as a subsets of AI and DL as a subset of ML (Robins, 2023).

Deep Learning, a specialized subset of Machine Learning, employs Artificial Neural Networks (ANNs) with multiple layers to model complex patterns in large datasets. These ANNs are inspired by the biological neural networks in the human brain, be it in a simplified manner. Figure 3.2 shows the similarities between a biological neuron and an artificial one. In the case of an artifical neuron, synapses are more commonly know as weights or parameters ANNs consist of interconnected nodes or neurons arranged in layers. The fundamental building block of an ANN is the perceptron, which corresponds with a single-layer neural network. Stacking multiple perceptrons creates a Multilayer Perceptron (MLP), which includes an input layer to receive data, several hidden layers for processing, and an output layer to deliver results (Géron, 2017). An example of a standard MLP Artifical Neural Network can be seen in Figure 3.3.

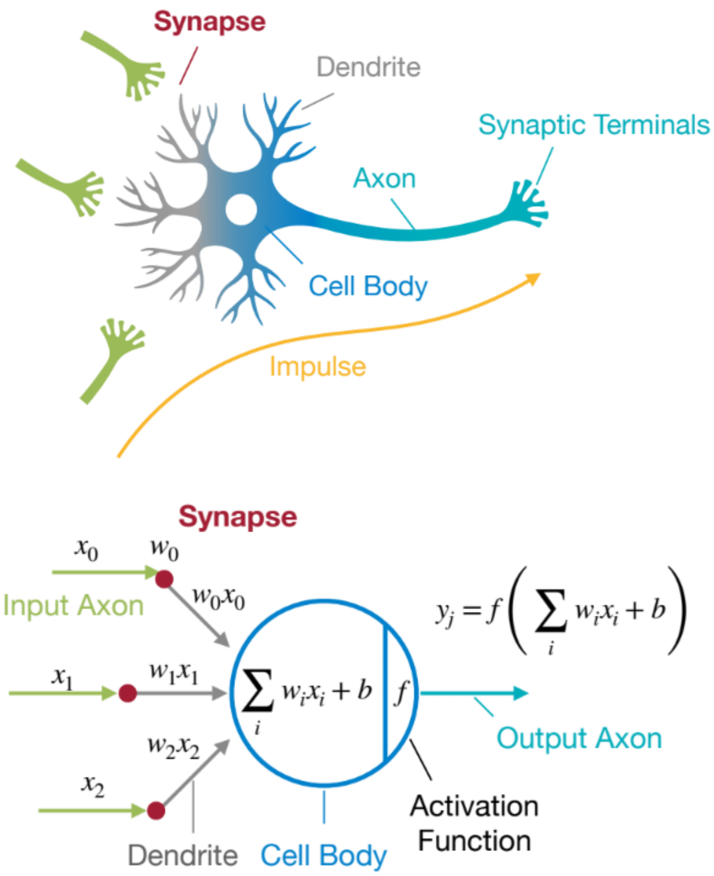

FIGURE 3.2: Similarities between a biological and an artificial neuron (Han, 2023).

This architecture allows DL models to deconstruct and understand the input data progressively through each layer, enabling the model to be useful for complex tasks such as image recognition or natural language processing (Jaiswal, 2024). A key aspect of DL is its ability to automatically discover important characteristics within the data, without the need for human intervention to pick out these features. This ability to learn from data by itself has significantly increased DL's use across many fields (Gillis et al., 2023).

3.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) (LeCun et al., 1989) are a specialized kind of ANN designed primarily for processing data that has a grid-like structure, such as images. A standard fully connected ANN struggles with processing entire images

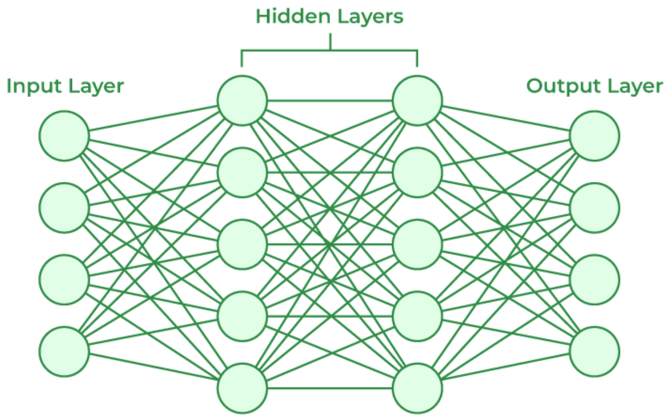

FIGURE 3.3: A MLP Artifical Neural Network with an input layer, two so-called hidden layers and an output layer (GfG, 2023).

effectively because it treats each pixel separately, without recognizing the patterns and structures formed by groups of pixels together. This makes it less efficient for tasks involving complex visual data. This means that for an image with thousands of pixels, a fully connected network would require millions of weights, leading to a massive number of parameters that make the network prone to overfitting and computationally expensive (Mishra, 2021). A CNN typically consist of several different types of layers, those layers will be explained in the following paragraphs.

3.2.1 Data representation in CNNs

In CNNs, data is represented and manipulated as tensors, which are multi-dimensional arrays. A color image is typically represented as a 3D tensor, with dimensions corresponding to the image's height, width, and color channels (such as RGB). Beyond just holding image data, tensors are used throughout CNNs to store weights, biases, and the outputs from various layers. This data format allows CNNs to efficiently process and analyze complex visual information by organizing it in a structured manner.

3.2.2 Convolutional layer

The convolutional layer applies a mathematical operation called convolution to the input image. This operation involves sliding a learnable filter, or kernel (a matrix containing weights), across the image and computing the dot product of the filter with the local patches of the input. Each dot product provides a value in a new output matrix, known as a feature map, which represents a specific feature detected in the input, see figure 3.4. This process allows the layer to capture patterns such as edges, textures, or more complex shapes in the image. Different filters can emphasize different aspects of the input image (Géron, 2017). An example is shown in figure 3.5

In addition to the convolution operation, each convolutional filter has a bias term (Turing, 2022). After the filter is applied to the input through the convolution operation, the bias is added to the result before passing it through a non-linear activation function. The bias term allows the activation function to be shifted to the left or right, which can help the network better fit the data. This is crucial for the learning process because it provides the network with an additional degree of freedom, by making it possible that even if the weighted sum of the inputs to a neuron is zero, the neuron can still be activated if the bias allows it. This makes the model more adaptable and capable of achieving higher performance on a variety of tasks. In Figure 3.2 the bias is denoted as b.

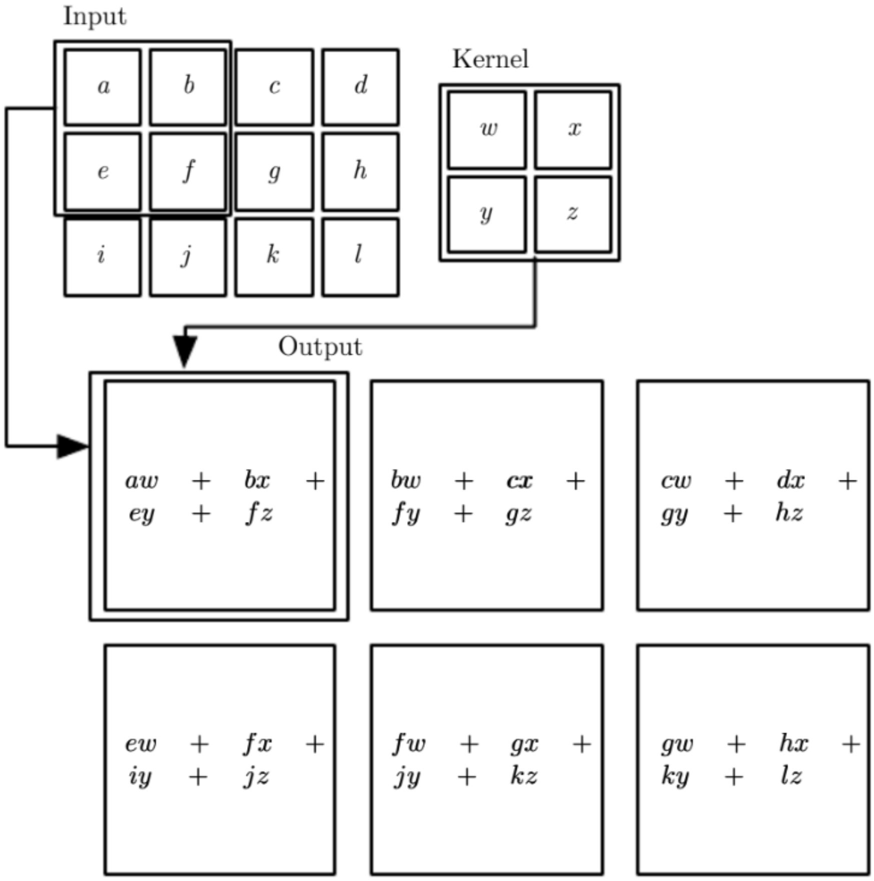

FIGURE 3.4: A convolutional operation entails taking the dot product of the kernel with selected input patch (Goodfellow et al., 2016).

In CNNs, stride and zero-padding are two important concepts that significantly affect how the convolutional layers operate on the input image (Géron, 2017). Stride refers to the number of pixels by which the filter moves across the input image. A stride of 1 means the filter moves one pixel at a time, scanning the entire image closely. Increasing the stride reduces the dimensions of the output feature map because the filter skips over pixels and covers the image more quickly. This can be useful to reduce the computational load and control the level of detail captured in the feature maps. Zero-padding involves adding layers of zeros around the border of the input image. This serves several purposes:

- It allows the use of a convolutional filter near the edges of the image, ensuring that every input pixel can be centered by the filter, which is especially important for capturing information at the edges.

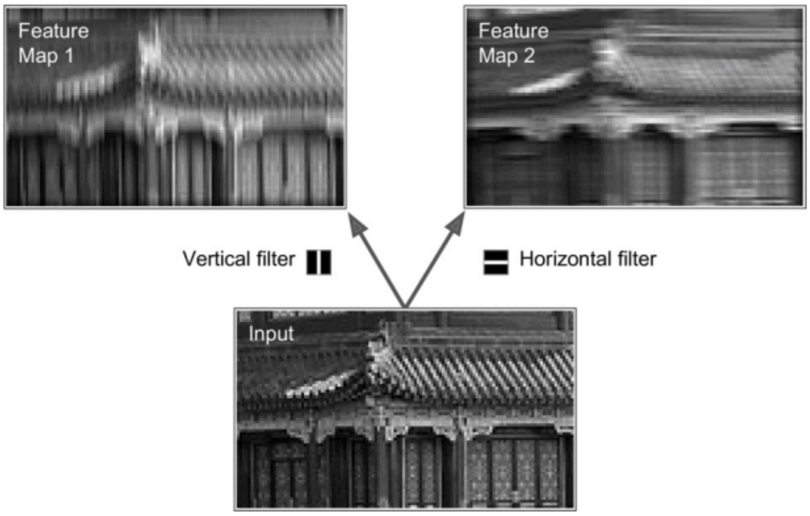

FIGURE 3.5: Different filters can emphasize different aspects of the input image (Géron, 2017).

- It helps control the dimensions of the output feature maps. Without zero-padding, the size of the feature maps decreases with each convolutional layer, which can be undesirable for deep networks. By applying zero-padding, you can maintain the spatial dimensions of the input through the layers, allowing for deeper networks.

Figure 3.6 shows the full process of a convolution. A 3x3 kernel with a stride of 1 is applied on a 5x5 input map with zero-padding. The output is a 3x3 feauture map.

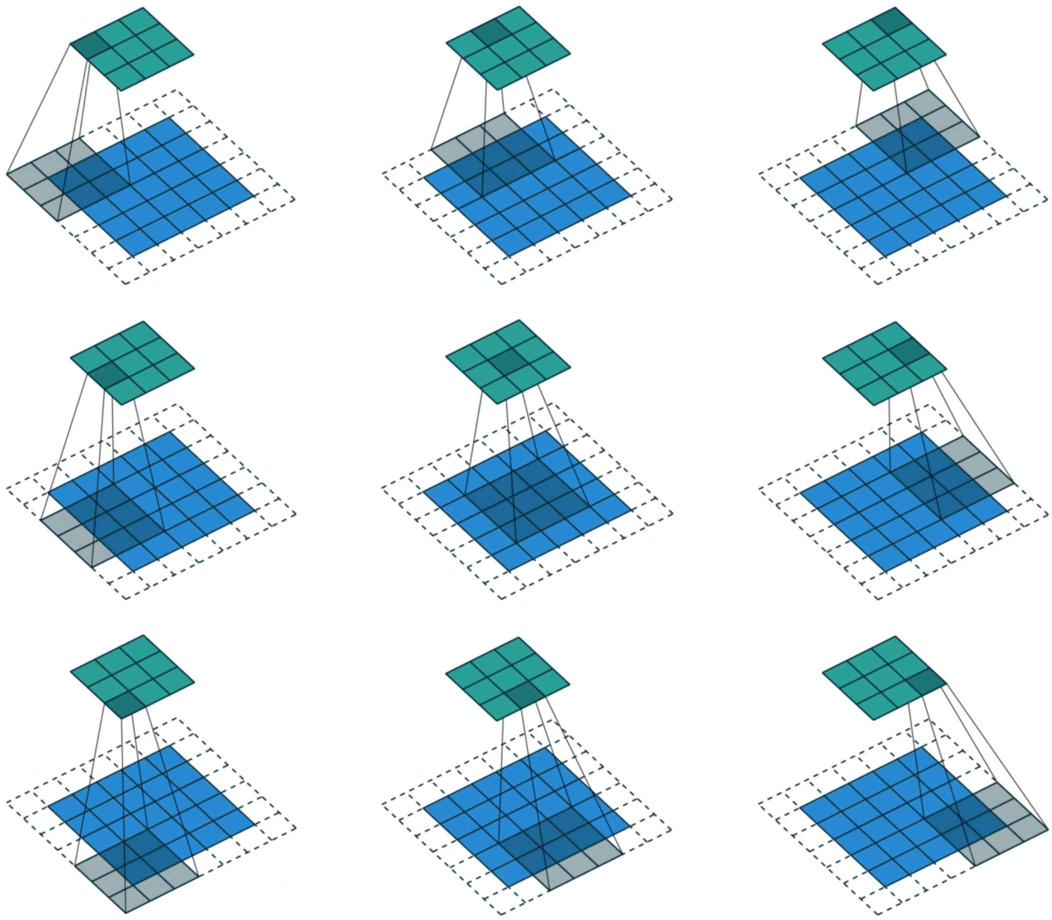

FIGURE 3.6: A complete convolution (Dumoulin and Visin, 2016).

3.2.3 Activation layer

After the convolution operation, the feature map goes through an activation function, typically the Rectified Linear Unit (ReLU) (Szeliski, 2011). This layer introduces non-linearity into the model, allowing it to learn more complex patterns. The ReLU function works by taking each value produced by the network and turning any negative number into zero. Positive values are left unchanged. This process helps the network focus on the most important features in the image and improves its ability to learn and make decisions based on those features. There are many other activation functions that also see common use, two of those are displayed in Figure 3.7.

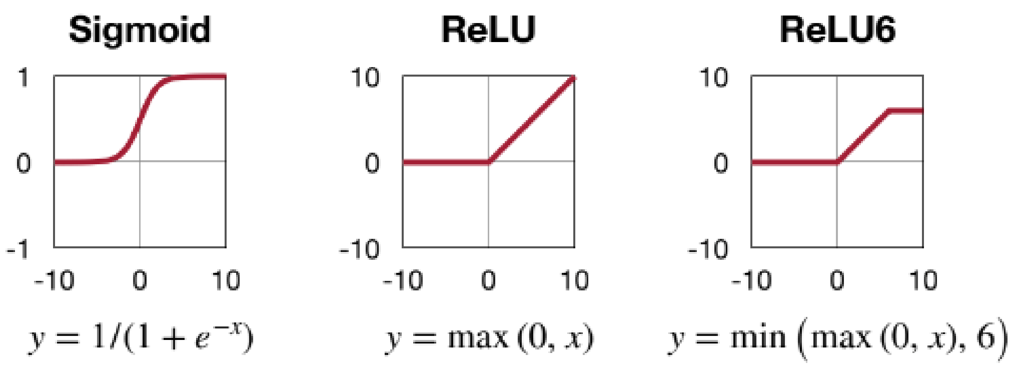

FIGURE 3.7: Three types of activation functions (Han, 2023).

3.2.4 Pooling layer

The pooling layer reduces the dimensionality of each feature map while retaining the most essential information. Max pooling is a technique that breaks down the input image into several small blocks without any overlap between them. For each of these blocks, it takes the highest value and uses that as a representation for the entire block. The process is the same for average pooling, except that the average value is taken instead of the maximum value (Mishra, 2021). This pooling process helps the network to recognize important features in the image, regardless of their size or how they're positioned. It also simplifies the network's structure, making it faster and more efficient in processing images.

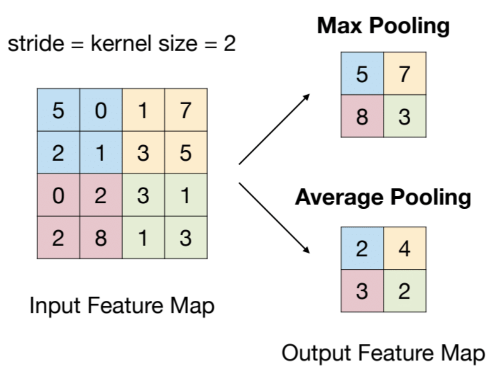

FIGURE 3.8: Max and average pooling (Han, 2023).

3.2.5 Fully connected layer

Towards the end of the network, fully connected layers are used, where every input is connected to every output by a learnable weight (Géron, 2017). These layers combine features learned by the network over the entire image to identify specific patterns. Typically, the last fully connected layer, in combination with a softmax activation function, is used to assign probabilities to different classes based on the features detected by the network.

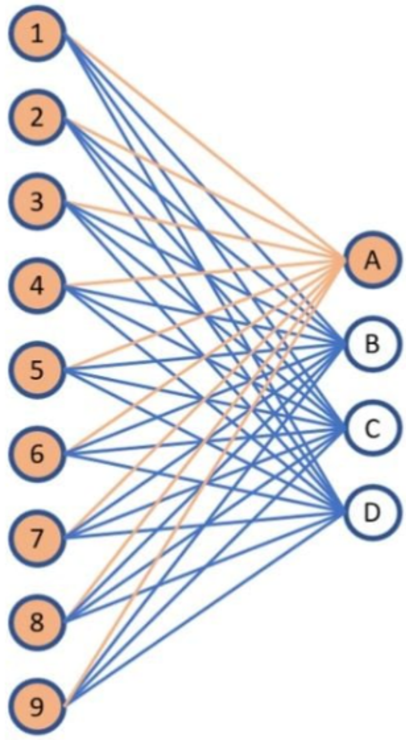

FIGURE 3.9: Fully connected layer (Unzueta, 2022).

