Exploring Seasonal Data Drift in Flood Prevention: Enhancing Explainability in Computer Vision Models

Graham Davies

Master of Science in Data Science
The University of Bath
2024

Exploring Seasonal Data Drift in Flood Prevention: Enhancing Explainability in Computer Vision Models

Submitted by: Graham Davies

Copyright

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Master of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Abstract

This paper explores the automation of culvert monitoring through the use of computer vision models, particularly in the context of seasonal data drift, where environmental changes affect model accuracy. Current methods rely on manual inspections, which are labour-intensive and present safety challenges. Trash screens, designed to prevent culvert blockages, can lead to flooding if not well-maintained.

The research investigates the performance of models trained on season-specific data and generalised models across all seasons. Findings reveal that season-specific models outperform generalised models during their respective seasons but suffer when applied across different periods. Conversely, generalised models exhibit more consistent performance across diverse datasets. Explainability techniques showed that the models generally focus on relevant features, though occasional misdirected attention highlights areas for future improvement.

This study emphasises the importance of continuous retraining to account for environmental changes. It suggests incorporating the sequential nature of captured images and reassures the computer vision methods are ready for real-world implementation. The research contributes to the advancement of automated culvert monitoring, improving flood prevention strategies through more accurate and interpretable models.

Contents

List of Figures

List of Tables

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor, Dr Andrew Barnes, for his continuous support throughout this study. His patience, advice, and passion for the subject guided me through the entire process, enabling me to complete and present a dissertation that I am truly proud of. I would also like to extend my thanks to Dr Thomas Kjeldsen for his invaluable advice and dedication to effective trash screen implementation. Both Dr Barnes and Dr Kjeldsen share a deep commitment to the United Nations Sustainable Development Goals, and it has been an honour to contribute to this important initiative.

I would also like to extend my gratitude to the University of Bath's Computer Science Department and to all the lecturers who provided the essential knowledge required to complete this study.

Finally, I would like to express my gratitude to the University of Reading for providing the publicly available dataset on which this research is based. The effort involved in labelling this data and making it accessible to the public is deeply appreciated.

Chapter 1

Introduction

Culverts are vital components of modern infrastructure, allowing water to pass beneath roads, railways, and other structures, thereby preventing disruptions in the natural flow of watercourses. As urban areas continue to expand and the effects of climate change intensify, the resilience of these structures becomes increasingly critical. Extreme weather events, particularly intense rainfall, can block culverts and cause flooding, which poses significant risks to public safety and infrastructure.(Miller and Hutchins, 2017) Ensuring the reliability of culverts through consistent monitoring and maintenance is therefore essential to mitigate these risks (Mohammadi, Sherafat and Rashidi, 2023).

Trash screens, designed to filter out debris and prevent blockages, are a common feature in culvert systems. However, while these screens are effective at catching large objects, they can also accumulate debris, potentially exacerbating the risk of flooding if not properly managed (Blanc et al., 2014). The traditional approach to maintaining these systems involves regular manual inspections, which are both labour-intensive and costly. Additionally, during periods of high water flow, the removal of blockages can pose serious safety risks to maintenance crews. Given these challenges, there is a pressing need for an automated, early-warning system that can accurately assess the risk of blockages in culverts, thereby enabling more efficient and safer flood prevention strategies (Vandaele, Dance and Ojha, 2024).

Recent advances in machine learning, particularly in computer vision, offer promising solutions for automating the monitoring of culvert conditions (Iqbal, Barthélemy and Perez, 2022). However, developing a robust and reliable model for this purpose is complicated by several factors. The dynamic nature of culvert environments, with seasonal variations in vegetation, water levels, and lighting conditions, introduces significant challenges (Cornelius Smith et al., 2023). These changes can lead to a phenomenon known as data drift, where the performance of a model deteriorates over time as the input data gradually diverges from the data on which the model was originally trained (Kore et al., 2024).

This research focuses on investigating the impact of seasonal data drift on the performance of computer vision models applied to trash screens. By training models on data from different seasons and evaluating their performance, this study aims to identify and quantify the effects of seasonal variations on model accuracy. Additionally, the research seeks to enhance the interpretability of these models through the application of various explainability techniques, providing insights into the decision-making processes of the models and guiding future improvements in flood prevention strategies.

Chapter 2

Literature and Technology Survey

This literature and technology survey is structured into four main sections. The first section explores previous applications of machine learning in flood prevention, providing a contextual background for the current research. The second section delves into two key concepts relevant to this project: convolutional neural networks and transfer learning, explaining their significance and potential utility. The third section focuses on existing research in the specific area of interest, data drift, highlighting its importance to this study. The final section introduces explainability techniques and highlights their importance in understanding how convolutional neural networks function.

2.1 Applications of Machine Learning in Flood Prevention

This section reviews prior research focused on the application of machine learning techniques in flood prevention, particularly in the context of culvert blockage detection.

Iqbal, Barthélemy and Perez (2022) experimented on whether a visual and hydraulic blockage could be interrelated from a single image. Using a conventional deep learning pipeline (feature extraction and regression using an artificial neural network) and applying transfer learning to already existing end-to-end models, a relationship between visual and hydraulic blockages was found. However, it was noted that the dataset contained the same lighting and background, and only had variations in culvert type, water levels and debris. While this approach showed promising results, the generalisability of the findings is limited due to the controlled nature of the dataset. In real-world environments, where lighting and vegetation vary significantly, model performance could deteriorate, highlighting the need for more robust datasets that reflect these variations. Iqbal, Barthélemy and Perez (2022)

2.1.1 New site implementation

The feasibility of implementing computer vision techniques at novel sites was explored by Vandaele, Dance and Ojha (2024), who examined binary classification, image similarity matching, and anomaly detection techniques. Notably, image similarity matching, using a Siamese network architecture, was found to be particularly effective for training models to classify data at new sites. With only five reference images, this approach optimised the

balance between manual annotation efforts and model performance, facilitating the practical deployment of computer vision solutions at previously unseen locations. However, while this approach demonstrated high performance at new sites, image cropping was implemented in the study. This technique theoretically aims to focus the model on the trash screen, yet it remains unclear what the model's focus is prior to its implementation.

2.1.2 Object Detection

In the study by Iqbal et al. (2022), object detection methods were examined using YOLOv4 and Fast R-CNN models, along with partially simulated data. The models' performance was hindered by background noise visually similar to vegetation present in the data. However, Iqbal et al. (2022) notes that this method may work for a specific use case, but is likely to fail in varying lighting conditions. This reveals a significant challenge in applying machine learning techniques in dynamic environments, such as natural culverts. Background noise can interfere with accurate feature detection, particularly when models are not trained on a diverse dataset. Additionally, the study did not address the prediction of blockages, indicating a gap in research where models need to be robust enough to adapt in real time to changing environmental conditions.