Chapter 4

Compression techniques

In this Chapter the three main compression techniques are explained, namely: quantization, pruning and knowledge distilation.

4.1 Quantization

Quantization compresses a neural network by decreasing the numerical precision of its weights and activations (Li et al., 2023). This process aims to maintain the network's predictive performance while significantly reducing its computational and storage demands. When applying quantization, continuous or high-precision numerical values are transformed into discrete or lower-precision values. In the context of deep neural networks this usually means converting 32-bit floating-point numbers (which are standard in training neural networks) into 8-bit integers or other lower bit-widths for both the weights and activations of a network. The input (in the context of CNNs: images) of a neural network is also quantized. The example of a quantized image given in Figure 4.1 effectively illustrates the essence of the quantization process: the data is simplified, yet the crucial details are preserved.

FIGURE 4.1: Example of the effect quantization has on an image (Weksler, 2021).

The quantization process can be split up in two parts: the first being based on the distribution of quantization levels (uniform, non-uniform, and mixed precision quantization) and the second one based on the timing and methodology of the quantization process (static quantization, dynamic quantization, and quantization-aware training). Each category has its own set of methodologies, advantages, and challenges, which we will be explained in the following sections.

4.1.1 Uniform quantization

This technique applies the same level of precision reduction across all numerical values within a neural network (Gholami et al., 2022). By uniformly adjusting the bit-width of the numbers representing weights and activations, uniform quantization simplifies the network's computational requirements. Although this method is widely compatible with various hardware platforms, it might not fully account for the detailed distributions of data within the model, which could affect performance.

4.1.2 Non-uniform quantization

Non-uniform quantization customizes the precision reduction to match the specific distribution of the model's values. It employs a more detailed quantization scale where data values are more concentrated and a broader scale where values are spread out. This customized approach is relatively better at preserving a models performance. However, the complexity of this method may affect its compatibility with certain hardware platforms and reduce efficiency (June, 2023).

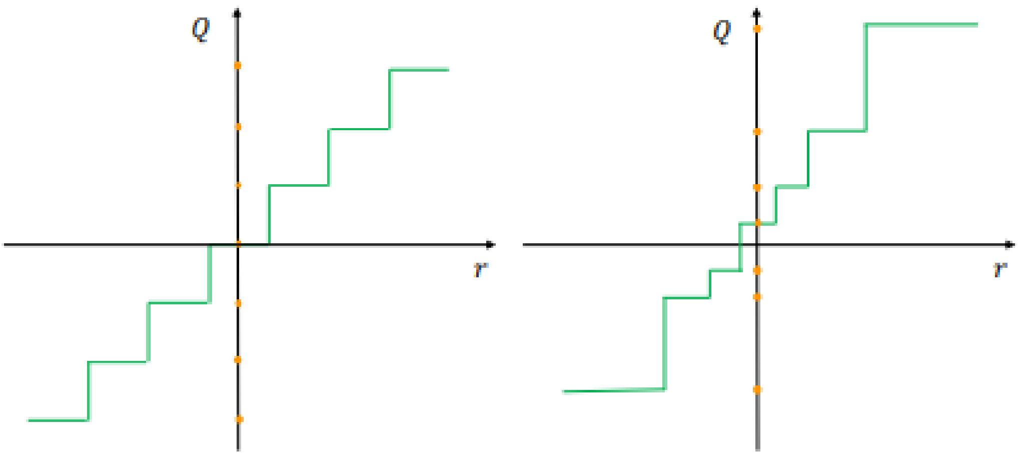

FIGURE 4.2: Uniform (left) vs non-uniform quantization (right). Real values from a continuous range (r) are mapped to discrete, lower-precision values to a quantized domain (Q). In uniform quantization, the spacing between these quantized values is consistent, while in non-uniform quantization, the spacing can differ. This reflects the variable quantization levels. (Gholami et al., 2022).

4.1.3 Mixed precision quantization

Instead of uniformly quantizing every component of the network to the same bit-width, mixed-precision quantization strategically assigns higher precision (more bits) to parts of the model that are crucial for maintaining model performance, and lower precision (fewer bits) to less critical areas. This approach helps in balancing the trade-off between model size, computational efficiency, and performance (Gholami et al., 2022).

4.1.4 Static quantization

Performed as a post-training step, static quantization involves a preliminary calibration phase where a representative dataset is used to analyze the model's behavior and establish optimal quantization parameters, such as scale and zero-point (Gholami et al., 2022). These parameters are used to map the floating point to its quantized value as an integer. Formula 4.1 shows how the quantized value X_q is derived from the floating point 32 datapoint X_{fp32}. S is the scale, z the zero-point and the *round* function rounds the result to the nearest integer.

$$X_q = \text{round}\left(\frac{X_{\text{fp32}}}{S} + z\right) \tag{4.1}$$

4.1.5 Dynamic quantization

Dynamic quantization (also applied post-training) is applied at runtime and typically only quantizes the model's weights while leaving the activations in floating-point (Gholami et al., 2022). This means that each layer's activations must be quantized on-the-fly during inference. The scale and zero-point values are therefore calculated during inference and not ahead of time as is the case with static quantization. This results in computational overhead and increased latency. While dynamic quantization can be more flexible and potentially more accurate for models with highly variable data or those that benefit from higher precision activations, the added complexity and computation time make it less optimal for the fixed and high-throughput operations typical of CNNs.

4.1.6 Quantization Aware Training (QAT)

Quantization Aware Training integrates quantization into the training process, allowing the model to adapt to the quantization-induced noise (Novac et al., 2021). This method simulates quantization effects during training, enabling the model to learn with quantized parameters. While QAT is more computationally intensive than post-training quantization due to the training requirement, it often results in higher accuracy for the quantized model.

4.2 Pruning

Pruning is a widely used technique in model compression that aims to reduce the size and complexity of neural networks by less important parameters. The presumption of pruning is that not all parameters in a model are crucial for its performance. The goal is to reduce the network's complexity and memory requirements, ideally without significantly impacting its performance. Pruning can be divided into two main types: structured and unstructured pruning

4.2.1 Structured pruning

Structured pruning removes entire neurons, filters or layers from the network, which leads to a reduction in the dimensionality of the network (He and Xiao, 2023). This method retains the structure of the components of the network that are left unchanged, which is beneficial for compatibility with standard hardware accelerators like GPUs and TPUs. It simplifies the model's architecture and can lead to substantial computational speedups. However, because structured pruning removes large portions of a network at once, there is a risk that important parts are removed, which in turn might have a relatively significant negative impact on the model's performance.

4.2.2 Unstructured pruning

This form of pruning targets individual weights across the network without regard to their organization or position within the layers (Vadera and Ameen, 2022). By setting specific weights to zero, unstructured pruning creates a sparse model where many of the connections between neurons are effectively removed. This can lead to significant reductions in model size and potentially speed up computations in systems optimized for sparse matrix operations. However, the irregularity of the sparsity pattern can pose challenges for achieving computational speedups on conventional hardware, as these systems are typically optimized for dense matrix operations.

4.2.3 Pruning strategies

Pruning deep learning models typically follows one of two main strategies: one-shot pruning and iterative pruning. One-shot pruning involves a single pruning process, either during or after training. This method is computationally efficient, but might not result in the most efficient pruning . Iterative pruning, on the other hand, consists of multiple cycles of pruning followed by fine-tuning, which allows the model to adapt gradually to the reduced number of parameters. While iterative pruning tends to yield better results than one-shot pruning, it comes with the trade-off of higher computational demands.

In one-shot pruning, the model undergoes a one-time pruning process based on specific criteria, which can be efficient but might not yield the best results due to the abrupt removal of parameters. Iterative pruning, however, adopts a more gradual approach by repeatedly pruning and then fine-tuning the model, allowing it to better adapt to the changes and potentially leading to improved performance, despite requiring more computational resources.

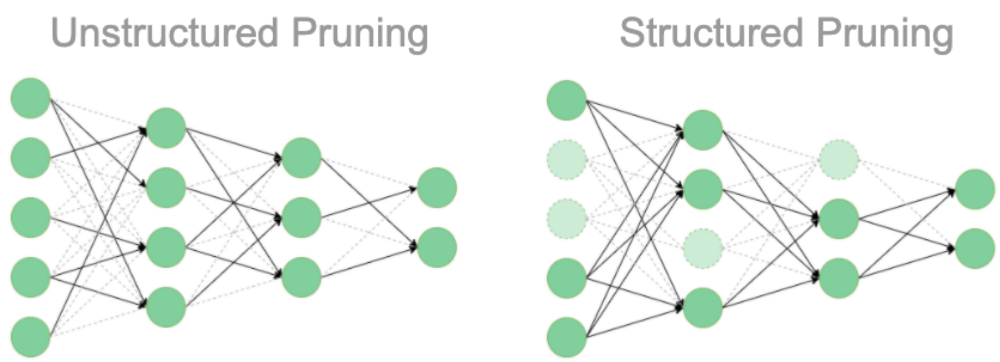

FIGURE 4.3: Unstructured vs structured pruning (Neuralmagic and Neuralmagic, 2023).

4.3 Knowledge distillation

Knowledge Distillation is a technique that aims to transfer the knowledge from a large, complex model (often referred to as the "teacher" model) to a smaller, more compact model (known as the "student" model) (Gou et al., 2021). The core idea behind this approach is to make the student model mimic the behavior of the teacher model as closely as possible.

The student model is trained using a dual-input strategy. This includes the raw data, as is used during traditional training, alongside the predictions or "soft targets" generated by the teacher model. Soft targets, which are the probabilities of each class predicted by the teacher, provide the student with insights into the teacher's reasoning process, not just the final decision. This blend of hard data and soft insights allows more nuanced decision-making capabilities from the teacher to be "distilled" into the student model.

Chapter 5

Sewer-ML: data, metrics and benchmarks

In this Chapter the Sewer-ML dataset will be introduced. First the contents and creation of the dataset will be explained. Secondly, commonly used performance metrics for CNNs will be discussed. And third and final part will delve into the CNNs that were trained on the Sewer-ML dataset and used to make predictions.

5.1 Data collection

The Sewer-ML dataset, introduced by Haurum and Moeslund (2021), consists of 75,618 sewer inspection videos collected between 2011 and 2019. This was done by three different water utilities in Denmark. Each video contains annotations by certified inspectors according to a Danish standard, which includes 18 distinct classes.

From these videos, individual frames are extracted at each annotated instance, creating a dataset that represents various conditions found within sewer systems. Each annotation marks a precise point in the video, denoting a specific class occurrence at a given moment and location within the sewer pipe. Annotations that are close to each other in the pipe are combined to form a multi-label dataset. Within each inspection video, an annotation is combined with all neighboring annotations found up to 0.3 meters before or 1.0 meters after the specified point in the pipe. The dataset not only identifies defects but also includes significant features like changes in pipe structure or connections (Haurum and Moeslund, 2021).

The final dataset contains 1,300,201 images, with a subset labeled as 'normal' indicating no defects, and another as 'defective' for those with annotations. The distribution of the sewer pipe images deemed either normal or defective over the training, validation and test datasets, are shown in Figure 5.2. The multi-label classification problem is framed as the prediction of the class labels, presented in Figure 5.1, except for the VA class. The absence of annotations implies a 'normal' pipe condition. The 'normal' class is therefore an implicit class.

5.2 Performance metrics

For the evaluation of the performance of the applied CNNs, several metrics are commonly used. Each of this metrics provide insights into different aspects of the model performance. Classification can have four possible outcomes:

Code	Description	CIW
VA	Water Level (in percentages)	0.0310
RB	Cracks, breaks, and collapses	1.0000
OB	Surface damage	0.5518
PF	Production error	0.2896
DE	Deformation	0.1622
FS	Displaced joint	0.6419
IS	Intruding sealing material	0.1847
RO	Roots	0.3559
IN	Infiltration	0.3131
AF	Settled deposits	0.0811
BE	Attached deposits	0.2275
FO	Obstacle	0.2477
GR	Branch pipe	0.0901
PH	Chiseled connection	0.4167
PB	Drilled connection	0.4167
OS	Lateral reinstatement cuts	0.9009
OP	Connection with transition profile	0.3829
OK	Connection with construction changes	0.4396

FIGURE 5.1: Class codes with description and the corresponding class-importance weights (Haurum and Moeslund, 2021).

Type	Training	Validation	Test	Total
Normal	552,820	68,681	69,221	690,722
Defective	487,309	61,365	60,805	609,479
Total	1,040,129	130,046	130,026	1,300,201

FIGURE 5.2: Distribution of images deemed normal or defective between the training, validation and test dataset splits. (Haurum and Moeslund, 2021).

- True Positives (TP): Instances correctly identified by the model as positive.
- True Negatives (TN): Instances correctly identified by the model as negative.
- False Positives (FP): Instances incorrectly identified by the model as positive.
- False Negatives (FN): Instances incorrectly identified by the model as negative.

Using these four possible outcomes, the following performance metrics can be calculated.

5.2.1 Precision

Precision quantifies the proportion of true positive predictions in the set of all positive predictions made by the model. It is particularly important in contexts where the cost of a false positive is high.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{5.1}$$

5.2.2 Recall

Recall (or Sensitivity) measures the proportion of actual positives that were correctly identified by the model. This metric is of particular interest in situations where missing a positive instance (a false negative) is costly.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{5.2}$$

5.2.3 F1 Score

The F1 Score is the, so called, harmonic mean of precision and recall. providing a balance between the two.

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5.3}$$

5.2.4 Accuracy

Accuracy is a metric that measures the proportion of true results (both true positives and true negatives) among the total amount of predictions made.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.4}$$

5.2.5 F2 score

When it comes to evaluating sewer defect classifications, there is not yet a consensus on which metric is ideal to apply (Haurum and Moeslund, 2020). Therefore, Haurum and Moeslund (2021) introduced a new metric that takes the relative importance per sewer inspection class into account. The classes are assigned scores based on their economic impact, determined by experts in the field. The scores are normalized between 0 and 1 to represent the class-importance weight (CIW). These weights represent a quantified relative importance for every class. The CIW for each class is shown in Figure 5.1.

The F2 score is based on the Fβ metric, shown in formula 5.5.

$$F_\beta = (1 + \beta^2) \frac{\text{Prc} \cdot \text{Rcll}}{\beta^2 \cdot \text{Prc} + \text{Rcll}} \tag{5.5}$$

Prc and Rcll represent precision and recall respectively. The β functions as a weight for the recall metric. When β is greater than 1, the formula gives more importance to recall compared to precision, which means it becomes more crucial for the model to capture all relevant instances in the dataset even at the cost of making more false positive errors. The β is set to 2 for this case.

The per-class F2 scores are calculated and then combined into a weighted average, via the novel $F2_{CIW}$ function (5.6). This function also uses the CIWs as input.

$$F2_{\text{CIW}} = \frac{\sum_{c=1}^{C} F2_c \cdot \text{CIW}_c}{\sum_{c=1}^{C} \text{CIW}_c} \tag{5.6}$$

5.3 Applied models and benchmarks

5.3.1 Sewer defect classification models

In recent times a variety of classification methods have been applied on the sewer defect classification problem, with the focus on three system configurations: end-to-end classifiers, binary classifiers working together as an ensemble and dual-stage classifiers (Haurum and Moeslund, 2021).

Chen et al. (2018) introduced such a dual-stage structure. The model consisted of a SqueezeNet (Iandola et al., 2016) followed by GoogleNet InceptionV3 model (Szegedy et al., 2014). The presence of significantly more normal images (images containing no defects) compared to any single type of abnormal sample (images containing a defect) can result in an unbalanced data distribution if all normal images are grouped into one class and each type of defect image into separate classes. This imbalance can negatively impact the model's performance. To address this issue, the detection process is structured in two phases: identifying abnormal images, which is followed by the classification of the specific types of defects. The model is initially trained using the ILSVRC2012 dataset (a large-scale dataset used for the so called "ImageNet Large Scale Visual Recognition Challenge". It includes millions of images across thousands of categories). Following this, the model is fine-tuned with sewer pipe image, effectively applying the principle of transfer learning to adapt the model to the specific task. For the data that was used to fine tune the model, approximately 10,000 normal images were selected from a detection video supplied by a pipeline robot company, along with about 2,000 images for each type of defect. A total of four defect classes were used in the dataset: Intrusion, blur, deposition, and obstacles.

Another hierarchical dual-stage classifier was created by Xie et al. (2019). In the dataset that was used to train the model, 50% of the images were classified as normal, with the rest divided among 16 distinct defect categories, creating a significant imbalance in the data. Once again, to tackle this problem, a hierarchical classification framework was developed, incorporating two CNNs. The first CNN is aimed at binary classification, differentiating normal from defective images (NDCNN), and the second is dedicated to identifying the specific defect type within the defective images, an interdefect classifier (IDCNN). Both CNN models have identical architectures, except for the final output layer. This allowed the second model to be developed by fine-tuning the first model's weights. For the binary classification, the dataset is split into two groups: normal images as positive examples and defective images as negative examples. This mitigates the data imbalance issue and facilitates the model to be trained from the ground up. Once the NDCNN model is adequately trained, its output layer is modified to a six dimensional vector to accommodate multi-class defect identification, thus creating the IDCNN model. The six dimensions vector was chosen because they aimed at identifying the six most common defect classes (barrier, high water level, stagger, fracture, deposition, disjunction). If an image is classified as defect but not classified as one of the six defect classes, it implies that it is one of the ten classes that is not often encountered and therefore the implicit label becomes 'other'. After the modification of the output layer, the IDCNN is fine-tuned using the defective images based on the pre-trained NDCNN weights. This approach overcomes the issue of insufficient defective image samples for training the IDCNN model from scratch by employing transfer learning, which uses the learned parameters from NDCNN for IDCNN. This method reduces the need for a large sample size.

Myrans et al. (2018) created a model that consists of binary classifiers working together as an ensemble. First sewer images were transformed using GIST descriptors, which condense the images into a comprehensive representation by capturing key spatial and textural information. This step ensures that the subsequent classification focuses on the most relevant features of the images. Following the transformation, the model uses Random Forest classifiers to determine the presence and type of defects. There are 13 Random Forest classifiers, each dedicated to one of the defect categories: joint, deposits, multiple faults, crack, surface, roots, infiltration, obstacles, other, broken/collapsed, hole, brickwork, and deformation. Each one of the 13 Random Forest classifiers is tasked with making a binary decision regarding the presence of its corresponding defect type. The model was trained on a dataset created by extracting images from CCTV footage collected by Wessex Water. which included over 2,260 labeled defects.

Hassan et al. (2019) used an AlexNet (Krizhevsky et al., 2017) to identify six different defect classes (debris, surface damage, lateral damage, joint open, joint faulty, and longitudinal crack). This model was created in the same manner as was done by Chen et al. (2018) namely, it was pre-trained on the ILSVRC2012 dataset and fine tuned with a dataset containing sewer pipe defect images. These images were retrieved from 6,605 CCTV sewer pipelines inspection videos. These videos were supplied by the Korea Institute of Civil Engineering and Building Technology and the training dataset totalled 47,072 images.

5.3.2 Sewer-ML benchmarks

To compare the performance of several relevant models in the sewer defect classification field, Haurum and Moeslund (2021) trained the models of Chen et al. (2018), Xie et al. (2019), Myrans et al. (2018), Hassan et al. (2019) from scratch, on their own Sewer-ML dataset. They also did this for the following general models: ResNet-101 (He et al., 2015), KSSNet (Wang et al., 2019), TResNet-M/L/XL (Ridnik et al., 2020). The training methodology that was adopted is based on the approach outlined by Goyal et al. (2018) for effective training of models on the ImageNet dataset.

Haurum and Moeslund (2021) start with the preparation of the images, which undergo several modifications. They are resized to 224x224 pixels and have a 50% likelihood of being flipped horizontally. Their brightness, contrast, saturation, and hue are also adjusted, varying by up to 10% from their original levels. Additionally, these images are standardized based on the mean and standard deviation of the dataset's channels. At the inference stage, images are merely resized to 224x224 and then standardized. In the case of using the InceptionV3 network, as referenced by Chen et al. (2018), the required image size is adjusted to 299x299 pixels. When the GIST features in the model of Myrans et al. (2018) are employed, the images are transformed into gray scale and resized to 128x128 pixels.

All deep learning models undergo training over 90 epochs, processing 256 samples per batch through Stochastic Gradient Descent (SGD) with added momentum. The initial learning rate is set at 0.1, with momentum at 0.9 and a weight decay factor set to 0.0001. To adjust the training intensity, the learning rate is scaled down by a factor of 0.1 during the 30th, 60th, and 80th epochs.

The training process utilizes the modified binary cross-entropy loss usable for multi-label classifiers, referenced in Equation 2.

The binary cross-entropy loss in equation 5.7 is used during training. This loss function is regularly used in the field of multi-label image classification (Haurum and Moeslund, 2021).

$$L(x, y) = -\frac{1}{C}\sum_{c=1}^{C}[w_c y_c \log(\sigma(x_c)) + (1 - y_c)\log(1 - \sigma(x_c))] \tag{5.7}$$

In equation 5.7, C signifies the total count of classes. y_c indicates the presence of class c in a given image, with a value of 1 if present and 0 if absent. The variable x_c represents the model's raw output for class c, and σ denotes the sigmoid function. The weight for class c, denoted as w_c, is calculated based on the proportion of negative to positive instances for that class.

Given the imbalance in the dataset, each positive observation of a class is assigned a weight, w_c, based on the ratio of negative to positive observations for that class, as defined in equation 5.8. This approach ensures that the contributions to the loss from underrepresented classes are amplified, whereas those from over represented classes are diminished. Additionally, in the case of the InceptionV3 network, an auxiliary classifier contributes a lesser weighted loss to the overall calculation.

$$w_c = \frac{N - N_c}{N_c} \tag{5.8}$$

After training all the models were run on both the validation and test dataset. The results can be seen in Table 5.1.

Model	Validation		Test	
	$F2_{CIW}$	$F1_{Normal}$	$F2_{CIW}$	$F1_{Normal}$
Xie et al.	48.57%	**91.08%**	48.34%	**90.62%**
Chen et al.	42.03%	3.96%	41.74%	3.59%
Hassan et al.	13.14%	0.00%	12.94%	0.00%
Myrans et al.	4.01%	26.03%	4.11%	27.48%
ResNet-101	53.26%	79.55%	53.21%	78.57%
KSSNet	54.42%	80.60%	54.55%	79.29%
TResNet-M	53.83%	81.23%	53.79%	79.91%
TResNet-L	**54.63%**	81.22%	**54.75%**	79.88%
TResNet-XL	54.42%	81.81%	54.24%	80.42%

TABLE 5.1: Performance metrics for each model on the validation and test sets (Haurum and Moeslund, 2021).

The $F1_{Normal}$ score is the $F1$-score for the normal pipes (pipes without a defect). This score reflects how good a model is in discerning defect pipes and normal pipes. Looking at the performances of the models on the validation dataset, it can be seen that the TResNet-L performs best with regards to the $F2_{CIW}$ score (54.63%), and the model of Xie et al. (2019) performs best with regards to the $F1_{Normal}$ score (91.08%). When looking at the best $F2_{CIW}$ and $F1_{Normal}$ scores for the test dataset, the same models have yet again the highest scores with 54.75% and 90.62% for the TResNet-L and the model of Xie et al. (2019) respectively.

Chapter 6

Methodology

This Chapter starts with a small introduction on the ResNet-101 architecture. This is needed to understand a part of the explanation of the results in Chapter 7. After this the applied compression methods are explained. Both the pretrained ResNet-101 and the TResNet-L were retrieved from the database provided by Haurum and Moeslund (2021). Coding was done on Google Colab.

6.1 ResNet-101 model architecture

A Residual Neural Network (ResNet) derives it name from the use of residual blocks. These blocks use so-called skip connections. Within each residual block, the input is channeled through a series of layers and then, through a skip connection, is also directly added to the output of these layers. Skip connections effectively allow the input to "skip over" some of the intermediate layers. This approach not only helps in mitigating the vanishing gradient problem but also enables the training of much deeper networks (He et al., 2015).

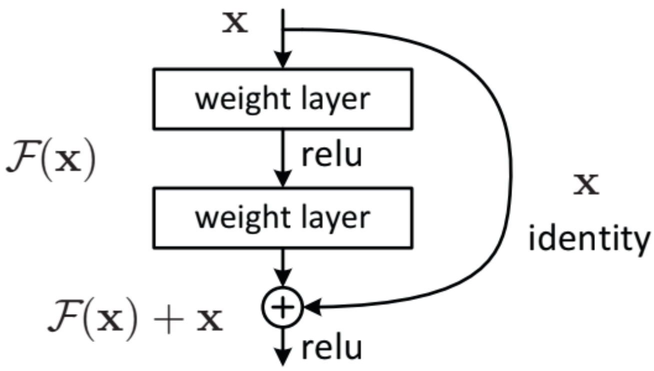

FIGURE 6.1: Residual block example (He et al., 2015).

ResNets also incorporate batch normalisation layers. This technique used to standardize the inputs to a layer for each mini-batch. This helps stabilize the learning process by reducing internal covariate shift, which is the change in the distribution of network activations due to the updating of weights (Ioffe and Szegedy, 2015). By normalizing the output of the previous layer to have a mean of zero and a variance of one, batch normalization allows for higher learning rates and makes the model less sensitive to its initial parameter settings.

6.2 Static quantization

The `state_dict` of the pretrained model contains the weights and in order to load these into a quantizable ResNet-101 architecture, first the ResNet-101 model definition code must be altered to prepare it for quantization. Appendix A contains the entire code, now only the changes will be discussed.

To specify where the quantization and dequantization should take place it the model, two commands are added to the `def __init__` method. At the beginning of the model `self.quant = torch.ao.quantization.QuantStub()` is used to mark the point where floating-point tensors should be converted to quantized tensors. At the end of the model `self.dequant = torch.ao.quantization.DeQuantStub()` is added to transition the values back to floating-point, ensuring the model's output can be used in applications expecting floating-point tensors. To actually initiate these defined quantization and dequantization commands at there specified locations in the model, `x = self.quant(x)` and `x = self.dequant(x)` are added to the `_forward_impl` method.

Also, in the `Bottleneck` part of the ResNet-101 model, a new line `self.skip_add = nn.quantized.FloatFunctional()` is added. This change replaces the standard addition method in skip connections. The `FloatFunctional` module is specifically designed to enable addition operations for quantized tensors, ensuring that the summed quantized values are a correct representation of the real values.

After these changes are applied, the `state_dict` of the pretrained model can now be loaded in a quantizable ResNet-101 structure and the model can be quantized. Calibration is performed with several datasets of different sizes. After quantization the model is ran on the validation dataset, in order to quantify the performance. The code used to quantize the model and run it on the validation dataset is shown in appendix B.

6.3 Layer fusion

Layer fusion is a process where suitable sets of consecutive layers are merged into a single, more efficient layer. This can reduce memory usage and computational overhead, making the model faster during inference (Alwani et al., 2016).

The layer fusion process is started with fusing the first convolutional layer (conv1), followed by batch normalization (bn1) and ReLU activation (relu). These are fused into a single module using the `torch.ao.quantization.fuse_modules` function. Each Bottleneck block in ResNet-101 consists of three sets of convolutional and batch normalization layers. The code `fuse_bottleneck_layers` function iteratively fuses these layers across all Bottleneck blocks within the model. For Bottleneck blocks that contain a downsample module (used to match dimensions between input and output of the block), the convolutional and batch normalization layers within downsample are also fused.

After the layers are fused, static quantization is performed in the same manner as before. The entire code can be found in appendix C

6.4 Iterative pruning

Iterative pruning is chosen over one-shot pruning, because it yields a better performance (Min and Motani, 2022). L1 norm unstructured pruning is performed. This entails the pruning of the weights with the lowest absolute values first, as they are deemed less significant for the model. A pruning rate of 0.1 is chosen per iteration, effectively pruning away 10% of the original amount of weights for each iteration. After each pruning operation, the model is fine tuned to recover the model performance loss. This is done with a dataset containing 10,000 randomly selected images from the training dataset. Adam optimizer with a learning rate of 1e-4 is chosen and for the loss function the standard binary cross entropy loss is picked. After the fine tuning is done, the model is run on the validation dataset and the model is saved. That model is then used for the next iteration. This is done for a total of ten iterations, ending with all layers having zero weights. See appendix D for the full code.

6.5 Iterative pruning in combination with layer fusion and static quantization

The model first undergoes iterative pruning up until a pruning rate that is deemed best in regards to balancing the performance loss and pruning rate. After that, the model undergoes layer fusion and static quantization. All processes are done in the same manner as described in the previous paragraphs, only they are done in sequence this time.

6.6 Work order

The above described techniques are first applied to the ResNet-101 model. After this, the TResNet-L will be compressed. This is only done via pruning, because the TResNet-L model needs to be ran on a GPU in order to execute the *In-Place Activated BatchNorm* layers. However, quantized inference currently is not supported for GPUs (*Quantization — PyTorch 2.3 documentation* n.d.).

Chapter 7

Results

In this Chapter the yielded results of the compression techniques will be discussed.

7.1 ResNet-101

7.1.1 Static quantization

The model undergoes quantization, transitioning from a 32-bit representation to an 8-bit one. This effectively reduces the model's size by a factor of four. The performances with regards to the $F2_{CIW}$ score and the $F1_{Normal}$ scores are plotted in Figure 7.1 and Figure 7.2 respectively. These graphs display the model's performance relative to the size of the dataset used for calibration during the quantization procedure. It can be seen that in both cases the calibration process reaches a plateau quickly. This implies that the initial batch of images is enough to determine the scale factors and zero points necessary to adequately convert the 32 -bit model to the quantized 8-bit model. Additional calibration is therefore unnecessary. This aligns with findings from Hubara et al. (2021). While this study primarily focuses on 4-bit quantization using methods like AdaQuant, AdaRound, and QAT-KLD, the principle of using a small calibration set to minimize quantization errors is similar.

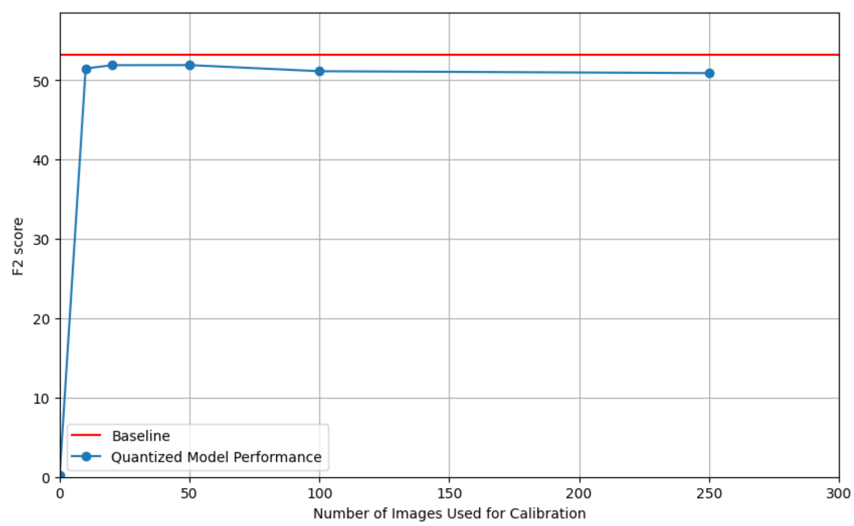

FIGURE 7.1: Quantized model $F2_{CIW}$ score for each model based on the size of the calibration dataset.

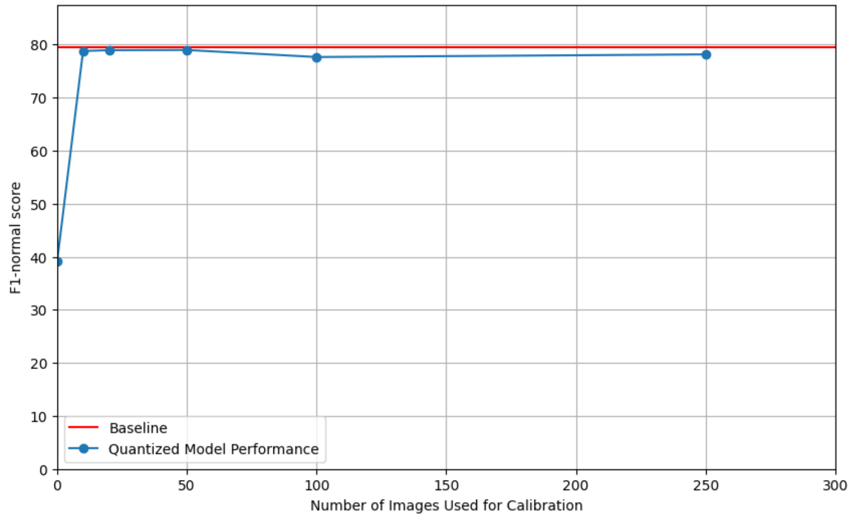

FIGURE 7.2: Quantized model $F1_{Normal}$ score for each model based on the size of the calibration dataset.

This phenomenon could be the result of extensive use of batch normalization layers in a ResNet-101. Namely, every convolutional layer in the model is followed up by a batch normalisation layer. Batch normalization normalizes the activations in the network (Ioffe and Szegedy, 2015), which could help in stabilizing the range of values that the observers encounter during calibration. By making the distribution of activations more consistent across batches, it might be easier for the observers, that were put into the model during the calibration step, to find suiting parameters that are used to map the tensor values from floating point to integer.

The skip connections can also contribute to more stable activations across layers by combining the outputs of convolutional blocks with the input activations. This stability might help in maintaining a consistent range of values across the network, allowing observers to quickly determine suitable quantization parameters.

The $F2_{CIW}$ and $F1_{Normal}$ scores for the standard ResNet-101 are 53.26 and 79.55 respectively. When looking at the scores for the quantized model that uses a dataset of 50 images for calibration, slightly lesser scores are achieved , namely: a $F2_{CIW}$ score of 51.92 and a $F1_{Normal}$ score of 78.98, which equals a decline of 2.52% and 0.72% respectively. This slight decline in performance is as expected for a quantized model that is four times smaller than the original model.

Quantizing the model also improved the inference speed and the corresponding image throughput. Table 7.1 displays the inference speed, throughput and model size of each version of the ResNet-101 model that has been produced within this thesis. The inference speed and image throughput are calculated on both the CPU and L4 GPU, which is a GPU that is designed for inference acceleration of AI tasks. For inference on the CPU, a throughput (images processed per second) increase of 64.50% compared to the standard ResNet-101 model is observed and for inference on the L4 GPU the increase is 87.81%.

Model	CPU			L4 GPU			
	Inference Time (s)	Throughput (img/s)	Change %	Inference Time (s)	Throughput (img/s)	Change %	Model Size (MB)
Standard ResNet-101	0.151	6.62	–	0.062	16.16	–	162.82
Quantized	0.092	10.89	+64.50%	0.033	30.34	+87.81%	42.45
Quantized & Fusion	0.077	12.92	+95.02%	0.023	44.35	+174.50%	41.68
Pruned 30%	0.137	7.32	+10.57%	0.061	16.29	+0.80%	162.82
Pruned 30%, Quantized & Fusion	0.073	13.77	+108.01%	0.022	45.60	+182.16%	41.68

TABLE 7.1: Inference speed, throughput on CPU and L4 GPU, and size of each model with percentage change in throughput relative to Standard ResNet-101.

7.1.2 Layer fusion

This time layer fusion was applied to ResNet-101 model before applying quantization. This modification has not impacted the calibration process, since the calibration plateau is reached as quickly in the case of quantization without layer fusion for both the $F2_{CIW}$ and $F1_{Normal}$ score. These are plotted in Figure 7.3 and Figure 7.4 respectively. The exact numeric values for the models calibrated with a dataset consisting of 50 images can be found in table E.1. The model performs slightly worse than the quantized model without layer fusion, however the difference can be considered negligible, since the difference is less than 0.5%. The performance improvement that layer fusion brings is realised in the improved inference speed and corresponding throughput of the model, not only compared to the standard ResNet-101 model, but also to the quantized model without layer fusion. Compared to the standard ResNet-101 model the throughput increased by 95.02% and on the L4 GPU the increase reached 174.50%. Compared to the quantized model without layer fusion, this amounted to a percentage point increase of 30.52% and 86.69% for throughput on the CPU and L4 GPU respectively. The model size was also slightly reduced by 0.77 MB. These findings demonstrate the positive impact layer fusion has on inference speed-up while maintaining near similar performance as the non layer fused quantized model.