2.1.3 Probabilistic Approaches

Streftaris et al. (2013) proposed a Bayesian stepwise approach to predict blockages at trash screens using meteorological, channel, land use, and social deprivation data. This approach successfully predicted significant debris loads that could induce flooding, demonstrating the utility of probabilistic methods in flood risk management.

Similarly, Yazdi (2018) used Monte Carlo simulations to enhance flood prevention strategies. The study emphasised the importance of avoiding bottlenecks by suggesting that bypass lines in bottleneck areas could increase system resiliency.

While probabilistic approaches offer valuable insights into predicting flood risks, they tend to be computationally expensive and reliant on domain-specific data. This reliance could be a limitation when applying these techniques to more generalised flood prevention models that need to be widely applicable across various environments.

2.1.4 Data Preprocessing Techniques

Due to the absence of comprehensive blockage-related datasets, Iqbal, Barthélemy and Perez (2023) explored the use of synthetic data. By combining synthetic culvert data with real images, and training on synthetic data followed by fine-tuning on real-world data, they observed a slight improvement in detection performance. This study demonstrates the potential of synthetic data to supplement real-world datasets in scenarios where data scarcity is a challenge. While synthetic data provides an interim solution, it cannot fully replicate the complexities of real-world environments, potentially limiting its effectiveness in real-world applications.

Cornelius Smith et al. (2023) further highlighted the importance of data preprocessing, demonstrating that data augmentation techniques improved model accuracy when determining whether a culvert was blocked. The study suggests that additional preprocessing techniques, such as image subtraction, should be investigated to enhance model robustness. However,

while data augmentation is beneficial, it cannot fully resolve issues such as data drift, which is caused by systematic changes over time and may require more sophisticated solutions.

2.2 Background in Computer Vision

This section introduces key concepts in computer vision, focusing on their relevance to image classification tasks in the context of flood prevention.

2.2.1 Convolutional Neural Networks

A Convolutional Neural Network (CNN) is a specialised type of Artificial Neural Network designed for image processing tasks. CNNs are structured in three dimensions: height, width, and depth. The height and width represent the spatial dimensions, while depth represents the activation volume (O'Shea and Nash, 2015).

Architecture

The architecture of a CNN typically includes three types of layers, excluding the input and output layers: convolutional layers, pooling layers, and fully connected (dense) layers (O'Shea and Nash, 2015). A basic illustration of the CNN architecture is shown in Figure 2.1.

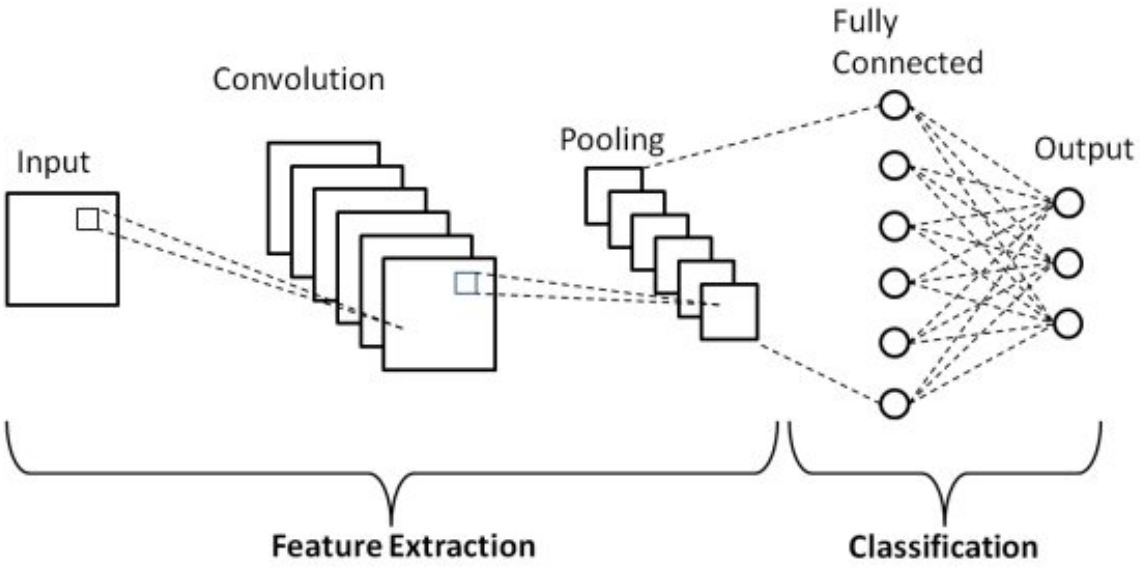


Figure 2.1: The Basic architecture of a CNN.(Phung and Rhee, 2019)

The convolutional and pooling layers are integral to the feature extraction process. The convolutional layer identifies patterns and features within specific regions of the input image, while the pooling layer performs downsampling, reducing the number of parameters and computational load.

The fully connected layer functions as the classification component, producing scores used for classification, similar to fully connected layers in standard artificial neural network architectures.

Applications

The first significant breakthrough in image classification using CNNs occurred in 2012 with the development of AlexNet, which won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) by a large margin (Russakovsky et al., 2015). Since then, CNNs have undergone numerous improvements and are now the preferred choice for image classification tasks (Ghosh et al., 2020).

2.2.2 Transfer Learning

Transfer learning is a technique that addresses the challenge of limited data by leveraging pre-trained models. This approach involves using the generic features learned by a model on a large dataset, such as ImageNet, and fine-tuning the model for a specific task (Yamashita et al., 2018).

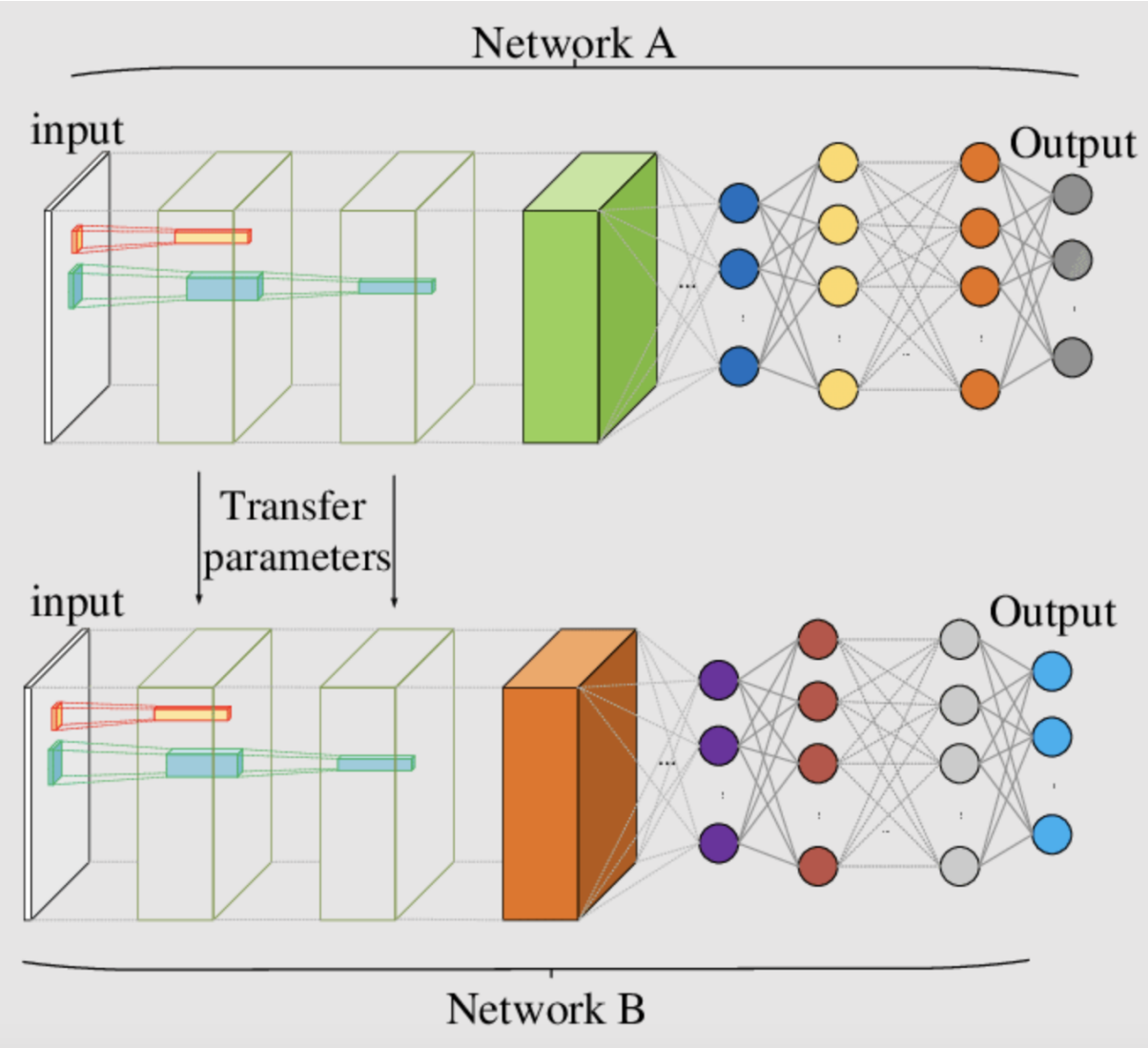


Figure 2.2: The Concept of Transfer Learning.(Lemley, Bazrafkan and Corcoran, 2017)

Figure 2.2 demonstrates the concept of transfer learning. Network A represents a model trained with specific data with specific classification network layers. Network B, transfers the parameters in the feature extraction component of the model, and trains new classification

layers to the end of the network. This is relevant to this study as we can use a pre-built model architecture design and parameters, and adapt the classification part to blockage detection.

Transfer learning has been widely used in studies relevant to this project, enabling the efficient adaptation of models to specific tasks even when data is limited.

2.3 Data Drift

Data drift is defined as the systematic change in the underlying distribution of input features (Kore et al., 2024). It is particularly relevant to the goal of this project as culvert images can change over time due to seasonal differences in vegetation, lighting conditions, and fluctuating water levels. This leads to a gradual divergence between the distribution of the newer data and the data the model was originally trained on, which is known as data drift. There appears to be much research in detecting drift, but very little that proposes counter-drift measures.

2.3.1 Drift Detection

Kore et al. (2024) investigated various drift detection techniques in a clinical computer vision context, finding that aggregate performance metrics are often inadequate for detecting data drift. Data-based drift detection was shown to correlate strongly with sample size, a consideration that is particularly relevant when dealing with moderately sized datasets. While these findings are valuable, they also highlight a limitation: detecting drift is only the first step. Addressing drift in a way that preserves model performance remains a key challenge.

Senarathna et al. (2023) proposed a method based on changes in the distribution of prediction probabilities in a classification neural network. Their experiments on various types of data drift, including weather-induced drift, demonstrated the effectiveness of this method in detecting data drift. This highlights the importance of continuous monitoring and adaptation in machine learning models deployed in dynamic environments. However, the computational cost of continuous drift detection, particularly in resource-constrained environments, is a challenge that has yet to be adequately addressed.

2.3.2 Counter Drift Techniques

Roschewitz et al. (2023) investigated data drift in medical image data, which experiences image drift through replacements of scanner hardware and updates to image processing. Through the use of unsupervised prediction alignment, where one applies linear piecewise cumulative distribution matching between the prediction distribution on the unseen dataset and the reference prediction distribution, this method was found to be successful in solving the data drift endured during image processing. However, while this method shows promise in medical contexts, its feasibility in flood prevention is unclear due to the real-time demands of flood monitoring systems.

A novel unsupervised technique is proposed by Goel and Ganatra (2023). In doing so, they used the transfer learning technique of Unsupervised Domain Adaptation (UDA). UDA aims to solve feature transferability during fine-tuning and to align the source and target domain distributions simultaneously for image classification and object detection tasks. Through weather-based tests, this method was found to be effective in domain adaptive image classification and object

detection. However, it remains to be seen whether UDA techniques are applicable in resource-constrained flood monitoring contexts, where computational power and data availability might be limited.

Another well-known technique to counter data drift is through the use of data augmentation. Zhang et al. (2019) proposed a novel augmentation technique with successful results. As previously mentioned, Cornelius Smith et al. (2023) also experimented with data augmentation and found positive results. Despite the promise of these techniques, data augmentation alone may not fully mitigate the effects of drift in environments subject to continuous and unpredictable changes.

Although it is noted to be a slightly different concept, Disabato and Roveri (2019) explored concept drift and proposed an active approach. In their approach, the system detects when the underlying concept changes (concept drift) and reacts by adapting the model. It uses transfer learning to use the knowledge gained by the CNN before the drift occurred. By transferring some of this learned knowledge to the new CNN that will operate after the drift, the model can more quickly and effectively adjust to the changed concept without having to completely relearn everything from scratch.