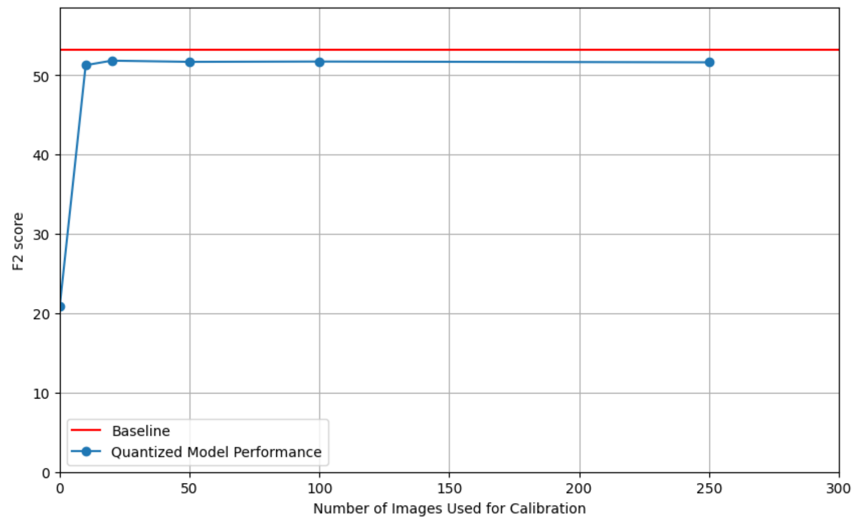

FIGURE 7.3: Layer fusion and quantized model $F2_{CIW}$ score for each model based on the size of the calibration dataset.

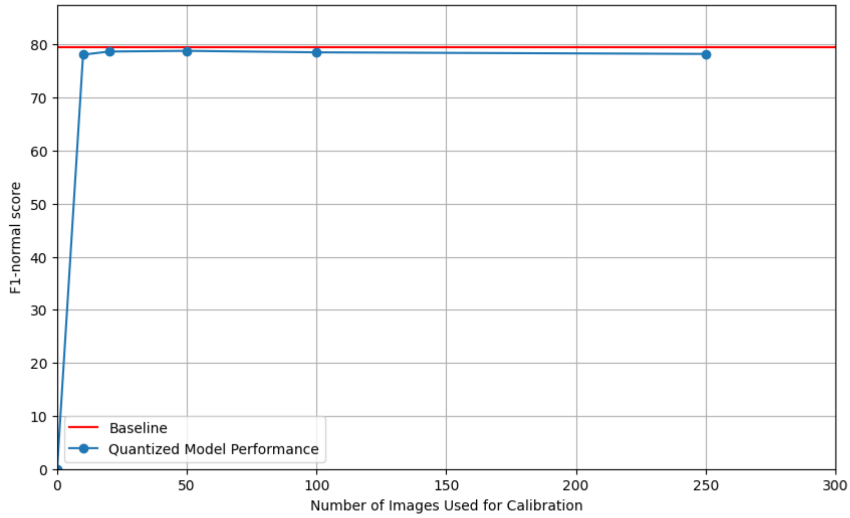

FIGURE 7.4: Layer fusion and quantized model $F1_{Normal}$ score for each model based on the size of the calibration dataset.

7.1.3 Iterative pruning

After pruning at a rate of 0.1, an immediate and significant drop in the $F2_{CIW}$ score can be seen, see Figure 7.5. This suggests that even minimal pruning has significant effect on the model ability to accurately predict the defect classes, including the ones with a high class importance weight.

As the pruning rate increases to 0.3, the $F2_{CIW}$ score tends to stabilize. This could indicate that the pruning process has moved past the initial critical weights and is now removing weights that contribute less to model performance. The plateau suggests that the model has a degree of resilience and can tolerate some level of pruning without further significant losses in performance. For pruning rates beyond 0.3, the $F2_{CIW}$ score keeps on gradually declining.

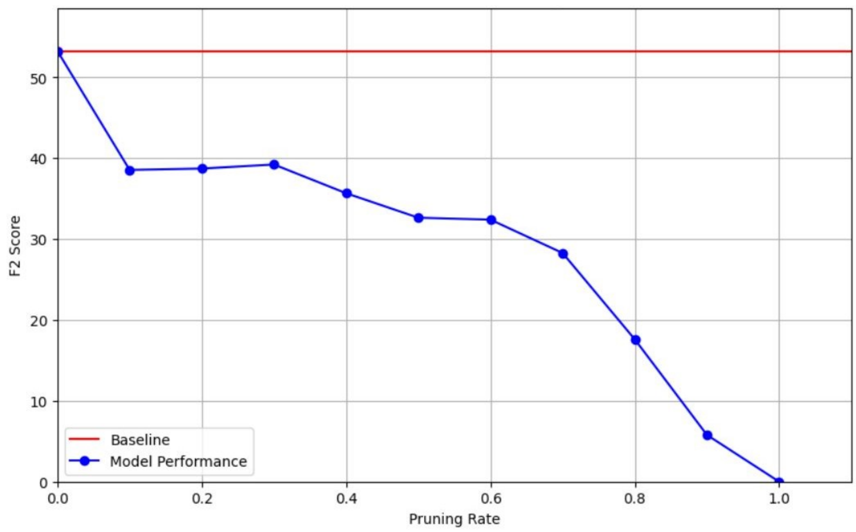

FIGURE 7.5: Pruned ResNet-101 $F2_{CIW}$ score for each pruning rate.

When looking at the per-class $F2$ scores in Table E.2, a significant decrease can be seen in the vast majority of classes for the 10% and 30% pruned models. The same can be seen in the per-class recall scores in Table E.4. This is as expected, since the $F2$ score prioritizes recall over precision. These declines indicate that the model's effectiveness in recognizing true defects has been diminished. In particular, the performance drop in high-importance classes such as RB (cracks, breaks, and collapses) and FS (displaced joint) is alarming, because overlooking such defects could result in severe consequences for the structural integrity of the sewer piper and therefore its functionality.

The decrease in $F2$ score and recall indicates that the model is now prone to missing more actual defects than before pruning. This shift towards fewer predictions per class reflects a change in the model's classification behavior, prioritizing the avoidance of false positives over the detection of all true positives. While this may lead to a higher precision, as can be observed in Table E.3, the model's usability is significantly reduced in practical application, where the cost of false negatives is high.

Looking at Figure 7.6, an increase of the $F1_{Normal}$ score can be observed up until the 70% pruning rate mark. This means that the model is better at predicting non-defective pipes. However, this is simply the result of the model becoming more conservative with regards to assigning defect classes, as a normal pipe is defined by the absence of any defect classes. This conservatism of the model is also reflected in the decline of the $F2_{CIW}$ score. Table F.2 displays the total amount of images that were deemed normal by each variant of the ResNet-101 model. Comparing the standard ResNet-101 and the 30% pruned version, an increase from 47,234 to 76,672 (increase of 62%) can be observed in the amount of normal predictions. To put this into perspective, the actual amount of normal images in the validation dataset is 68,681. In Table F.1 the per-class prediction count is shown. A very significant drop for every class is observed. The three classes with the highest class importance weight (RB, OS and FS) show a drop of 94.08%, 94.85% and 46.61% respectively. This confirms that the model changed to be more conservative in assigning defect classes.

In Table 7.1, no change in model size can be observed. This is to be expected, because in the applied pruning method the weights that are pruned are set to zero. Therefore, The total amount of variables and, consequently, the model size of the network does not change. Significant inference speed-up compared to the standard ResNet-101 can also not be observed for the same reason. Namely, the weights might be zero, but the amount of MACs (multiply-accumulate operations) does not change. To realise possible model size reduction and inference speed-up, techniques that employ sparsity aware model saving and inference must be utilized. This is something that can be addressed in future research.

7.1.4 Iterative pruning in combination with layer fusion and static quantization

This model is performance wise similar to the pruned-only version, as can be seen in Figure 7.7 for the $F2_{CIW}$ score and in Figure 7.8 for the $F1_{Normal}$ score . It shows a significant decrease in the $F2_{CIW}$ score to 40.09, pointing to reduced effectiveness in detecting defects, see Figure. The $F1_{Normal}$ score, however, rises to 87.41, which

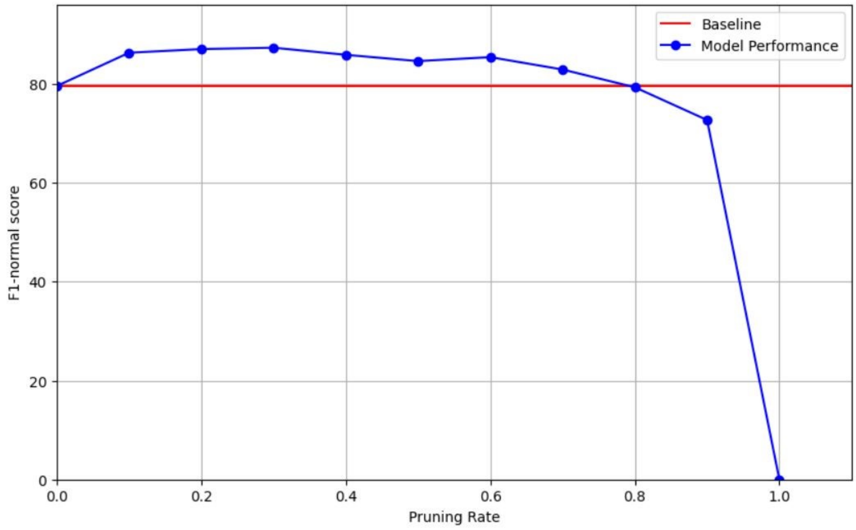

FIGURE 7.6: Pruned ResNet-101 $F1_{Normal}$ score for each pruning rate.

indicates an improved ability to identify pipes without defects. Both these phenomena are similar to those that can be observed for the pruned-only model, although these results are slightly better. The $F1_{Normal}$ score has a negligible improvement of a tenth of a percent, the $F2_{CIW}$ score however increased by 2.2% . Han et al. (2016) also found that model performance of a pruned-quantized model was superior over the pruned-only version.

The trend of the model becoming more conservative in its predictions, resulting in fewer false positives, is also present here. This can be noted by the increase in images classified as normal, from 47,234 to 75,886, as shown in Table F.2.

Also, similar to the quantized model with layer fusion, there is an improvement in the model's inference speed and size compression. The inference speed on the CPU improves to 0.073 seconds, and throughput increases to 13.77 images per second. On the L4 GPU, the inference speed further increases to 0.022 seconds with a throughput of 45.60 images per second, as detailed in Table 7.1. These improvements are the most significant of all models, with an increase of 108.01% for the throughput on the CPU, and increase of 182.16% on L4 GPU, both relative to the standard ResNet-101 model. The model is equal to the model were layer fusion and quantization at 41.68 MB.

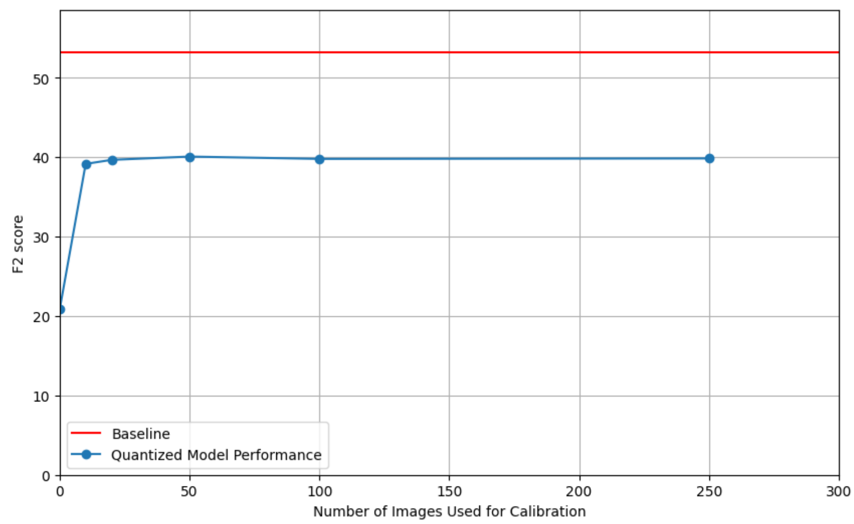

FIGURE 7.7: 30% pruned, layer fusion and quantized model $F2_{CIW}$ score for each model based on the size of the calibration dataset.

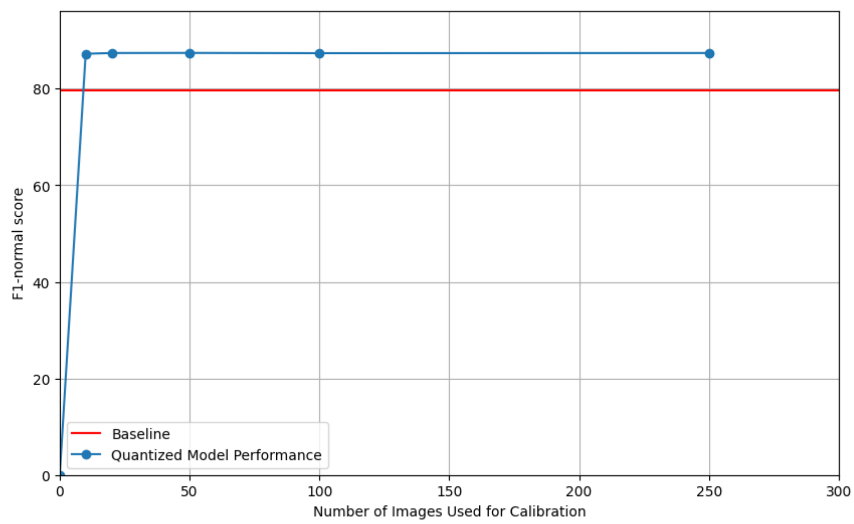

FIGURE 7.8: 30% pruned, layer fusion and quantized model $F1_{Normal}$ score for each model based on the size of the calibration dataset.

7.2 TResNet-L

7.2.1 Iterative pruning

In examining the performance of the TResNet-L model in response to iterative pruning, as depicted in Figure 7.9, a similar initial drop in the $F2_{CIW}$ score at a pruning rate of 0.1 is observed, comparable to the ResNet-101 model. This initial decline suggests a notable impact on the model's ability to accurately classify defects, especially those carrying a high class importance weight. However, unlike the trend seen in the ResNet-101 model, the TResNet-L $F2_{CIW}$ score stabilizes and remains relatively consistent until a pruning rate of 0.8 is reached. This suggests that the TResNet-L model has a higher tolerance for weight removal without substantial loss in its ability to identify defects. The per-class metrics in Table G.1 also see a initial decline in similar fashion as the for the pruning of the ResNet-101 model.

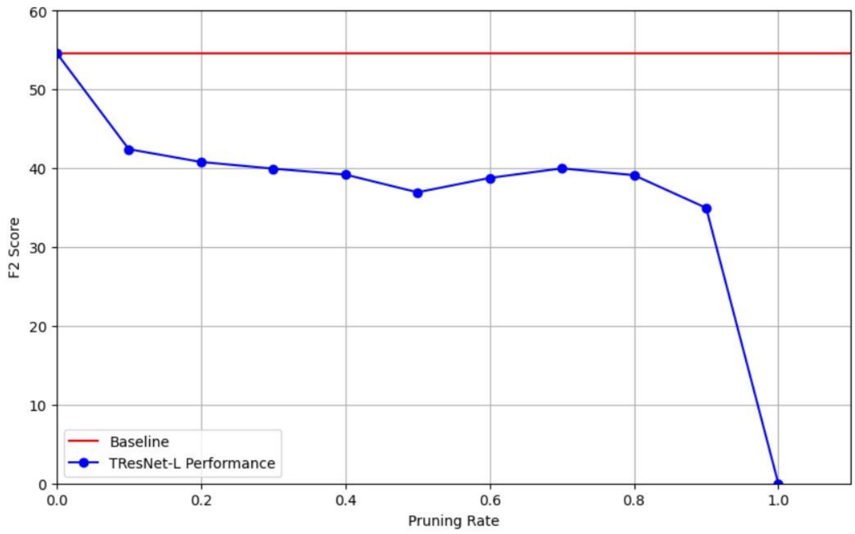

FIGURE 7.9: TResNet-L $F2_{CIW}$ score for each score for each pruning rate.

The $F1_{Normal}$ score also goes up, comparable with the Resnet-101 model, but this time stays above the baseline score of the standard TResNet-L model up until a pruning rate of 0.9, see Figure 7.10. The same conservative prediction behaviour as in the ResNet-101 model can be observed for per-class prediction counts in Table G.1. This results in a higher total count for predicted normal images, see Table G.2.

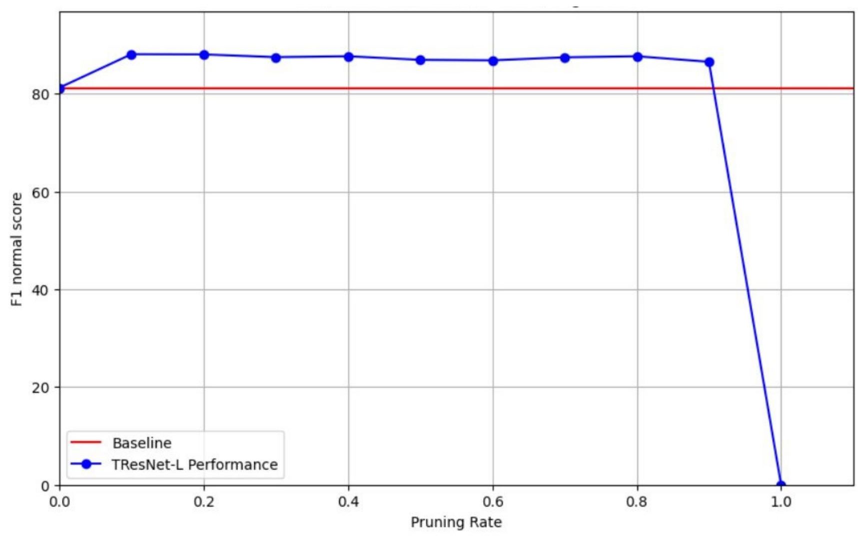

FIGURE 7.10: TResNet-L $F1_{Normal}$ score for each score for each pruning rate.