2.4 The Importance of Explainability in Flood Prevention Models

Explainability is essential for high-stakes applications such as flood prevention, as confidence in CNNs diminishes without clear insight into their decision-making processes. This lack of transparency renders it unsafe to apply these models in high-risk or safety-critical fields without first developing methods to explain their decisions (Haar, Elvira and Ochoa, 2023). Achieving model accuracy alone is insufficient; domain experts and decision-makers must be able to trust and comprehend the models. Deep learning methods, such as CNNs, can include numerous layers and millions of parameters, making them 'black-box' models due to their lack of interpretability (Barredo Arrieta et al., 2020). Despite the strong performance of deep learning models like CNNs in tasks such as culvert blockage detection, their opacity raises concerns about public safety and infrastructure, which may be at risk (Barredo Arrieta et al., 2020).

2.4.1 Facilitating Collaboration Between AI Developers and Domain Experts

Explainability serves to bridge the gap between AI developers and domain experts, thereby enabling collaboration in flood management. In a medical setting, Holzinger et al. (2017) argues that the black-box nature of deep learning models must be both traceable and explainable for legal and privacy reasons. Similarly, while AI experts handle the technical aspects of model development, domain experts, such as hydrologists, must be able to understand and trust the model outputs. Explainable models foster transparency and trust, which are essential for their implementation in real-world settings (Holzinger et al., 2017).

2.5 Conclusion

Explainability techniques are key in flood prevention models as they enhance reliability, model understanding and collaboration between AI experts and flood prevention experts. These techniques support better debugging, regulatory compliance, and informed decision-making, contributing to the long-term reliability of machine learning applications in critical infrastructure.

Moreover, addressing data drift is essential for developing robust models that can adapt to seasonal and environmental changes. This study aims to explore the impact of data drift on model performance, thereby advancing the understanding of how these factors influence machine learning in flood prevention. Together, explainability and managing data drift are central to ensuring the effectiveness and sustainability of flood monitoring systems.

Chapter 3

Project Overview and Technical Requirements

This chapter outlines the overall objectives and the technical requirements necessary to ensure the successful completion of the project.

3.1 Aim

This project aims to achieve a series of interrelated objectives, each of which is crucial to the overall outcomes of the research. The initial objective is to develop a model capable of accurately determining whether a trash screen is blocked, with performance metrics comparable to those of previous studies. The primary goal is to investigate the existence of seasonal data drift in computer vision models applied to trash screens. By training a model with the same architecture as the initial stage but using data from different seasons, this research seeks to identify and quantify any variations in performance attributable to seasonal changes. If seasonal data drift is detected, the secondary aim is to explore the underlying mechanisms driving this phenomenon. The ultimate objective is to provide meaningful insights that enhance the understanding of computer vision models for trash screens, thereby improving flood prevention strategies and public safety.

Figure 3.1 illustrates the core objective underpinning this project.

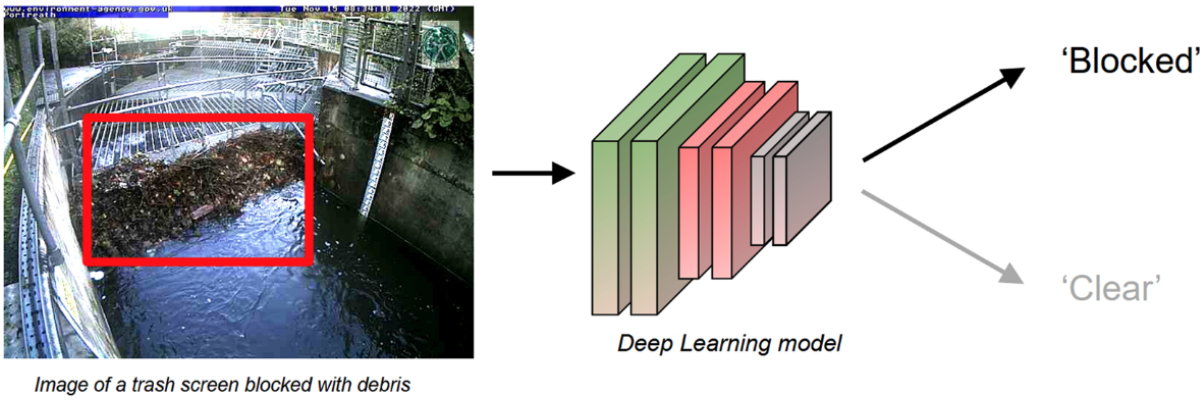


Figure 3.1: The core aim of the project (Vandaele, Dance and Ojha, 2024).

3.2 Objectives

1. Develop a classifier using transfer learning with ResNet-50 to accurately detect whether a culvert is at risk of flooding based on image data.
2. Utilise the architecture of the original classifier to train a model for each season and compare the performance of these models across different seasonal datasets. This comparison will help determine the presence and extent of seasonal data drift.
3. To evaluate the performance of seasonal models through various explainability techniques, including Integrated Gradients, Saliency Mapping, Smooth Grad-CAM, and Occlusion Sensitivity. These evaluations will be conducted against a baseline model to identify and understand differences in model behaviour across seasons.

By fulfilling these objectives, this project aims to advance the development of an effective and efficient automated system for culvert flood prevention, ultimately enhancing public safety and the resilience of critical infrastructure.

3.3 Technical Requirements

This section outlines the data and Python packages to be used throughout the project. The dataset was selected due to its comprehensiveness and the fact that it is publicly available, eliminating the need for manual labelling. The chosen tools are noted for their capabilities in handling data processing, machine learning, and deep learning tasks.

3.3.1 Data Requirements

An extensive dataset is required for the purpose of this research. The dataset was publicly sourced from the University of Reading (Vandaele, 2023) and comprises images captured from 54 distinct camera sites installed by the Environment Agency (UK). These images were manually labelled into three categories:

- **Blocked**: Where the trash screen appears obstructed.
- **Clear**: Where the trash screen appears unobstructed.
- **Other**: Where the condition of the trash screen is uncertain or unclear.

The dataset includes images collected over different seasons, allowing for the analysis of seasonal variations.

3.3.2 Python Packages

The project will utilise several Python packages to facilitate efficient data processing, model development, and visualisation. The following packages are integral to the project's success:

- **Pandas** (Wes McKinney, 2010): A powerful library providing fast, flexible data structures designed for the intuitive manipulation and analysis of tabular data.
- **NumPy** (Harris et al., 2020): A core package for numerical computations in Python, offering efficient matrix and vector operations fundamental to machine learning.

- **Matplotlib** (Hunter, 2007): A versatile plotting library used for creating static, animated, and interactive visualisations in Python, essential for data exploration and result presentation.
- **Seaborn** (Waskom, 2021): A high-level interface built on top of Matplotlib, designed for creating attractive and informative statistical graphics.
- **Scikit-learn** (Pedregosa et al., 2011): A widely-used library that provides simple and efficient tools for data mining, data analysis, and machine learning.
- **PyTorch** (Paszke et al., 2019): A deep learning framework that facilitates tensor computation with strong GPU acceleration, essential for building and training neural networks.
- **Captum** (Kokhlikyan et al., 2019): A library for model interpretability that provides tools to understand the predictions of PyTorch models, particularly useful for visualising the inner workings of CNNs.
- **TorchCam** (Fernandez, 2020): A package offering easy-to-use CNN visualisation tools, allowing for the generation of class activation maps to interpret model predictions.
- **PIL (Python Imaging Library)** (Clark, 2015): Used for opening, manipulating, and saving various image file formats. The `Image` module from PIL will be utilised for image processing tasks.
- **Torchvision** (maintainers and contributors, 2016): A package containing popular datasets, model architectures, and image transformations for computer vision tasks. The `ResNet50` model from `torchvision.models` will be particularly important for this project.

Chapter 4

Design

4.1 Data Design and Exploratory Data Analysis

4.1.1 Initial Dataset Overview and Labelling

The dataset was organised into site-specific directories, each containing subdirectories labelled 'blocked', 'clear', or 'other'. These subdirectories represent the three image categories. An initial analysis of the dataset revealed imbalances across the different classes, with the 'other' category particularly prevalent in some seasons.

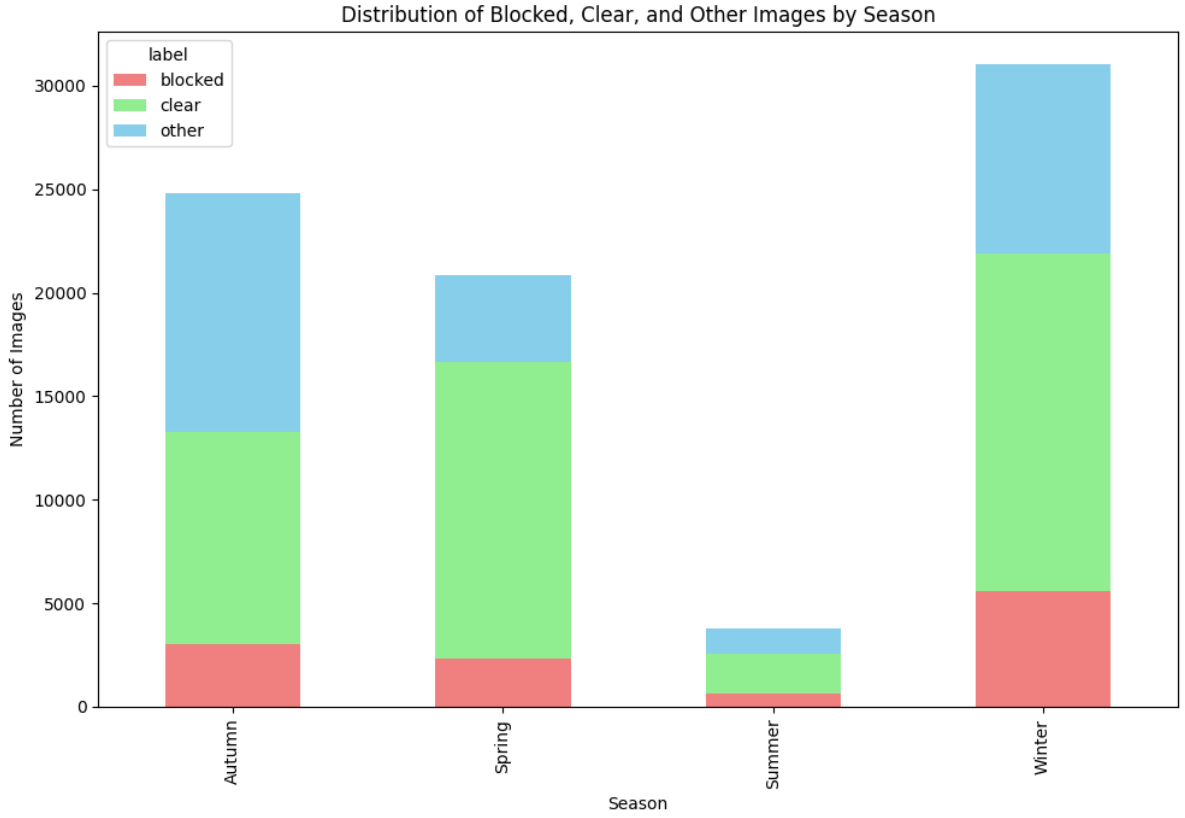


Figure 4.1: Distribution of image counts by season.

Table 4.1 provides a summary of the image counts for each label across the seasons, while

	Autumn	Winter	Spring	Summer	Total
Blocked	3033	5604	2337	617	11591
Clear	10236	16304	14320	1943	42803
Other	11538	9131	4180	1209	26058
Total	24807	31039	20837	3769	80452

Table 4.1: Image counts by label and season (unbalanced dataset).

Figure 4.1 visualises this distribution. Notably, there are far fewer images captured during the summer, and the proportion of images labelled as 'other' varies between seasons. The dataset suggests that it was more challenging to assign a definitive label in autumn compared to spring, possibly due to seasonal variations like floating leave which might not cause a direct blockage.

4.1.2 Data Preprocessing

Exclusion of 'Other' Images

Since the study focuses on a binary classification task (i.e., 'blocked' or 'clear'), all images labelled as 'other' were excluded from the dataset. This significantly reduced the overall dataset size, leaving only 'blocked' and 'clear' images for further analysis.

Basic Data Cleaning and Initial Imbalance

Before balancing, the dataset displayed significant class imbalances. Certain camera sites contained only 'clear' or only 'blocked' images, further contributing to bias in the dataset. Figure 4.2 shows the distribution of images over time, highlighting gaps in data collection and periods of skewed class representation.

As shown in Figure 4.2, a notable gap in data collection occurred between mid-May and mid-September, with a concentration of images around March. This gap, combined with an imbalance in the number of 'blocked' and 'clear' images, necessitated further balancing steps.

4.1.3 Seasonal Data Balancing

Balancing Process

To address the imbalance between 'blocked' and 'clear' images, a custom balancing process was implemented. This involved selecting an equal number of 'blocked' and 'clear' images from each camera site. The smallest class count at each site was used as the sampling limit to ensure balance across the dataset.

Post-Balancing Statistics

Following the balancing process, the dataset was significantly reduced, with each season now containing a near-equal representation of 'blocked' and 'clear' images. Table 4.2 shows the post-balancing image counts. Due to the extremely small number of images collected during summer (only 66), this season was excluded from further analysis.