7.3 Practical implications

In sewer asset management, the use of predictive models can be useful for supporting decision making regarding maintenance and repairs. Accurate detection of severe defects allows sewer asset managers to prioritize repairs effectively, focusing resources on the most urgent issues to maintain the structural integrity and functionality of sewer systems. This implies that a sewer asset manager might judge the usability of a model by its ability to correctly identify the most critical defect per image, as this typically dictates the urgency of possible repair interventions. The most urgent defect for a given image is determined by the defect with the highest CIW score among all identified defects. This ranking directly influences the priority of potential interventions compared to other defective sections within the sewer system. In Table 7.2 a count of images with correctly identified highest CIW score defects for all the versions of the ResNet-101 model that were created in this thesis is presented.

Model	Images with Correct Highest CIW Defect	Recall Score (%)
Standard ResNet-101	56,289	91.70%
Quantized	56,720	92.43%
Quantized & fusion	56,572	92.19%
Pruned 30%	36,042	58.74%
Pruned 30%, quantized & fusion	36,662	59.75%

TABLE 7.2: Count of images with correctly identified highest CIW defect and recall scores for various model configurations, based on the ground truth total of 61,365 defective images.

The standard ResNet-101, quantized model, and layer fused & quantized model show high effectivenes with recall scores of 91.70%, 92.43%, and 92.19% respectively. These scores suggest that a significant majority of the most critical defects are correctly identified, ensuring that necessary repair actions can be informed accurately. In contrast, the 30% pruned model and the 30% pruned, in combination with layer fusion and quantization model, exhibit lower recall scores at 58.74% and 59.75%. This decline in performance indicates a higher likelihood of missing critical defects, which could delay or prevent essential maintenance actions.

An interesting phenomenon is that both the quantized and the layer fusion in combination with quantization models have a higher recall score than the standard ResNet-101 model. This is the result of those models being more likely to assign defects to images, as per table F.1. However, these models maintain highly usable compared to the standard model, because they also maintain a significant $F1_{Normal}$ scores.

Chapter 8

Discussion

8.1 Compressed models

Several compressed models have been created of the pre-trained ResNet-101 model provided by Haurum and Moeslund (2021). The reduction of the ResNet-101 model from a 32-bit to an 8-bit representation not only achieved a significant reduction in model size but also improved inference speed as a result of a lowered computational load when processing input. This acceleration is crucial for edge devices, since these devices are often constrained by limited computational power.

Model size reduction was also realized with the use of pruning. Although both the ResNet-101 and TResNet-L models saw a significant initially drop in performance, for both a plateau followed. For the TResNet-L this plateau lasted up until a pruning rate of 80%. This stabilization suggests that, at least after the initial performance drop, the networks can maintain some predictive capabilities despite the reduction of redundant parameters. In this thesis sparsity aware saving and inference was not realised due to the relative practical difficulty of this and therefore the potential size reduction and inference speed improvement are not reported. However, this is possible and when pruning away 80% of a models variables away, a significant compression and speed-up is to be expected.

The combination of pruning with layer fusion and quantization for the ResNet-101 has demonstrated that the impact of quantization on the performance of the pruned model is minimal. This once again demonstrates the effectiveness of quantization. This combination is promising, but with the current results a the ResNet-101 that is compressed using layer fusion and quantization is the feasible model for edge deployment due to the combination of potential significant inference speed-up, model size reduction and performance preservation. The 92.19% recall score for the highest CIW score defect detection is also a strength of this model that is used for a task where false negatives could have grave consequences.

8.2 Future studies

The observation that the current pruning method leads to an initial decline in model performance underscores a critical area for future research. The aim should be to develop a more refined pruning approach that is more effective at minimizing performance losses with the current strategy. Different pruning strategies should also be explored. The pruning method introduced by Frankle and Carbin (2019) has proven to be an effect method and has the potential to be effective for the models in this thesis as well. In this method, the techniques used to decide which parts to prune and how to fine-tune the network are different compared to the techniques applied in this thesis. Realising sparsity aware saving of the model and model inference should also be researched in order to achieve the potential compression benefits of the pruning technique. Furthermore, a significant difference between inference speed of a model ran on either the CPU or L4 GPU has been observed. This phenomenon emphasizes the importance of hardware optimization to achieve the highest possible performance of a given model. Therefore, research that focuses on the implementations of compressed sewer defect inspection models on optimal hardware is advisable.

Chapter 9

Conclusion

9.1 Implementation

Research question 1 : How are the compression techniques implemented into the original models?

The ResNet-101 model underwent a static quantization process. Initially, a calibration step was performed using a subset of the training dataset to accurately determine scale factors and zero points for the conversion from 32-bit floating-point to 8-bit integers. This step ensured that the quantized model would not have severely diminished performance. In the implementation of layer fusion, the first three layers were merged (convolutional, batch normalization and ReLU layers) and in the Bottleneck blocks the convolutional layers were merged with batch normalization layers. For the pruning step L1 norm unstructured pruning is performed. A pruning rate of 0.1 is chosen per iteration, effectively pruning away 10% of the original amount of weights for each iteration. After each pruning operation, the model is fine tuned to recover the model performance loss. This is done with a dataset containing 10,000 randomly selected images from the training dataset. Adam optimizer with a learning rate of 1e-4 is chosen and for the loss function the standard binary cross entropy loss is picked.

9.2 Performance

Research question 2: How do the compressed models perform compared to the uncompressed models?

The $F2_{CIW}$ and $F1_{Normal}$ scores for the standard ResNet-101 are 53.26 and 79.55, respectively. For the quantized model, calibrated with a dataset of 50 images, the scores slightly decrease to 51.92 and 78.98 for $F2_{CIW}$ and $F1_{Normal}$, representing declines of 2.52% and 0.72%, respectively. In terms of throughput on the CPU, there is a 64.50% increase over the standard ResNet-101, and for the L4 GPU, the increase is 87.81%. Quantization reduces the model's size by fourfold. Layer fusion contributes to enhanced inference speed and throughput, not only compared to the standard ResNet-101 but also to the quantized model without layer fusion, with throughput improvements of 95.02% and 174.50% on the CPU and L4 GPU, respectively.

Iterative pruning was applied to both the ResNet-101 and TResNet-L models. After the first pruning iteration at a rate of 0.1, there is a noticeable drop in the $F2_{CIW}$ score. Subsequently, the ResNet-101 maintains a stable performance up to a pruning rate of 0.3 ($F2_{CIW}$ score of 39.23 and $F1_{Normal}$ score of 87.30), while the TResNet-L does so up to a rate of 0.8 ($F2_{CIW}$ score of 39.13 and $F1_{Normal}$ score of 87.68). The increased $F1_{Normal}$ score indicates that the models have become more conservative in classifying defects, leading to more pipes being classified as normal. The 30% pruned ResNet-101 model, which also underwent layer fusion and static quantization, matches the performance of the model pruned at a rate of 0.3. For this model, throughput on the CPU improved by 108.01% and on the L4 GPU by 182.16%, relative to the unmodified ResNet-101 model.

These results underscore the trade-offs and benefits of compressing the models. The slight decreases in $F2_{CIW}$ and $F1_{Normal}$ scores for the quantized ResNet-101 reflect a minor impact on defect detection performance, which is an acceptable trade-off considering the substantial gains in computational efficiency. The significant increases in throughput on both the CPU and L4 GPU indicate that quantization and layer fusion techniques effectively enhance the model's inference speed, making it more suitable for real-time applications on resource-constrained devices. Iterative pruning results show an initial performance drop, but the models stabilize and maintain reasonable defect detection capabilities up to certain pruning rates.

9.3 Usability

Research question 3: How usable are the compressed models for sewer asset management?

Compressed models can enhance Dutch sewer asset management by accurately detecting critical defects, while using minimal hardware resources. Accurate detection of severe defects is necessary for timely decision-making in regards to maintenance and repair. This in term is crucial for upholding the sewer systems' structural integrity. The quantized ResNet-101, both the one with and without layer fusion, even improve compared to the standard model in regards to correctly identifying the highest CIW defect class present in pipes deemed defective in the validation dataset. This model behaviour is positive because the priority of a sewer asset manager is the discovery of the defect that carry the highest risk with them if not treated in time. This improved efficiency in defect recognition helps in optimizing repair schedules and resource allocation, thus reducing operational costs as well.

The findings of this thesis demonstrate that sewer defect detection models can be compressed while maintaining good performance. This indicates that accurate defect detection is achievable with reduced hardware requirements. Additionally, it highlights that advanced compression techniques can enhance the efficiency of sewer maintenance and repair, making it feasible for future implementation on edge devices that perform sewer inspections.

9.4 Recommendations

The following recommendations can be made:

- Exploration of different pruning parameters with the current pruning strategy and also trying different pruning strategies to try and minimise performance loss when pruning the models.
- Training a model from scratch using quantization aware training.

- Combining a multi-label model with a binary model to create a two-stage classifier. The combination of Xie et al. (2019) binary model with the ResNet-101 or TResNet-L showed a performance increase (Haurum and Moeslund, 2021).
- Static quantization below 8-bit.
- Application of a compression technique not used in this thesis, namely knowledge distillation.
- Sewer asset managers should consider investing in compressed model technology to enable sewer inspection at the edge. This approach reduces reliance on cloud-based systems and has the potential to lower operational costs. Managers can start by implementing pilot programs to test the effectiveness of compressed models in field conditions and conducting a cost-benefit analysis to understand the financial implications and potential savings. Collaborating with technology providers will be crucial for integrating these models into existing inspection workflows and staying updated on advancements in model compression techniques.

Appendix A

Static quantization: preparation of the ResNet-101 architecture

```python

import torch
from torch import Tensor
import torch.nn as nn
import torch.quantization
from typing import Type, Any, Callable, Union, List, Optional
from torch.quantization import QuantStub, DeQuantStub

try:
    from torch.hub import load_state_dict_from_url
except ImportError:
    from torch.utils.model_zoo import load_url as
     ↪ load_state_dict_from_url


model_urls = {
     'resnet101':
      ↪ 'https://download.pytorch.org/models/resnet101-5d3b4d8f.pth'
}

def conv3x3(in_planes: int, out_planes: int, stride: int = 1, groups:
 ↪ int = 1, dilation: int = 1) -> nn.Conv2d:
    """3x3 convolution with padding"""
    return nn.Conv2d(in_planes, out_planes, kernel_size=3,
     ↪ stride=stride,
                     padding=dilation, groups=groups, bias=False,
                      ↪ dilation=dilation)

def conv1x1(in_planes: int, out_planes: int, stride: int = 1) ->
 ↪ nn.Conv2d:
    """1x1 convolution"""
    return nn.Conv2d(in_planes, out_planes, kernel_size=1,
     ↪ stride=stride, bias=False)

class BasicBlock(nn.Module):
    expansion: int = 1
```

```

    def __init__(
        self,
        inplanes: int,
        planes: int,
        stride: int = 1,
        downsample: Optional[nn.Module] = None,
        groups: int = 1,
        base_width: int = 64,
        dilation: int = 1,
        norm_layer: Optional[Callable[..., nn.Module]] = None
    ) -> None:
        super(BasicBlock, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        if groups != 1 or base_width != 64:
            raise ValueError('BasicBlock only supports groups=1 and
            ↪  base_width=64')
        if dilation > 1:
            raise NotImplementedError("Dilation > 1 not supported in
            ↪  BasicBlock")
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = norm_layer(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = norm_layer(planes)
        self.downsample = downsample
        self.stride = stride
        self.skip_add = nn.quantized.FloatFunctional()

    def forward(self, x: Tensor) -> Tensor:
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out = self.skip_add.add(out, identity)
        out = self.relu(out)

        return out

class Bottleneck(nn.Module):
    expansion: int = 4

```

```
    def __init__(
        self,
        inplanes: int,
        planes: int,
        stride: int = 1,
        downsample: Optional[nn.Module] = None,
        groups: int = 1,
        base_width: int = 64,
        dilation: int = 1,
        norm_layer: Optional[Callable[..., nn.Module]] = None
    ) -> None:
        super(Bottleneck, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        width = int(planes * (base_width / 64.)) * groups
        self.conv1 = conv1x1(inplanes, width)
        self.bn1 = norm_layer(width)
        self.conv2 = conv3x3(width, width, stride, groups, dilation)
        self.bn2 = norm_layer(width)
        self.conv3 = conv1x1(width, planes * self.expansion)
        self.bn3 = norm_layer(planes * self.expansion)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.stride = stride
        self.skip_add = nn.quantized.FloatFunctional()

    def forward(self, x: Tensor) -> Tensor:
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        if self.downsample is not None:
            identity = self.downsample(x)

        out = self.skip_add.add(out, identity)
        out = self.relu(out)

        return out


class ResNet(nn.Module):
    def __init__(
```

```
        self,
        block: Type[Union[BasicBlock, Bottleneck]],
        layers: List[int],
        num_classes: int = 1000,
        zero_init_residual: bool = False,
        groups: int = 1,
        width_per_group: int = 64,
        replace_stride_with_dilation: Optional[List[bool]] = None,
        norm_layer: Optional[Callable[..., nn.Module]] = None
    ) -> None:
        super(ResNet, self).__init__()
        if norm_layer is None:
            norm_layer = nn.BatchNorm2d
        self._norm_layer = norm_layer

        self.inplanes = 64
        self.dilation = 1
        if replace_stride_with_dilation is None:
            replace_stride_with_dilation = [False, False, False]
        if len(replace_stride_with_dilation) != 3:
            raise ValueError("replace_stride_with_dilation should be None "
                             "or a 3-element tuple, got {}".format(replace_stride_with_dilation))
        self.groups = groups
        self.base_width = width_per_group
        self.quant = torch.ao.quantization.QuantStub()
        self.conv1 = nn.Conv2d(3, self.inplanes, kernel_size=7, stride=2, padding=3,
                               bias=False)
        self.bn1 = norm_layer(self.inplanes)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2,
                                       dilate=replace_stride_with_dilation[0])
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2,
                                       dilate=replace_stride_with_dilation[1])
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2,
                                       dilate=replace_stride_with_dilation[2])
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * block.expansion, num_classes)
        self.dequant = torch.ao.quantization.DeQuantStub()
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
            elif isinstance(m, (nn.BatchNorm2d, nn.GroupNorm)):
```

```
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)



    def _make_layer(self, block: Type[Union[BasicBlock, Bottleneck]],
     ↪  planes: int, blocks: int,
                    stride: int = 1, dilate: bool = False) ->
                    ↪  nn.Sequential:
        norm_layer = self._norm_layer
        downsample = None
        previous_dilation = self.dilation
        if dilate:
            self.dilation *= stride
            stride = 1
        if stride != 1 or self.inplanes != planes * block.expansion:
            downsample = nn.Sequential(
                conv1x1(self.inplanes, planes * block.expansion,
                ↪  stride),
                norm_layer(planes * block.expansion),
            )

        layers = []
        layers.append(block(self.inplanes, planes, stride, downsample,
         ↪  self.groups,
                            self.base_width, previous_dilation,
                            ↪  norm_layer))
        self.inplanes = planes * block.expansion
        for _ in range(1, blocks):
            layers.append(block(self.inplanes, planes,
             ↪  groups=self.groups,
                                base_width=self.base_width,
                                ↪  dilation=self.dilation,
                                norm_layer=norm_layer))

        return nn.Sequential(*layers)

    def _forward_impl(self, x: Tensor) -> Tensor:
        # Quantize the input
        x = self.quant(x)

        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

```

```
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)

        # Dequantize the output
        x = self.dequant(x)

        return x

    def forward(self, x: Tensor) -> Tensor:
        return self._forward_impl(x)

def resnet101_quantizable(pretrained=False, progress=True, **kwargs):
    model = ResNet(Bottleneck, [3, 4, 23, 3], **kwargs)

    return model
```