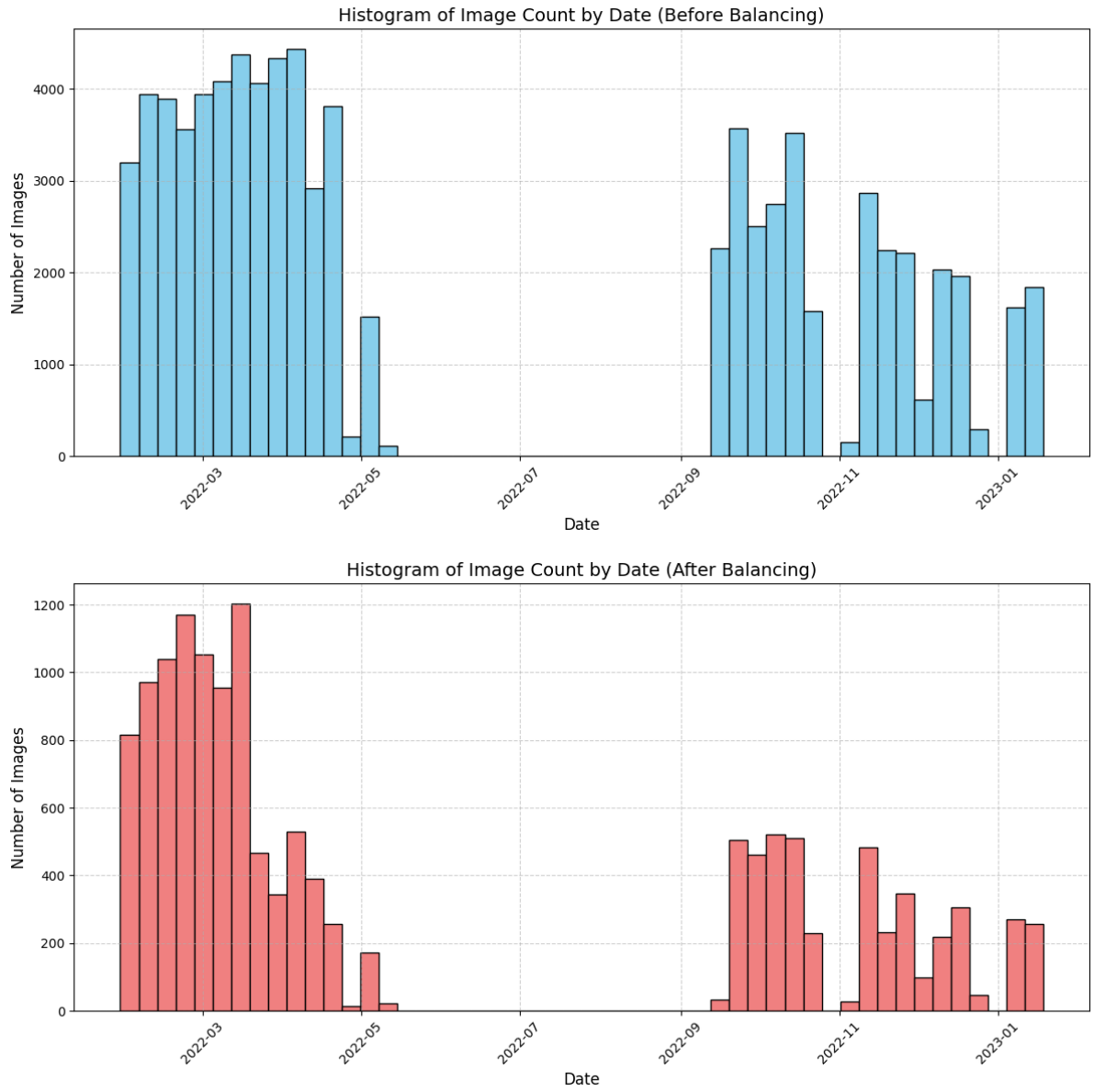


Figure 4.2: Distribution of image counts by date - before and after balancing.

4.1.4 Time-Based Analysis and Seasonal Assignment

Timestamp Extraction

Each image filename contained an embedded timestamp in the format `YYYY_MM_DD_HH_MM`, which represented the capture date and time. These timestamps were extracted and converted to datetime objects using the `pd.to_datetime` function. This step allowed for time-based analysis of the data, helping to assess the impact of seasonal variations on model performance.

Seasonal Subsets

Based on the extracted timestamps, each image was assigned to one of four predefined seasonal subsets:

- **Spring:** 20 March to 20 June

Season	Autumn	Winter	Spring	Summer	Total
Blocked	1975	3940	1023	33	6971
Clear	1975	3940	1023	33	6971
Other	0	0	0	0	0
Total	3950	7880	2046	66	13942

Table 4.2: Image counts by label and season (balanced dataset).

- **Summer:** 21 June to 22 September
- **Autumn:** 23 September to 21 December
- **Winter:** 22 December to 19 March of the following year

The exclusion of the summer season from the final analysis was due to the insufficient number of images available for this period. This seasonal segmentation was critical for investigating the impact of seasonal variations—such as lighting, foliage, and weather—on the classifier's performance.

4.1.5 Final Dataset Preparation for Modelling

Image Resizing and Normalisation

Before training, all images were resized to 224x224 pixels to match the input requirements of the ResNet-50 architecture. Additionally, normalisation was applied using the mean and standard deviation from the ImageNet dataset. This step helped to accelerate convergence during model training by ensuring the data was consistent with the model's expected input format.

Train-Test and Train-Validation Split

Following the exclusion of summer images, the final dataset was divided into training and test sets using an 80:20 split. This ensured that the model could be evaluated on unseen data, providing a reliable measure of its performance. The training set was further divided into an 80:20 train-validation split, enabling more accurate performance estimation during the training process. The images were labelled with 'blocked' assigned a value of 1 and 'clear' assigned a value of 0.

4.2 Design of Classifier

This section details the design and implementation of the classifier used in this research. The classifier, based on a CNN architecture, was specifically designed to perform image classification tasks. The choice of architecture, hyperparameters, and training procedures is discussed in detail, with justifications provided for each decision made during the design process.

4.2.1 Choice of Architecture

Transfer learning, a technique where features and weights from a pre-trained model are utilised and fine-tuned for new data, was used in this research. The backbone model used is based on the ResNet architecture, specifically ResNet-50. ResNet, or Residual Network, is recognised for

its ability to train very deep networks by using residual connections to mitigate the vanishing gradient problem He et al. (2015). The choice of ResNet-50 was further justified by its proven performance in binary blockage detection, as demonstrated by Vandaele, Dance and Ojha (2024).

4.2.2 Hyperparameter Selection

The selection of hyperparameters was important to the successful training of the classifier. The following hyperparameters were chosen:

- **Batch Size:** A batch size of 64 was used to balance the memory constraints of the GPU and the stability of gradient updates.
- **Optimiser:** The Adam optimiser Kingma and Ba (2017) was selected due to its adaptive learning rate capabilities, which help maintain a balance between speed and accuracy during training.
- **Learning Rate:** A learning rate of 0.001 was initially set to begin convergence, as the Adam optimiser dynamically adjusts it.
- **Epochs:** The model was trained for up to 25 epochs, with early stopping applied if validation accuracy failed to improve for 5 consecutive epochs.
- **Weight Initialisation:** The pre-trained weights from the ResNet-50 model, initialised with the ImageNet dataset, were fine-tuned to adapt the model for the specific binary classification task.

4.2.3 Training Procedure

The training process was conducted using the PyTorch framework, which provides robust tools for implementing and training deep learning models. This facilitated the implementation of the training, utilising built-in features such as the Adam optimiser. The overall training procedure involved two key steps:

1. **Training a Binary Classifier on the Entire Dataset:** Initially, a binary classifier was trained on the entire dataset, which combined images from all seasons. This approach enabled the development of a model capable of generalising across various seasonal conditions, such as differing lighting, weather, and foliage scenarios throughout the year.
2. **Training Binary Classifiers for Each Season:** To further investigate the impact of seasonal variations on model performance, separate binary classifiers were trained on data from each season — Winter, Spring, and Autumn. The same hyperparameters and early stopping criteria were applied to ensure consistency across all seasonal models. This step allowed for the evaluation of the classifier's performance within a more controlled environment, isolated from the seasonal variations present in the full dataset.

During training, the cross-entropy loss function was used to measure the difference between the predicted and actual class probabilities. Cross-entropy loss is standard for binary classification problems as it effectively penalises incorrect predictions and rewards correct ones. After minor adjustments to optimise the stopping criterion, all models reached the early stopping condition and produced learning curves similar to Figure 4.3.

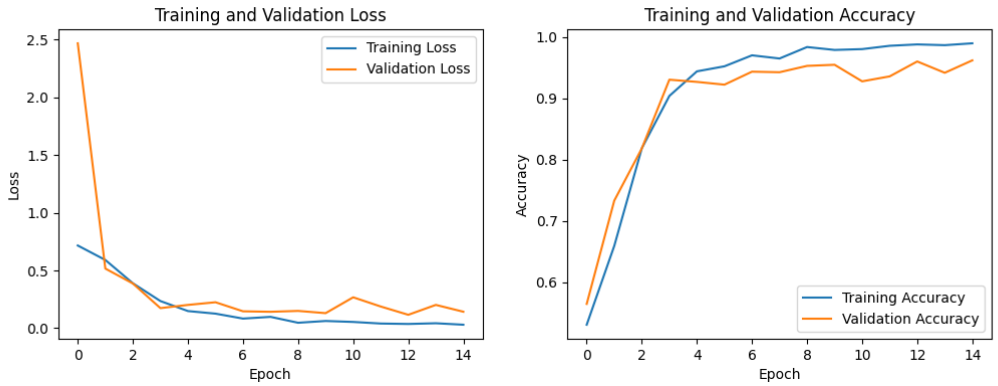


Figure 4.3: Learning curve illustrating a successful training process.

4.2.4 Model Evaluation

Post-training, the classifier's performance was evaluated on a separate test set, which had not been used during training. Accuracy was calculated to provide an assessment of the model's performance.

4.3 Design and Implementation of Methods

This section outlines the design and implementation of the methods used in this research, focusing on various model interpretability techniques. These methods were selected to gain insights into the decision-making processes of convolutional neural networks (CNNs), with an emphasis on enhancing model interpretability. The methods include Saliency Mapping, Integrated Gradients, Occlusion Sensitivity and Smooth Gradient Class Activation Mapping (SmoothGrad-CAM++). The selection of these methods is justified by their ability to provide visual explanations for model predictions, thereby contributing to the interpretability of otherwise opaque deep learning models.

The design of the analysis involves using images from a single site and comparing them across three seasons. The site chosen is located in Shepton Mallet, Somerset, selected due to the availability of 'blocked' images. The image selected for each season was as close as possible to the central date, i.e., the dates furthest from the solstices and equinoxes. To ensure further consistency, all images were captured at 08:30 a.m. Each technique is applied to each image, and the results are then compared.

The original images of each site can be noted in Figure 4.4. These images for the demonstration of findings are deemed to be useful as they are clear and contain a similar amount of debris blocking the trash screen. It is also worth noting the difference in vegetation in each image.

4.3.1 Common Steps for Saliency Mapping, Integrated Gradients, Occlusion Sensitivity, and SmoothGrad-CAM++

All four methods share several preprocessing and model inference steps. The following steps apply to each method:

1. **Input Image Preprocessing:** The input image is preprocessed to ensure compatibility with the ResNet-50 model. This involves resizing the image to 224x224 pixels, converting

(a) Autumn image

(b) Winter image

(c) Spring image

Figure 4.4: Original Images

it into a tensor, and normalising it based on the mean and standard deviation of the ImageNet dataset.

2. **Model Forward Pass:** Once preprocessed, the image is passed through the fine-tuned ResNet-50 model, which outputs class probabilities. The class with the highest probability is selected as the predicted class and used for further analysis in each method.

4.3.2 Saliency Mapping

Saliency mapping Simonyan, Vedaldi and Zisserman (2014) identifies the pixels in an input image that most influence the model's output. This method involves computing the gradient of the output with respect to the input image and taking the absolute value to produce a saliency map. The saliency map highlights the areas of the input that cause the largest change in the output, thus indicating the parts of the image that are most important for the model's decision.

The saliency maps are generated using gradient-based methods, which involve backpropagating the gradient of the predicted class score with respect to the input image. Specifically, the research utilises the maximum absolute gradient across all colour channels to produce a saliency map. The steps involved are as follows:

3. **Gradient Calculation:** To generate the saliency map, the gradient of the class score of the predicted class with respect to the input image is calculated. This is done by setting the image tensor to require gradients and performing a backward pass from the selected class score to the input pixels.

4. **Attribution Map Computation:** The gradient values represent the sensitivity of the model's prediction to changes in each pixel of the input image. By taking the maximum absolute value of the gradients across the RGB channels, a single-channel saliency map is obtained. This map highlights the pixels that most strongly influence the model's decision.

5. **Normalisation and Clipping:** The saliency map is normalised to enhance contrast, making the important regions more visually discernible. This normalisation scales the map values to the [0, 1] range.

6. **Visualisation:** Finally, the saliency map is resized to match the original image dimensions and overlaid on the original image to visualise the regions that the model focuses on when making its prediction. This overlay provides a clear and interpretable visual representation of the model's decision-making process.