Appendix B

Static quantization ResNet-101

```
from google.colab import drive
drive.mount('/content/drive')


%run /content/xxCHANGEDQR101TEST.py

import torch
import torchvision.models as models
import os
from PIL import Image
import csv
from torchvision import transforms
import time
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from torch.utils.data import DataLoader, Dataset
import numpy as np
import pandas as pd
import zipfile


model_path = '/content/drive/My Drive/Msc thesis/SewerML/Models/resnet101-e2e-version_1.pth'

Qmodel = resnet101_quantizable(num_classes=17, pretrained=False).to('cpu')

# Load the state dictionary from the .pth file
state_dict = torch.load(model_path)

# Load the state dictionary into the modified model
Qmodel.load_state_dict(state_dict['state_dict'], strict=False)

# Set the model to evaluation mode
Qmodel.eval()
```

```
def load_labels(path):
    img2label = {}
    with open(path, newline='') as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')

        header = next(csv_reader, None)

        for row in csv_reader:
            filename = row[0]
            labels = [int(val) for i, val in enumerate(row[1:]) if
             ↪  header[i + 1] not in ['WaterLevel', 'VA', 'ND',
             ↪  'Defect']]
            img2label[filename] = labels

    return img2label

def load_data(data_path, labels_path):
    images = []
    labels = []
    img2label = load_labels(labels_path)
    for filename in os.listdir(data_path):
        img_path = os.path.join(data_path, filename)
        img = Image.open(img_path).convert('RGB')
        if img is not None and filename in img2label:
            images.append(img)
            labels.append(img2label[filename])

    return images, labels

data_path = "/content/drive/My Drive/Msc
 ↪  thesis/SewerML/Data/Valduizend"
labels_path = "/content/SewerML_Val.csv"
test_data, test_labels = load_data(data_path, labels_path)

data_path = "/content/drive/My Drive/Msc thesis/SewerML/Data/Quant250"
labels_path = "/content/SewerML_Train.csv"
train_data, train_labels = load_data(data_path, labels_path)



mean_values = [0.523, 0.453, 0.345]
std_dev_values = [0.210, 0.199, 0.154]

def transform_data(data):
    for index, img in enumerate(data):
        preprocess = transforms.Compose([
            transforms.Resize(256),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            transforms.Normalize(mean=mean_values, std=std_dev_values),
        ])
```

```python
        input_tensor = preprocess(img)
        input_batch = input_tensor.unsqueeze(0)
        data[index] = input_batch
    return data

# Apply the transformation to the test_data and train_data
test_data = transform_data(test_data)
train_data = transform_data(train_data)



def evaluate(model, data, target, threshold=0.5):
    model.eval()
    total_time, correct = 0, 0

    with torch.no_grad():
        for img, target_labels in zip(data, target):
            start = time.time()
            output = model(img)
            end = time.time()
            delta = end - start
            total_time += delta

            # Apply sigmoid activation to the output to obtain class probabilities
            probabilities = torch.sigmoid(output).squeeze().tolist()

            # Convert probabilities to binary predictions based on a threshold
            pred_labels = [1 if p >= threshold else 0 for p in probabilities]

            # Compare predicted labels with ground truth labels
            if pred_labels == target_labels:
                correct += 1

    inference_time = total_time / len(data)
    accuracy = accuracy_score(target_labels, pred_labels)
    overall_f1 = f1_score(target_labels, pred_labels, average='weighted')

    return inference_time, accuracy, overall_f1

float_model = Qmodel.to('cpu')


inference_time, accuracy, overall_f1= evaluate(float_model, test_data, test_labels)

print("Baseline Inference Time: ", inference_time)
print("Baseline Accuracy: ", accuracy, '%')
```

```
print("Overall F1 Score: ", overall_f1)

## Initial baseline model which is FP32
model_fp32 = float_model
model_fp32.eval()

# Sets the backend for x86
model_fp32.qconfig = torch.quantization.get_default_qconfig('fbgemm')

# Prepares the model for calibration.
# Inserts observers in the model that will observe the activation tensors during calibration
model_fp32_prepared = torch.quantization.prepare(model_fp32, inplace = False)

# Calibrate over the train dataset. This determines the quantization params for activation.
evaluate(model_fp32_prepared, train_data, train_labels)

# Converts the model to a quantized model(int8)
model_quantized = torch.quantization.convert(model_fp32_prepared) # Quantize the model

# Evaluates the quantized model on the test dataset
inference_time, accuracy, overall_f1 = evaluate(model_quantized, test_data, test_labels)

print("Baseline Inference Time: ", inference_time)
print("Baseline Accuracy: ", accuracy, '%')
print("Overall F1 Score: ", overall_f1)

torch.save(model_quantized.state_dict(), '250TrainQuantized_model.pth')



# Instantiate your custom model
model = resnet101_quantizable(num_classes=17)

# Prepare the model for quantization if it's not already
model.qconfig = torch.quantization.get_default_qconfig('fbgemm')
torch.quantization.prepare(model, inplace=True)
torch.quantization.convert(model, inplace=True)

# Load the quantized state dictionary
model.load_state_dict(torch.load('/content/250TrainQuantized_model.pth'))

# Set the model to evaluation mode
model.eval()

# Import the validation dataset
```

```
!cp '/content/drive/My Drive/Msc thesis/SewerML/Data/valid00.zip'
↪ '/content/valid00.zip'

!cp '/content/drive/My Drive/Msc thesis/SewerML/Data/valid01.zip'
↪ '/content/valid01.zip'


# Define the path to your zip files
zip_files = ['valid00.zip', 'valid01.zip']  # Add more zip files if
↪ needed

# Define the destination folder within Colab
destination_folder = '/content/Validationset'

# Unzip each zip file
for zip_file in zip_files:
    with zipfile.ZipFile(zip_file, 'r') as zip_ref:
        zip_ref.extractall(destination_folder)

# Check if the images are successfully unzipped
unzipped_files = os.listdir(destination_folder)
print(f"Images successfully unzipped to: {destination_folder}")
print(f"Unzipped {len(unzipped_files)} files.")



# Define the path to the folder containing test images
test_image_folder = '/content/Validationset'

# Define mean and standard deviation values for normalization
mean_values = [0.523, 0.453, 0.345]
std_dev_values = [0.210, 0.199, 0.154]

# Create a transform to preprocess the images (resize and
↪ normalization)
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=mean_values, std=std_dev_values),
])

class ImageDataset(Dataset):
    def __init__(self, image_folder, transform=None):
        self.image_folder = image_folder
        self.transform = transform
        self.image_files = os.listdir(image_folder)

    def __len__(self):
        return len(self.image_files)

    def __getitem__(self, idx):
```

```python
        image_file = self.image_files[idx]
        image_path = os.path.join(self.image_folder, image_file)
        image = Image.open(image_path).convert("RGB")
        if self.transform:
            image = self.transform(image)
        return image, image_file

# Create the dataset and data loader
dataset = ImageDataset(test_image_folder, transform=transform)
batch_size = 32
data_loader = DataLoader(dataset, batch_size=batch_size, shuffle=False,
    num_workers=2)

# Set model to evaluation mode
model.eval()

all_predictions = []
all_filenames = []

# Perform batched inference
for images, filenames in data_loader:
    # Perform prediction
    with torch.no_grad():
        outputs = model(images)

    # Apply sigmoid activation to obtain probabilities
    probabilities = torch.sigmoid(outputs).cpu().numpy()

    all_predictions.extend(probabilities)
    all_filenames.extend(filenames)

# Convert predictions and filenames into numpy arrays
all_predictions = np.array(all_predictions)
all_filenames = np.array(all_filenames)

# Creation of DataFrame with the data
data = {
    'Filename': all_filenames,
}

# Definition of the classes
for i, category in enumerate(['RB', 'OB', 'PF', 'DE', 'FS', 'IS', 'RO',
    'IN', 'AF', 'BE', 'FO', 'GR', 'PH', 'PB', 'OS', 'OP', 'OK']):
    data[category] = all_predictions[:, i]

df = pd.DataFrame(data)

# Extract the numeric portion from the 'Filename' column for sorting
df['Numeric_Filename'] =
    df['Filename'].str.extract(r'(\d+)').astype(int)
```

```
# Sort the DataFrame by the 'Numeric_Filename' column and drop the
↪ helper column
df_sorted = df.sort_values(by='Numeric_Filename',
↪ ascending=True).drop(columns=['Numeric_Filename'])

# Save the sorted DataFrame to a CSV file
output_csv_path = '/content/drive/My Drive/Msc
↪ thesis/SewerML/Data/Predictions/ResNet101/Quant/Quant250Resnet_Entire_Val.csv'
df_sorted.to_csv(output_csv_path, index=False)

```

Appendix C

Layer fusion ResNet-101

```python
from google.colab import drive
drive.mount('/content/drive')


 %run /content/xxCHANGEDQR101TEST.py

import torch
import torch.nn as nn
import torch.ao.quantization
import os
from PIL import Image
import csv
import time
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import zipfile
from torchvision import transforms
from torch.utils.data import DataLoader, Dataset
import numpy as np
import pandas as pd

model_path = '/content/drive/My Drive/Msc thesis/SewerML/Models/resnet101-e2e-version_1.pth'

model = resnet101_quantizable(num_classes=17, pretrained=False).to('cpu')

# Load the state dictionary from the .pth file
state_dict = torch.load(model_path)

# Load the state dictionary into the modified model, excluding the final classification layer
model.load_state_dict(state_dict['state_dict'], strict=False)

# Set the model to evaluation mode
model.eval()

model_fp32=model
```

```python

model_fp32.qconfig = torch.quantization.get_default_qconfig('fbgemm')

model_fp32_fused = torch.ao.quantization.fuse_modules(model_fp32, [['conv1', 'bn1', 'relu']])


def fuse_bottleneck_layers(layer):
    for name, bottleneck_module in layer.named_children():
        # Fuse layers within the Bottleneck blocks
        torch.ao.quantization.fuse_modules(bottleneck_module, [['conv1', 'bn1'], ['conv2', 'bn2'], ['conv3', 'bn3']], inplace=True)

        # Check if the 'downsample' module exists in the Bottleneck block
        if hasattr(bottleneck_module, 'downsample') and isinstance(bottleneck_module.downsample, nn.Sequential):
            # Print the 'downsample' structure before fusion
            print(f"Before fusion in Bottleneck {name} downsample:")
            print(bottleneck_module.downsample)

            # Attempt to fuse Conv2d and BatchNorm2d layers within 'downsample'
            fused = torch.ao.quantization.fuse_modules(bottleneck_module.downsample, [['0', '1']], inplace=True)

            # Check if fusion was successful (fused is not None)
            if fused:
                print(f"After fusion in Bottleneck {name} downsample:")
                print(bottleneck_module.downsample)
            else:
                print(f"No fusion applied in Bottleneck {name} downsample.")

# Assuming 'model_fp32_fused' is your model prepared for layer fusion
# Apply the fusion process to layers 1, 2, 3, and 4
fuse_bottleneck_layers(model_fp32_fused.layer1)
fuse_bottleneck_layers(model_fp32_fused.layer2)
fuse_bottleneck_layers(model_fp32_fused.layer3)
fuse_bottleneck_layers(model_fp32_fused.layer4)

print(model_fp32_fused)



def load_labels(path):
    img2label = {}
    with open(path, newline='') as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
```

```

        # Skip the header row
        header = next(csv_reader, None)

        for row in csv_reader:
            filename = row[0]
            labels = [int(val) for i, val in enumerate(row[1:]) if
             ↪  header[i + 1] not in ['WaterLevel', 'VA', 'ND',
             ↪  'Defect']]
            img2label[filename] = labels

    return img2label

def load_data(data_path, labels_path):
    images = []
    labels = []
    img2label = load_labels(labels_path)
    for filename in os.listdir(data_path):
        img_path = os.path.join(data_path, filename)
        img = Image.open(img_path).convert('RGB')
        if img is not None and filename in img2label:
            images.append(img)
            labels.append(img2label[filename])

    return images, labels

data_path = "/content/drive/My Drive/Msc
 ↪  thesis/SewerML/Data/Valduizend"
labels_path = "/content/SewerML_Val.csv"
test_data, test_labels = load_data(data_path, labels_path)

data_path = "/content/drive/My Drive/Msc thesis/SewerML/Data/Quant250"
labels_path = "/content/SewerML_Train.csv"
train_data, train_labels = load_data(data_path, labels_path)

from torchvision import transforms

mean_values = [0.523, 0.453, 0.345]
std_dev_values = [0.210, 0.199, 0.154]

def transform_data(data):
    for index, img in enumerate(data):
        preprocess = transforms.Compose([
            transforms.Resize(256),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            transforms.Normalize(mean=mean_values, std=std_dev_values),
        ])
        input_tensor = preprocess(img)
        input_batch = input_tensor.unsqueeze(0)
        data[index] = input_batch
```

```
    return data

# Apply the transformation to the test_data and train_data
test_data = transform_data(test_data)
train_data = transform_data(train_data)



def evaluate(model, data, target, threshold=0.5):
    model.eval()
    total_time, correct = 0, 0

    with torch.no_grad():
        for img, target_labels in zip(data, target):
            start = time.time()
            output = model(img)
            end = time.time()
            delta = end - start
            total_time += delta

            # Apply sigmoid activation to the output to obtain class probabilities
            probabilities = torch.sigmoid(output).squeeze().tolist()

            # Convert probabilities to binary predictions based on a threshold
            pred_labels = [1 if p >= threshold else 0 for p in probabilities]

            # Compare predicted labels with ground truth labels
            if pred_labels == target_labels:
                correct += 1

    inference_time = total_time / len(data)
    accuracy = accuracy_score(target_labels, pred_labels)
    overall_f1 = f1_score(target_labels, pred_labels, average='weighted')

    return inference_time, accuracy, overall_f1

model_fp32_prepared = torch.quantization.prepare(model_fp32_fused, inplace=False)

evaluate(model_fp32_prepared, train_data, train_labels)

model_int8 = torch.quantization.convert(model_fp32_prepared, inplace=False)

torch.save(model_int8.state_dict(), '250FullyQuantFuse_250FineTune_model.pth')

```

```
# Import the validation dataset
!cp '/content/drive/My Drive/Msc thesis/SewerML/Data/valid00.zip'
↪ '/content/valid00.zip'

!cp '/content/drive/My Drive/Msc thesis/SewerML/Data/valid01.zip'
↪ '/content/valid01.zip'



# Define the path to your zip files
zip_files = ['valid00.zip', 'valid01.zip']

# Define the destination folder within Colab
destination_folder = '/content/Validationset'

# Unzip each zip file
for zip_file in zip_files:
    with zipfile.ZipFile(zip_file, 'r') as zip_ref:
        zip_ref.extractall(destination_folder)

# Check if the images are successfully unzipped
unzipped_files = os.listdir(destination_folder)
print(f"Images successfully unzipped to: {destination_folder}")
print(f"Unzipped {len(unzipped_files)} files.")


# Loading the quantized model with the fused layers
model_fp32_fused = resnet101_quantizable(num_classes=17,
↪ pretrained=False).to('cpu')
model_fp32_fused.eval()

model_fp32_fused.qconfig =
↪ torch.quantization.get_default_qconfig('fbgemm')
model_fp32_fused = torch.ao.quantization.fuse_modules(model_fp32_fused,
↪ [['conv1', 'bn1', 'relu']])



def fuse_bottleneck_layers(layer):
    for name, bottleneck_module in layer.named_children():
        # Fuse layers within the Bottleneck blocks as before
        torch.ao.quantization.fuse_modules(bottleneck_module,
        ↪ [['conv1', 'bn1'], ['conv2', 'bn2'], ['conv3', 'bn3']],
        ↪ inplace=True)

        # Check if the 'downsample' module exists in the Bottleneck
        ↪ block
        if hasattr(bottleneck_module, 'downsample') and
        ↪ isinstance(bottleneck_module.downsample, nn.Sequential):
            # Print the 'downsample' structure before fusion
            print(f"Before fusion in Bottleneck {name} downsample:")
```

```
            print(bottleneck_module.downsample)

            # Attempt to fuse Conv2d and BatchNorm2d layers within 'downsample'
            fused = torch.ao.quantization.fuse_modules(bottleneck_module.downsample, [['0', '1']], inplace=True)

            # Check if fusion was successful (fused is not None)
            if fused:
                print(f"After fusion in Bottleneck {name} downsample:")
                print(bottleneck_module.downsample)
            else:
                print(f"No fusion applied in Bottleneck {name} downsample.")

# Assuming 'model_fp32_fused' is your model prepared for layer fusion
# Apply the fusion process to layers 1, 2, 3, and 4
fuse_bottleneck_layers(model_fp32_fused.layer1)
fuse_bottleneck_layers(model_fp32_fused.layer2)
fuse_bottleneck_layers(model_fp32_fused.layer3)
fuse_bottleneck_layers(model_fp32_fused.layer4)

torch.quantization.prepare(model_fp32_fused, inplace=True)
torch.quantization.convert(model_fp32_fused, inplace=True)

model_int8_fused = model_fp32_fused

model_int8_fused.load_state_dict(torch.load('/content/250FullyQuantFuse_250FineTune_moc

# Set the model to evaluation mode
model_int8_fused.eval()


# Define the path to the folder containing test images
test_image_folder = '/content/Validationset'

# Define mean and standard deviation values for normalization
mean_values = [0.523, 0.453, 0.345]
std_dev_values = [0.210, 0.199, 0.154]

# Create a transform to preprocess the images (resize and normalization)
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=mean_values, std=std_dev_values),
])

class ImageDataset(Dataset):
    def __init__(self, image_folder, transform=None):
```

```
        self.image_folder = image_folder
        self.transform = transform
        self.image_files = os.listdir(image_folder)

    def __len__(self):
        return len(self.image_files)

    def __getitem__(self, idx):
        image_file = self.image_files[idx]
        image_path = os.path.join(self.image_folder, image_file)
        image = Image.open(image_path).convert("RGB")
        if self.transform:
            image = self.transform(image)
        return image, image_file

# Create the dataset and data loader
dataset = ImageDataset(test_image_folder, transform=transform)
batch_size = 32  # You can adjust the batch size
data_loader = DataLoader(dataset, batch_size=batch_size, shuffle=False,
↪ num_workers=4)

# Ensure the model is in evaluation mode and on CPU
model_int8_fused.eval()

all_predictions = []
all_filenames = []

# Perform batched inference
for images, filenames in data_loader:
    # Perform prediction
    with torch.no_grad():
        outputs = model_int8_fused(images)

    # Apply softmax to obtain probabilities
    probabilities = torch.sigmoid(outputs).cpu().numpy()

    all_predictions.extend(probabilities)
    all_filenames.extend(filenames)

# Convert predictions and filenames into numpy arrays (if not already
↪ in this format)
all_predictions = np.array(all_predictions)
all_filenames = np.array(all_filenames)

# Create a DataFrame with the data
data = {
    'Filename': all_filenames,
    # Add your prediction categories here
}
```

```
for i, category in enumerate(['RB', 'OB', 'PF', 'DE', 'FS', 'IS', 'RO', 'IN', 'AF', 'BE', 'FO', 'GR', 'PH', 'PB', 'OS', 'OP', 'OK']):
    data[category] = all_predictions[:, i]

df = pd.DataFrame(data)

# Extract the numeric portion from the 'Filename' column for sorting
df['Numeric_Filename'] = df['Filename'].str.extract(r'(\d+)').astype(int)

# Sort the DataFrame by the 'Numeric_Filename' column and drop the helper column
df_sorted = df.sort_values(by='Numeric_Filename', ascending=True).drop(columns=['Numeric_Filename'])

# Save the sorted DataFrame to a CSV file
output_csv_path = '/content/drive/My Drive/Msc thesis/SewerML/Data/Predictions/ResNet101/QuantFuse/250FullyQuantFuse_250FineTune_F
df_sorted.to_csv(output_csv_path, index=False)
```

Appendix D

ResNet-101 iterative pruning

```
from google.colab import drive
drive.mount('/content/drive')

import torch
import torchvision.models as models
import torch.nn.utils.prune as prune
import zipfile
import os
from PIL import Image
import csv
import os
from torchvision import transforms
from torch.utils.data import DataLoader, Dataset
import numpy as np
import pandas as pd
import torch.optim as optim
import torch.nn as nn
import torch.nn.functional as F

model_path = '/content/drive/My Drive/Msc
↪ thesis/SewerML/Models/resnet101-e2e-version_1.pth'

model = resnet101_quantizable(num_classes=17,
↪ pretrained=False).to('cpu')

# Load the state dictionary from the .pth file
state_dict = torch.load(model_path)

# Load the state dictionary into the modified model, excluding the
↪ final classification layer
model.load_state_dict(state_dict['state_dict'], strict=False)

# Set the model to evaluation mode
model.eval()

# Importing the validation dataset
!cp '/content/drive/My Drive/Msc thesis/SewerML/Data/valid00.zip'
↪ '/content/valid00.zip'
```

```
!cp '/content/drive/My Drive/Msc thesis/SewerML/Data/valid01.zip'
 ↪ '/content/valid01.zip'



# Define the path to your zip files
zip_files = ['valid00.zip', 'valid01.zip']  # Add more zip files if
 ↪ needed

# Define the destination folder within Colab
destination_folder = '/content/Validationset'

# Unzip each zip file
for zip_file in zip_files:
    with zipfile.ZipFile(zip_file, 'r') as zip_ref:
        zip_ref.extractall(destination_folder)

# Check if the images are successfully unzipped
unzipped_files = os.listdir(destination_folder)
print(f"Images successfully unzipped to: {destination_folder}")
print(f"Unzipped {len(unzipped_files)} files.")



class MultiLabelDataset(Dataset):
    def __init__(self, data_path, labels_path, transform=None):
        self.data_path = data_path
        self.transform = transform
        self.img_labels = self.load_labels(labels_path)

    def load_labels(self, labels_path):
        img2labels = {}
        available_files = set(os.listdir(self.data_path))
        excluded_columns = ['WaterLevel', 'VA', 'ND', 'Defect']

        with open(labels_path, newline='') as csv_file:
            csv_reader = csv.reader(csv_file, delimiter=',')
            header = next(csv_reader)  # Header row with column names

            # Determine the indices of columns to exclude
            excluded_indices = [header.index(col) for col in
             ↪ excluded_columns if col in header]

            for row in csv_reader:
                filename = row[0]
                if filename in available_files:
                    # Include only columns not in excluded_indices
                    labels = torch.tensor([int(row[i]) for i in
                     ↪ range(1, len(row)) if i not in
                     ↪ excluded_indices], dtype=torch.float32)
```

```
                    img2labels[filename] = labels

        return img2labels

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_name = list(self.img_labels.keys())[idx]
        img_path = os.path.join(self.data_path, img_name)
        image = Image.open(img_path).convert('RGB')
        label = self.img_labels[img_name]
        if self.transform:
            image = self.transform(image)
        return image, label

from torch.utils.data import DataLoader


# Define transformations
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
    ↪ 0.225]),
])

# Create the dataset and data loader
train_dataset = MultiLabelDataset(
    data_path="/content/drive/My Drive/Msc
    ↪ thesis/SewerML/Data/Quant10k",
    labels_path="/content/SewerML_Train.csv",
    transform=transform
)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True,
↪ num_workers=4)

device = torch.device("cuda")
model.to(device)



def prune_model(model, amount):
    for name, module in model.named_modules():
        if isinstance(module, (torch.nn.Conv2d, torch.nn.Linear,
        ↪ torch.nn.BatchNorm2d)):
            # Save the original state of the module for comparison
            original_weight = None
            original_bias = None
            if hasattr(module, 'weight'):
```

```python
                original_weight = module.weight.detach().clone()
            if hasattr(module, 'bias') and module.bias is not None:
                original_bias = module.bias.detach().clone()

            # Count nonzero elements before pruning
            original_nonzeros = torch.count_nonzero(original_weight).item() if original_weight is not None else 0
            original_bias_nonzeros = torch.count_nonzero(original_bias).item() if original_bias is not None else 0

            # Apply pruning
            prune.l1_unstructured(module, name='weight', amount=amount)
            if module.bias is not None:
                prune.l1_unstructured(module, name='bias', amount=amount)

            # Count nonzero elements after pruning
            pruned_nonzeros = torch.count_nonzero(module.weight).item()
            pruned_bias_nonzeros = torch.count_nonzero(module.bias).item() if hasattr(module, 'bias') and module.bias is not None else 0

            # Calculate and display the percentage change for weights
            percentage_change_weights = 100 * (1 - pruned_nonzeros / original_nonzeros) if original_nonzeros > 0 else 0
            print(f'Pruning {name}... Weight nonzeros: {original_nonzeros} -> {pruned_nonzeros} ({percentage_change_weights:.2f}% change)')

            # Calculate and display the percentage change for bias, if applicable
            if original_bias is not None:
                percentage_change_bias = 100 * (1 - pruned_bias_nonzeros / original_bias_nonzeros) if original_bias_nonzeros > 0 else 0
                print(f'  Bias nonzeros: {original_bias_nonzeros} -> {pruned_bias_nonzeros} ({percentage_change_bias:.2f}% change)')



# Function to apply the mask hook
def apply_mask_hook(module, input):
    if hasattr(module, 'weight_mask'):
        module.weight.data.mul_(module.weight_mask)
    if hasattr(module, 'bias_mask') and module.bias is not None:
        module.bias.data.mul_(module.bias_mask)

```

```python
# Function to fine-tune the model
def fine_tune_model(model, train_loader, num_epochs=10):
    # Register the hook for each pruned layer
    for name, module in model.named_modules():
        if isinstance(module, (nn.Conv2d, nn.Linear, nn.BatchNorm2d)):
            if hasattr(module, 'weight_mask') or (hasattr(module, 'bias_mask') and module.bias is not None):
                module.register_forward_pre_hook(apply_mask_hook)

    # Define the optimizer and loss function
    optimizer = optim.Adam(model.parameters(), lr=1e-4)
    criterion = nn.BCEWithLogitsLoss()

    # Fine-tuning loop
    model.train()
    for epoch in range(num_epochs):
        total_loss = 0
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)

            optimizer.zero_grad(set_to_none=True)
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            total_loss += loss.item()

        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {total_loss / len(train_loader):.4f}")



def remove_pruning_reparameterization(model):
    for module in model.modules():
        if isinstance(module, (nn.Conv2d, nn.Linear, nn.BatchNorm2d)):
            prune.remove(module, 'weight')
            if module.bias is not None:
                prune.remove(module, 'bias')


def validate_and_save_predictions(model, iteration, predictions_save_dir):
    test_image_folder = '/content/Validationset'

    # Define mean and standard deviation values for normalization
    mean_values = [0.523, 0.453, 0.345]
    std_dev_values = [0.210, 0.199, 0.154]

    # Create a transform to preprocess the images (resize and normalization)
```

```
    transform = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=mean_values, std=std_dev_values),
    ])

    class ImageDataset(Dataset):
        def __init__(self, image_folder, transform=None):
            self.image_folder = image_folder
            self.transform = transform
            self.image_files = os.listdir(image_folder)

        def __len__(self):
            return len(self.image_files)

        def __getitem__(self, idx):
            image_file = self.image_files[idx]
            image_path = os.path.join(self.image_folder, image_file)
            image = Image.open(image_path).convert("RGB")
            if self.transform:
                image = self.transform(image)
            return image, image_file

    # Create the dataset and data loader
    dataset = ImageDataset(test_image_folder, transform=transform)
    batch_size = 32  # Adjust the batch size as needed
    data_loader = DataLoader(dataset, batch_size=batch_size,
    ↪ shuffle=False, num_workers=4)

    # Move the model to the CUDA device
    device = torch.device("cuda" if torch.cuda.is_available() else
    ↪ "cpu")
    model.to(device)
    model.eval()

    all_filenames = []
    all_predictions = []

    for images, filenames in data_loader:
        images = images.to(device)
        with torch.no_grad():
            outputs = model(images)
            probabilities = torch.sigmoid(outputs).cpu().numpy()  #
            ↪ Adjust for your model's output
        all_filenames.extend(filenames)
        all_predictions.extend(probabilities)

    all_predictions_array = np.vstack(all_predictions)

    data = {'Filename': all_filenames}
```

```python
    for i, category in enumerate(['RB', 'OB', 'PF', 'DE', 'FS', 'IS', 'RO', 'IN', 'AF', 'BE', 'FO', 'GR', 'PH', 'PB', 'OS', 'OP', 'OK']):
        data[category] = all_predictions_array[:, i]

    df = pd.DataFrame(data)

    # Extract the numeric portion from the 'Filename' column for sorting
    df['Numeric_Filename'] = df['Filename'].str.extract(r'(\d+)').astype(int)

    # Sort the DataFrame by the 'Numeric_Filename' column and drop the helper column
    df_sorted = df.sort_values(by='Numeric_Filename', ascending=True).drop(columns=['Numeric_Filename'])

    # Save the sorted DataFrame to a CSV file
    output_csv_path = os.path.join(predictions_save_dir, f'predictions_iteration_{iteration}.csv')
    df_sorted.to_csv(output_csv_path, index=False)





# Define the directory for saving models and predictions
model_save_dir = '/content/drive/My Drive/Msc thesis/SewerML/Models/Pruned Resnet101'
predictions_save_dir = '/content/drive/My Drive/Msc thesis/SewerML/Data/Predictions/ResNet101/Pruned'

# Define the number of pruning iterations
num_iterations = 10

for iteration in range(1, num_iterations + 1):
    # Step 1: Pruning
    print(f"--- Iteration {iteration} ---")
    print("Pruning...")
    prune_model(model, amount=0.1 *iteration)  # Increase pruning amount in each iteration

    # Step 2: Fine-tuning
    print("Fine-tuning...")
    fine_tune_model(model, train_loader)

    # Step 3: Make pruning permanent
    print("Making pruning permanent...")
    remove_pruning_reparameterization(model)

    # Step 4: Save the model
```

```
    model_save_path = os.path.join(model_save_dir,
     ↪  f'pruned_resnet101_iteration_{iteration}.pth')
    torch.save(model.state_dict(), model_save_path)
    print(f"Model saved to {model_save_path}")

    # Step 5 & 6: Validate and save predictions
    print("Validating and saving predictions...")
    validate_and_save_predictions(model, iteration,
     ↪  predictions_save_dir)

print("Iterative pruning completed.")
```

Appendix E

Performance metrics ResNet-101 models

Model	$F2_{CIW}$	$F1_{Normal}$
Standard ResNet-101	53.26	79.55
Quantized	51.92	78.98
Quantized & fusion	51.70	78.81
Pruned 10%	38.56	86.28
Pruned 30%	39.23	87.30
Pruned 30%, quantized & fusion	40.09	87.41

TABLE E.1: Overall $F2_{CIW}$ and $F1_{Normal}$ scores for all variants of the ResNet-101 model.

Model	RB	OB	PF	DE	FS	IS	RO	IN	AF	BE	FO	GR	PH	PB	OS	OP	OK
Standard ResNet-101	42.45	84.34	51.08	35.34	87.49	19.98	37.81	47.47	59.18	59.87	10.39	64.78	61.24	44.03	34.81	54.23	71.60
Quantized	41.26	84.10	49.56	34.47	87.53	18.73	35.05	46.91	57.95	59.30	9.59	63.26	60.13	42.15	31.68	51.54	82.87
Quantized & fusion	41.68	84.20	48.65	33.74	87.47	18.84	36.05	45.86	58.29	58.97	9.64	63.18	59.42	41.14	31.08	50.67	82.89
Pruned 10%	16.94	58.85	49.80	21.15	64.48	9.56	25.21	25.36	23.63	27.00	21.24	51.18	43.10	50.05	30.38	51.37	62.23
Pruned 30%	12.43	60.27	49.26	24.76	73.13	12.27	28.59	31.74	23.75	28.74	19.73	58.12	41.84	47.24	30.26	49.20	63.30
Pruned 30%, quantized & fusion	13.34	62.18	50.54	24.73	73.21	12.28	30.70	33.09	23.89	29.74	19.47	57.78	42.92	47.25	29.84	52.43	90.11

TABLE E.2: Class $F2$ scores for all variants of the ResNet-101 model.

Model	RB	OB	PF	DE	FS	IS	RO	IN	AF	BE	FO	GR	PH	PB	OS	OP	OK
Standard ResNet-101	13.99	59.82	18.21	10.29	68.89	5.05	11.56	16.44	25.88	26.96	2.32	32.76	26.02	14.23	10.04	19.80	97.61
Quantized	13.13	58.15	17.21	9.89	67.89	4.65	10.15	16.07	24.30	26.23	2.11	30.21	24.90	13.22	8.73	18.03	60.09
Quantized & fusion	13.35	58.95	16.68	9.60	68.46	4.68	10.63	15.41	24.76	25.77	2.12	30.03	24.25	12.69	8.50	17.51	60.07
Pruned 10%	40.48	74.15	83.70	67.15	83.84	50.36	69.82	50.68	66.33	71.01	17.90	82.11	82.16	51.07	58.22	72.52	79.75
Pruned 30%	34.88	73.50	80.21	59.95	78.35	30.72	66.48	45.07	63.15	66.77	22.75	74.25	81.63	53.59	56.11	73.74	82.75
Pruned 30%, quantized & fusion	33.01	72.59	78.43	59.14	78.31	30.92	62.87	43.74	62.40	64.63	20.32	74.49	81.06	52.93	52.79	72.64	83.26

TABLE E.3: Class precision scores for all variants of the ResNet-101 model.

Model	RB	OB	PF	DE	FS	IS	RO	IN	AF	BE	FO	GR	PH	PB	OS	OP	OK
Standard ResNet101	86.38	93.96	93.07	90.28	93.82	76.50	87.42	89.83	87.25	86.16	80.90	85.73	92.57	92.42	90.81	95.92	67.13
Quantized	88.82	94.66	93.47	90.97	94.36	77.30	90.54	90.15	88.64	86.59	83.25	87.08	93.04	93.07	92.34	96.24	66.25
Quantized & fusion	88.75	94.30	93.42	90.92	94.00	77.30	89.58	90.61	88.12	87.01	83.08	87.26	93.24	93.59	92.56	96.24	66.07
Pruned 10%	14.79	55.96	45.23	18.06	60.96	7.95	21.73	22.55	20.36	23.38	22.28	46.77	38.52	49.80	27.13	47.88	93.98
Pruned 30%	10.71	57.68	44.93	21.59	71.93	10.67	25.03	29.55	20.54	25.16	19.10	55.13	37.30	45.88	27.13	45.42	92.38
Pruned 30%, quantized & fusion	11.61	60.03	46.41	21.59	72.04	10.67	27.22	31.19	20.70	26.21	19.26	54.71	38.40	46.01	26.91	49.02	92.00

TABLE E.4: Class recall scores for all variants of the ResNet-101 model.

Appendix F

Predicted class count

Model	Images with No Defects Predicted
Standard ResNet-101	47,234
Quantized	46,545
Quantized & Fusion	46,470
Pruned 10%	80,929
Pruned 30%	76,672
Pruned 30%, Quantized & Fusion	75,886

TABLE F.2: Number of images with no defects predicted by all variants of the ResNet-101 model.

Class	Groundtruth Counts	ResNet-101	Quantized		Quantized & Fusion	
		Counts	Counts	% Chg	Counts	% Chg
RB	5538	34198	37454	+9.52%	36809	+7.63%
OB	23624	37110	38457	+3.63%	37788	+1.83%
PF	2021	10328	10973	+6.25%	11319	+9.60%
DE	2038	17884	18739	+4.78%	19306	+7.95%
FS	36218	49324	50343	+2.07%	49726	+0.82%
IS	881	13339	14659	+9.90%	14550	+9.08%
RO	2917	22050	26008	+17.95%	24572	+11.44%
IN	2812	15361	15770	+2.66%	16534	+7.64%
AF	9059	30539	33049	+8.22%	32239	+5.57%
BE	7929	25339	26179	+3.32%	26776	+5.67%
FO	597	20858	23521	+12.77%	23349	+11.94%
GR	6889	18026	19857	+10.16%	20014	+11.03%
PH	3432	12211	12824	+5.02%	13198	+8.08%
PB	765	4968	5387	+8.43%	5642	+13.57%
OS	457	4133	4833	+16.94%	4978	+20.45%
OP	612	2964	3266	+10.19%	3364	+13.50%
OK	19655	28883	29944	+3.67%	29969	+3.76%

Class	Pruned 10%		Pruned 30%		Pruned 30%, Quantized & Fusion	
	Counts	% Chg	Counts	% Chg	Counts	% Chg
RB	2023	-94.08%	1700	-95.03%	2035	-94.05%
OB	17828	-51.96%	18540	-50.04%	18405	-50.40%
PF	1092	-89.43%	1132	-89.04%	1161	-88.76%
DE	548	-96.94%	734	-95.90%	798	-95.54%
FS	26333	-46.61%	33249	-32.59%	34206	-30.65%
IS	139	-98.96%	306	-97.71%	323	-97.58%
RO	908	-95.88%	1098	-95.02%	1251	-94.33%
IN	1251	-91.86%	1844	-88.00%	1907	-87.59%
AF	2780	-90.90%	2947	-90.35%	3107	-89.83%
BE	2611	-89.70%	2988	-88.21%	2899	-88.56%
FO	743	-96.44%	501	-97.60%	600	-97.12%
GR	3924	-78.23%	5115	-71.62%	5189	-71.21%
PH	1609	-86.82%	1568	-87.16%	1586	-87.01%
PB	746	-84.98%	655	-86.82%	667	-86.57%
OS	213	-94.85%	221	-94.65%	213	-94.85%
OP	404	-86.37%	377	-87.28%	409	-86.20%
OK	13934	-51.76%	14747	-48.94%	15037	-47.94%

TABLE F.1: Model prediction counts and percentage changes relative to the standard ResNet-101 model.

Appendix G

Performance metrics TResNet-L models

Class	Groundtruth Counts	Standard TResNet-L	Pruned 10%		Pruned 80%	
		Counts	Counts	% Chg	Counts	% Chg
RB	5538	33790	2256	-93.32%	730	-97.84%
OB	23624	37130	19292	-48.04%	17336	-53.31%
PF	2021	10321	1455	-85.90%	1366	-86.76%
DE	2038	17080	563	-96.70%	750	-95.61%
FS	36218	48939	29878	-38.95%	30897	-36.87%
IS	881	10562	277	-97.38%	130	-98.77%
RO	2917	23299	752	-96.77%	1330	-94.29%
IN	2812	14107	1947	-86.20%	1596	-88.69%
AF	9059	28919	5361	-81.46%	2847	-90.16%
BE	7929	25499	4816	-81.11%	2976	-88.33%
FO	597	20339	245	-98.80%	202	-99.01%
GR	6889	16709	4740	-71.63%	4614	-72.39%
PH	3432	9960	1702	-82.91%	1694	-82.99%
PB	765	4320	561	-87.01%	502	-88.38%
OS	457	4196	225	-94.64%	238	-94.33%
OP	612	2351	412	-82.48%	377	-83.96%
OK	19655	29762	14666	-50.72%	17549	-41.04%

TABLE G.1: Model prediction counts and percentage changes relative to the standard TResNet-L model.

Model	Images with No Defects Predicted
Standard TResNet-L	48,686
10% Pruned	77,837
80% Pruned	77,543

TABLE G.2: Number of images with no defects predicted by selected variants of the TResNet-L model.

Model	RB	OB	PF	DE	FS	IS	RO	IN	AF	BE	FO	GR	PH	PB	OS	OP	OK
Standard TResNet-L	42.45	84.34	51.08	35.34	87.49	19.98	37.81	47.47	59.18	59.87	10.39	64.78	61.24	44.03	34.81	54.23	71.60
10% Pruned TResNet-L	18.38	64.05	59.55	23.47	70.82	15.65	23.75	35.16	36.70	40.81	18.42	59.60	48.02	45.43	34.34	54.37	91.53
80% Pruned TResNet-L	14.53	53.67	48.22	19.95	68.47	11.97	21.33	28.49	32.85	35.09	14.77	53.64	41.83	39.58	28.12	50.24	88.32

TABLE G.3: Per-class $F2$ scores for the TResNet-L model variants.

Model	RB	OB	PF	DE	FS	IS	RO	IN	AF	BE	FO	GR	PH	PB	OS	OP	OK
Standard TResNet-L	40.98	82.64	49.85	34.19	86.77	18.91	36.73	46.55	58.20	58.92	9.98	63.71	60.28	43.27	33.84	53.21	70.53
10% Pruned TResNet-L	39.76	75.56	78.08	72.65	82.84	42.96	78.46	47.66	56.95	61.92	39.59	81.22	87.07	58.65	62.67	75.49	83.58
80% Pruned TResNet-L	36.89	69.83	74.21	67.48	80.91	39.20	74.56	42.85	53.41	56.13	36.28	78.15	84.34	55.60	60.28	72.98	81.45

TABLE G.4: Per-class precision scores for the TResNet-L model variants.

Model	RB	OB	PF	DE	FS	IS	RO	IN	AF	BE	FO	GR	PH	PB	OS	OP	OK
Standard TResNet-L	41.07	83.21	50.17	35.70	87.12	19.30	38.01	47.80	59.50	59.51	10.87	64.22	61.53	44.78	35.20	54.45	71.85
10% Pruned TResNet-L	25.43	64.88	56.42	33.85	71.12	17.03	35.70	46.72	55.42	56.38	21.90	62.57	59.88	44.02	34.99	52.88	90.19
80% Pruned TResNet-L	21.98	57.65	52.33	30.45	69.03	14.77	33.97	43.84	52.28	53.14	19.53	58.74	57.18	41.47	33.22	51.62	87.99

TABLE G.5: Per-class recall scores for the TResNet-L model variants.

Bibliography

Alwani, Manoj, Han Chen, Michael Ferdman, and Peter Milder (2016). "Fused-layer CNN accelerators". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12. DOI: 10.1109/MICRO.2016.7783725.

Caradot, N., P. Rouault, F. Clemens, and F. Cherqui (2018). "Evaluation of uncertainties in sewer condition assessment". In: *Structure and Infrastructure Engineering* 14.2, pp. 264–273.

Chen, J. and X. Ran (2019). "Deep Learning With Edge Computing: A Review". In: *Proceedings of the IEEE* 107.8, pp. 1655–1674.

Chen, Kefan, Hong Hu, Chaozhan Chen, Long Chen, and Caiying He (2018). "An Intelligent Sewer Defect Detection Method Based on Convolutional Neural Network". In: *2018 IEEE International Conference on Information and Automation (ICIA)*, pp. 1301–1306. DOI: 10.1109/ICInfA.2018.8812445.

Choudhary, T., V. Mishra, A. Goswami, and J. Sarangapani (2020). "A comprehensive survey on model compression and acceleration". In: *Artifcial Intelligence Review* 53, 5113–5155.

Dirksen, J., F. H.L.R. Clemens, H. Korving, F. Cherqui, P. Le Gauffre, T. Ertl, H. Plihal, K. Müller, and C. T.M. Snaterse (2013). "The consistency of visual sewer inspection data". In: *Structure and Infrastructure Engineering* 9.3, pp. 214–228.

Du, Wei, Jianying Yu, Yi Gu, Ying Liu, Xiaobin Han, and Quantao Liu (Mar. 2019). "Preparation and application of microcapsules containing toluene-di-isocyanate for self-healing of concrete". In: *Construction and Building Materials* 202, pp. 762–769.

Dumoulin, Vincent and Francesco Visin (2016). "A guide to convolution arithmetic for deep learning". In: *ArXiv e-prints*. eprint: 1603.07285.

Elachachi, S.M., D. Breysse, and E. Vasconcelos (2006). "UNCERTAINTIES, QUALITY OF INFORMATION AND EFFICIENCY IN MANAGEMENT OF SEWER ASSETS". In: *Outcomes of the Joint International Conference on Computing and Decision Making in Civil and Building Engineering*. Joint International Conference on Computing, Decision Making in Civil, and Building Engineering. Montreal, Canada: Springer, pp. 2399–2408.

Fenner, R.A. (2000). "Approaches to sewer maintenance: a review". In: *Urban Water* 2, pp. 343–356.

Frankle, Jonathan and Michael Carbin (Mar. 2019). "The lottery ticket hypothesis: finding sparse, trainable neural networks." In: *arXiv (Cornell University)*. URL: https://arxiv.org/pdf/1803.03635v5.

Geld - Riool en raad (Oct. 2023). URL: https://www.rioolenraad.nl/benodigdheden/geld.

GfG (June 2023). *Artificial Neural Networks and its Applications*. URL: https://www.geeksforgeeks.org/artificial-neural-networks-and-its-applications/.

Gholami, Amir, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer (Jan. 2022). *A Survey of Quantization Methods for Efficient Neural Network Inference*, pp. 291–326. DOI: 10.1201/9781003162810-13. URL: https://doi.org/10.1201/9781003162810-13.

Gillis, Alexander S., Ed Burns, and Kate Brush (July 2023). *deep learning*. URL: https://www.techtarget.com/searchenterpriseai/definition/deep-learning-deep-neural-network.

Goodfellow, Ian J., Yoshua Bengio, and Aaron Courville (Nov. 2016). *Deep learning*. URL: https://dl.acm.org/citation.cfm?id=3086952.

Gou, Jianping, Baosheng Yu, Stephen J. Maybank, and Dacheng Tao (Mar. 2021). "Knowledge distillation: a survey". In: *International Journal of Computer Vision* 129.6, pp. 1789–1819. DOI: 10.1007/s11263-021-01453-z. URL: https://doi.org/10.1007/s11263-021-01453-z.

Goyal, Priya, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He (2018). *Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour*. arXiv: 1706.02677 [cs.CV].

Géron, Aurélien (Mar. 2017). *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. URL: http://cds.cern.ch/record/2699693.

Hahn, M.A., R.N. Palmer, M.S. Merrill, and A.B. Lukas (2002). "Expert System for Prioritizing the Inspection of Sewers: Knowledge Base Formulation and Evaluation". In: *Journal of Water Resources Planning and Management* 128.2, pp. 121–129.

Han, Song (2023). *EfficientML.ai Lecture 02: Basics of Neural Networks*. TinyML and Efficient Deep Learning Computing. Massachusetts Institute of Technology, Han Lab. URL: https://www.dropbox.com/scl/fi/2qx0cfz7vim0fdhrmy986/lec02.pdf?rlkey=wdjw92hwohp4bhyos8wf5iinb&dl=0.

Han, Song, Huizi Mao, and William J. Dally (2016). *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. arXiv: 1510.00149 [cs.CV].

Hassan, Syed Ibrahim, L. Minh Dang, Irfan Mehmood, Sungbin Im, Cheol Woong Choi, Jae-Mo Kang, Young-Soo Park, and Hyeonjoon Moon (Oct. 2019). "Underground sewer pipe condition assessment based on convolutional neural networks". In: *Automation in Construction* 106, p. 102849. DOI: 10.1016/j.autcon.2019.102849. URL: https://doi.org/10.1016/j.autcon.2019.102849.

Haurum, J.B. and T.B. Moeslund (2020). "A Survey on Image-Based Automation of CCTV and SSET Sewer Inspections". In: *Automation in Construction* 111.

Haurum, Joakim Bruslund and Thomas B. Moeslund (2021). "Sewer-ML: A Multi-Label Sewer Defect Classification Dataset and Benchmark". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 13456–13467.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2015). *Deep Residual Learning for Image Recognition*. arXiv: 1512.03385 [cs.CV].

He, Yang and Lingao Xiao (Jan. 2023). "Structured Pruning for Deep Convolutional Neural Networks: A Survey". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–20. DOI: 10.1109/tpami.2023.3334614. URL: https://doi.org/10.1109/tpami.2023.3334614.

Hubara, Itay, Yury Nahshan, Yair Hanani, Ron Banner, and Daniel Soudry (2021). "Accurate Post Training Quantization With Small Calibration Sets". In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, pp. 4466–4475. URL: https://proceedings.mlr.press/v139/hubara21a.html.

Iandola, Forrest N., Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer (2016). *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size*. arXiv: `1602.07360 [cs.CV]`.

Ioffe, Sergey and Christian Szegedy (2015). *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. arXiv: `1502.03167 [cs.LG]`.

Jaiswal, Sejal (Feb. 2024). *Multilayer Perceptrons in Machine Learning: A Comprehensive guide*. URL: `https://www.datacamp.com/tutorial/multilayer-perceptrons-in-machine-learning`.

Jordan, Michael I. and Tom M. Mitchell (July 2015). "Machine learning: Trends, perspectives, and prospects". In: *Science* 349.6245, pp. 255–260. DOI: `10.1126/science.aaa8415`. URL: `https://doi.org/10.1126/science.aaa8415`.

June, Florian (Nov. 2023). "Model Quantization 2: Uniform and non-Uniform Quantization". In: URL: `https://medium.com/@florian_algo/model-quantization-2-uniform-and-non-uniform-quantization-47ca5b5d3ec0`.

Kolvenbach, Hendrik, David Wisth, Russell Buchanan, Giorgio Valsecchi, Ruben Grandia, Maurice Fallon, and Marco Hutter (May 2020). "Towards autonomous inspection of concrete deterioration in sewers with legged robots". In: *Journal of field robotics* 37.8, pp. 1314–1327. DOI: `10.1002/rob.21964`. URL: `https://doi.org/10.1002/rob.21964`.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (May 2017). "ImageNet classification with deep convolutional neural networks". In: *Communications of The ACM* 60.6, pp. 84–90. DOI: `10.1145/3065386`. URL: `https://doi.org/10.1145/3065386`.

Langeveld, J. and F. Clemens (2015). "Special Issue on 'Sewer asset management'". In: *Urban Water Journal* 13.1, pp. 1–2.

LeCun, Y., L. D. Jackel, Bernhard E. Boser, John S. Denker, Hans Peter Graf, Isabelle Guyon, D. Henderson, Richard Howard, and W. Hubbard (Nov. 1989). "Handwritten digit recognition: applications of neural network chips and automatic learning". In: *IEEE Communications Magazine* 27.11, pp. 41–46. DOI: `10.1109/35.41400`. URL: `https://doi.org/10.1109/35.41400`.

Li, Duanshun, Anran Cong, and Shengli Guo (2019). "Sewer damage detection from imbalanced CCTV inspection data using deep convolutional neural networks with hierarchical classification". In: *Automation in Construction* 101, pp. 199–208.

Li, Zhuo, Hengyi Li, and Lin Meng (Mar. 2023). "Model Compression for Deep Neural Networks: A Survey". In: *Computers* 12.3, p. 60. DOI: `10.3390/computers12030060`. URL: `https://doi.org/10.3390/computers12030060`.

Min, John Tan Chong and Mehul Motani (July 2022). "DropNet: Reducing Neural Network Complexity via Iterative Pruning". In: *arXiv (Cornell University)*. DOI: `10.5555/3524938.3525805`. URL: `https://arxiv.org/abs/2207.06646`.

Mishra, Mayank (Dec. 2021). "Convolutional neural networks, explained - towards data science". In: URL: `https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939`.

Moradi, S. and T. Zayed (2017). "Real-Time Defect Detection in Sewer Closed Circuit Television Inspection Videos". In: *Pipelines 2017*, pp. 295–307.

Mounce, Stephen R., Will J. Shepherd, Joby B. Boxall, Horoshenkov Kiril lV., and Jordan H. Boyle (2021). "Autonomous robotics for water and sewer networks". eng. In: *Hydrolink* 2021/2. Ed. by International Association for Hydro-Environment Engineering, Research (IAHR), International Association for Hydro-Environment Engineering, and Research (IAHR), p. 62. URL: `https://hdl.handle.net/20.500.11970/109511`.

Myrans, Joshua, Richard Everson, and Zoran Kapelan (Oct. 2018). “Automated detection of fault types in CCTV sewer surveys”. In: *Journal of Hydroinformatics* 21.1, pp. 153–163. DOI: 10.2166/hydro.2018.073. URL: https://doi.org/10.2166/hydro.2018.073.

NEN-EN 13508-2 (2021). *Investigation and assessment of drain and sewer systems outside buildings - Part 2: Visual inspection coding system*. Standard. NEN ().

Neuralmagic and Neuralmagic (Mar. 2023). *Part 1: What is Pruning in Machine Learning?* URL: https://neuralmagic.com/blog/pruning-overview/.

Novac, Pierre-Emmanuel, Ghouthi Boukli Hacene, Alain Pégatoquet, Benoît Miramond, and Vincent Gripon (Apr. 2021). “Quantization and Deployment of Deep Neural Networks on Microcontrollers”. In: *Sensors* 21.9, p. 2984. DOI: 10.3390/s21092984. URL: https://doi.org/10.3390/s21092984.

Quantization — PyTorch 2.3 documentation (n.d.). URL: https://pytorch.org/docs/stable/quantization.html.

Ridnik, Tal, Hussam Lawen, Asaf Noy, Emanuel Ben Baruch, Gilad Sharir, and Itamar Friedman (2020). *TResNet: High Performance GPU-Dedicated Architecture*. arXiv: 2003.13630 [cs.CV].

Robins, Mark (June 2023). *The Difference Between Artificial Intelligence, Machine Learning, and Deep Learning*. URL: https://www.linkedin.com/pulse/difference-between-artificial-intelligence-machine-learning-robins?trk=public_profile_article_view.

Stichting RIONED (2020). *Visuele inspectie nieuwe stijl (2020)*. URL: https://www.riool.net/visuele-inspectie-nieuwe-stijl-2020-.

– (n.d.[a]). *Geld voor watertaken*. URL: https://www.riool.info/geld-voor-watertaken.

– (n.d.[b]). *Wat de gemeente doet*. URL: https://www.riool.info/wat-de-gemeente-doet.

Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich (2014). *Going Deeper with Convolutions*. arXiv: 1409.4842 [cs.CV].

Szeliski, Richard (May 2011). “Computer vision: algorithms and applications”. In: *Choice Reviews Online* 48.09, pp. 48–5140. DOI: 10.5860/choice.48-5140. URL: https://doi.org/10.5860/choice.48-5140.

Tanaka, T., K. Harigaya, and T. Nakamura (2014). “Development of a peristaltic crawling robot for long-distance inspection of sewer pipes”. In: *2014 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, pp. 1552–1557. DOI: 10.1109/AIM.2014.6878304.

Turing (Sept. 2022). *What Is the Necessity of Bias in Neural Networks?* URL: https://www.turing.com/kb/necessity-of-bias-in-neural-networks#what-is-bias-in-a-neural-network?.

Unzueta, Diego (Oct. 2022). *Fully Connected Layer vs. Convolutional Layer: Explained*. URL: https://builtin.com/machine-learning/fully-connected-layer.

USEPA (2016). *Clean watersheds needs survey 2012*. Tech. rep. United States Environmental Protection Agency, Washington, p. 120.

Vadera, Sunil and Salem Ameen (Jan. 2022). “Methods for Pruning Deep Neural Networks”. In: *IEEE Access* 10, pp. 63280–63300. DOI: 10.1109/access.2022.3182659. URL: https://doi.org/10.1109/access.2022.3182659.

van Riel, W., J. Langeveld, P. Herder, and F. Clemens (2012). “INFORMATION USE IN DUTCH SEWER ASSET MANAGEMENT”. In: *Proceedings of the 7th World Congress on Engineering Asset Management*. WCEAM. Daejon City, Korea: Springer, pp. 615–624.

van Riel, Wouter (2016). "On Decision-Making for Sewer Replacement". PhD thesis. Delft University of Technology.

Wang, Ya, Dongliang He, Fu Li, Xiang Long, Zhichao Zhou, Jinwen Ma, and Shilei Wen (2019). *Multi-Label Classification with Label Graph Superimposing*. arXiv: `1911.09243 [cs.CV]`.

Weksler, Matan (Dec. 2021). *Deep Dive Into Multi-Bit Weighted Quantization for CNNs*. URL: `https://medium.com/swlh/deep-dive-into-multi-bit-weighted-quantization-for-cnns-d2723afdc5db`.

Wirahadikusumah, R., D. Abraham, and T. Iseley (2001). "Challenging Issues in Modeling Deterioration of Combined Sewers". In: *Journal of Infrastructure Systems* 7.2, pp. 77–84.

WWAP (2017). *The United Nations World Water Development Report 2017. Wastewater: The Untapped Resource.* Tech. rep. United Nations World Water Assessment Programme, UNESCO, Paris, p. 120.

Xie, Qian, Dawei Li, Jinxuan Xu, Zhenghao Yu, and Jun Wang (Oct. 2019). "Automatic Detection and Classification of Sewer Defects via Hierarchical Deep Learning". In: *IEEE Transactions on Automation Science and Engineering* 16.4, pp. 1836–1847. DOI: `10.1109/tase.2019.2900170`. URL: `https://doi.org/10.1109/tase.2019.2900170`.

Xu, Fang, Qing Li, Jiasong Zhu, Zhipeng Chen, Dejin Zhang, Kechun Wu, Kai Ding, and Qingquan Li (Oct. 2022). "Sewer defect instance segmentation, localization, and 3D reconstruction for sewer floating capsule robots". In: *Automation in construction* 142, p. 104494. DOI: `10.1016/j.autcon.2022.104494`. URL: `https://doi.org/10.1016/j.autcon.2022.104494`.

Yin, Xianfei, Tianxin Ma, Ahmed Bouferguène, and Mohamed Al-Hussein (2021). "Automation for sewer pipe assessment: CCTV video interpretation algorithm and Sewer Pipe Video Assessment (SPVA) system development". In: *Automation in Construction* 125, p. 103622.

Zaidi, S.A.R., A.M. Hayajneh, M. Hafeez, and Q.Z. Ahmed (2022). "Unlocking Edge Intelligence Through Tiny Machine Learning (TinyML)". In: *IEEE Access* 110, pp. 100867–100877.

Zhao, Chenyang, Chuanfei Hu, Hang Shao, Zhe Wang, and Yongxiong Wang (2022). "Towards trustworthy multi-label sewer defect classification via evidential deep learning". In.