4.3.3 Integrated Gradients

Integrated Gradients Sundararajan, Taly and Yan (2017) is a method that attributes the prediction of a model to its input features by integrating gradients along the path from a baseline input to the actual input. It is chosen as a method in comparison to saliency mapping because it integrates along the entire path, rather than focusing on just the final layer. The baseline input is typically a neutral input, such as an all-zero vector. In this research, a zeroed-out (black) version of the input image is used as the baseline. The integration is performed along a straight line in the input space, and the gradients are accumulated to

produce a final attribution score for each input feature. The following steps outline the process used in this research:

3. **Baseline Definition:** The baseline input, in this case an all-zero (black) image of the same dimensions as the input image, is established. This baseline represents a neutral input with no features, serving as the starting point for the Integrated Gradients calculation.
4. **Gradient Integration:** The Integrated Gradients method (Kokhlikyan et al., 2019) computes the gradients of the predicted class score with respect to the input image, integrating these gradients along a straight-line path from the baseline to the input image. This integration results in an attribution map that indicates the contribution of each pixel to the final prediction.
5. **Attribution Map Generation:** The resulting attribution map is summed across the RGB channels to produce a single-channel map. This map highlights the areas of the image that have the most significant impact on the model's prediction.
6. **Normalisation and Clipping:** The attribution map is normalised to a [0, 1] range to enhance visual interpretability. Additionally, the map is clipped to remove extreme values, ensuring that the most relevant regions are clearly discernible.
7. **Visualisation:** Finally, the normalised attribution map is overlaid on the original image. This overlay provides a visual representation of the regions that significantly influence the model's decision, allowing for a clear interpretation of the model's reasoning process.

4.3.4 Occlusion Sensitivity

Occlusion Sensitivity Zeiler and Fergus (2013) is a method used to determine which parts of an input image are most important for a model's prediction by systematically occluding different regions of the image and observing the impact on the prediction. The image is divided into small patches, and each patch is sequentially occluded (e.g., by replacing it with a black square), with the model's output recorded for each occlusion. The change in prediction score indicates the importance of the occluded region. The Captum package Kokhlikyan et al. (2019) was used to implement the techniques. The following steps outline the process used in this research:

3. **Occlusion Strategy:** The Occlusion method (Kokhlikyan et al., 2019) works by sliding a fixed-size window across the image, masking out parts of the image within the window. In this study, the window was set to a shape of 15x15 pixels with a stride of 8 pixels, ensuring that each region is adequately occluded and tested for its impact on the prediction. The occlusion also operates across the RGB channels to account for colour information.
4. **Attribution Map Computation:** As the window slides across the image, the model's output for the predicted class is recorded. The decrease in the output score when a particular region is occluded is used to determine the importance of that region. This information is aggregated to form an attribution map that highlights the regions most critical to the model's decision.
5. **Normalisation and Clipping:** The attribution map is normalised to a [0, 1] range to enhance visual clarity. Clipping is applied to limit extreme values, ensuring the most

relevant areas are clearly visible.

6. **Visualisation:** Finally, the normalised attribution map is overlaid on the original image. This overlay provides a visual representation of the regions that significantly affect the model's prediction when occluded, offering insights into the model's focus areas during the decision-making process.

4.3.5 Smooth Gradient Class Activation Mapping++ (SmoothGrad-CAM++)

SmoothGrad-CAM++ (Omeiza et al., 2019) is a technique that combines the advantages of two methods: SmoothGrad (Smilkov et al., 2017) and Grad-CAM++ (Chattopadhay et al., 2018). SmoothGrad works by averaging gradients over multiple noisy versions of an input, helping to reduce noise and produce smoother visualisations, while Grad-CAM++ improves upon Grad-CAM by using weighted gradients to generate more accurate and refined localisation maps for class-specific heatmaps. By merging these methods, SmoothGrad-CAM++ produces clearer and more precise visualisations of a model's decision-making process. In this research, the technique was implemented using the TorchCam package (Fernandez, 2020), which automatically handles noise addition, gradient extraction, and weight calculation. The following steps outline the process used in this study:

3. **Gradient Extraction with Noise Addition:** To generate the SmoothGrad-CAM++ heatmap, the gradients of the output score for the predicted class with respect to the final convolutional layer of the model are computed across multiple noisy versions of the input image. Noise is automatically added internally by the SmoothGrad-CAM++ method, and the gradients are averaged to reduce noise and produce a smoother heatmap. (Fernandez, 2020)

4. **Importance Weight Calculation:** The extracted gradients are automatically weighted according to their importance by the SmoothGrad-CAM++ method(Fernandez, 2020) producing a more precise localisation of the class-specific features in the input image.

5. **Attribution Map Computation:** The weighted gradients are aggregated by the method to produce a refined heatmap that highlights the regions in the input image contributing most to the prediction. This heatmap is smoother and more precise than those produced by standard Grad-CAM, due to the integration of multiple noisy gradient calculations.

6. **Normalisation and Clipping:** The ReLU activation function is applied to the heatmap by the method to retain only positive values, which correspond to features that positively influence the prediction. The heatmap is then normalised to the [0, 1] range to ensure consistency and clarity in visualisation.

7. **Visualisation:** Finally, the normalised heatmap is resized to match the original image dimensions and overlaid on the original image using the 'overlay_mask' utility. This overlay provides an interpretable visual representation of the regions the model focused on during its decision-making process, with smoother and more accurate localisation of important features.

Chapter 5

Results

5.1 Binary Classifier

In Figure 5.1, the performance of each model trained on different seasons' data is compared. All models achieve test accuracy greater than 95%, on their respective test set indicating strong performance. Given that we are using a balanced dataset, a model that was randomly guessing would achieve an accuracy near 50%. The model with the lowest performance, the all-seasons model, achieves an accuracy of 95%. As the objective of this research is not to create a perfect classifier, the performance of each model is considered sufficient for this study.

5.1.1 Performance Comparison

To determine if data drift occurs across different seasons, we examine the accuracy of each model when tested on data from seasons other than the one on which it was trained. If seasonal data drift exists, we would expect to see varied accuracy results. The results are illustrated in Figure 5.1.

5.1.2 All-Seasons Model

The 'all-seasons' model, trained on a combination of autumn, spring, and winter data, performs consistently across all test datasets. It achieves 95% accuracy on its own test data and performs well on seasonal subsets of its training data. Specifically, the model performs best on the autumn data, achieving 96% accuracy, which is better than the 'all-seasons' test set. The spring and winter subsets, each with 94% accuracy also demonstrate strong performance. This indicates that the all-seasons model is well-generalised and capable of recognising patterns across different seasons, though it slightly underperforms on other test sets compared to the best-performing seasonal subsets.

5.1.3 Autumn Model

The autumn model shows strong performance when tested on autumn data, achieving 96% accuracy, indicating that it is well-tailored to the autumn season. However, its performance significantly drops when tested on other datasets. On the all-seasons data and the spring data, it achieves only 78% accuracy, which is 18% lower than its performance on the autumn

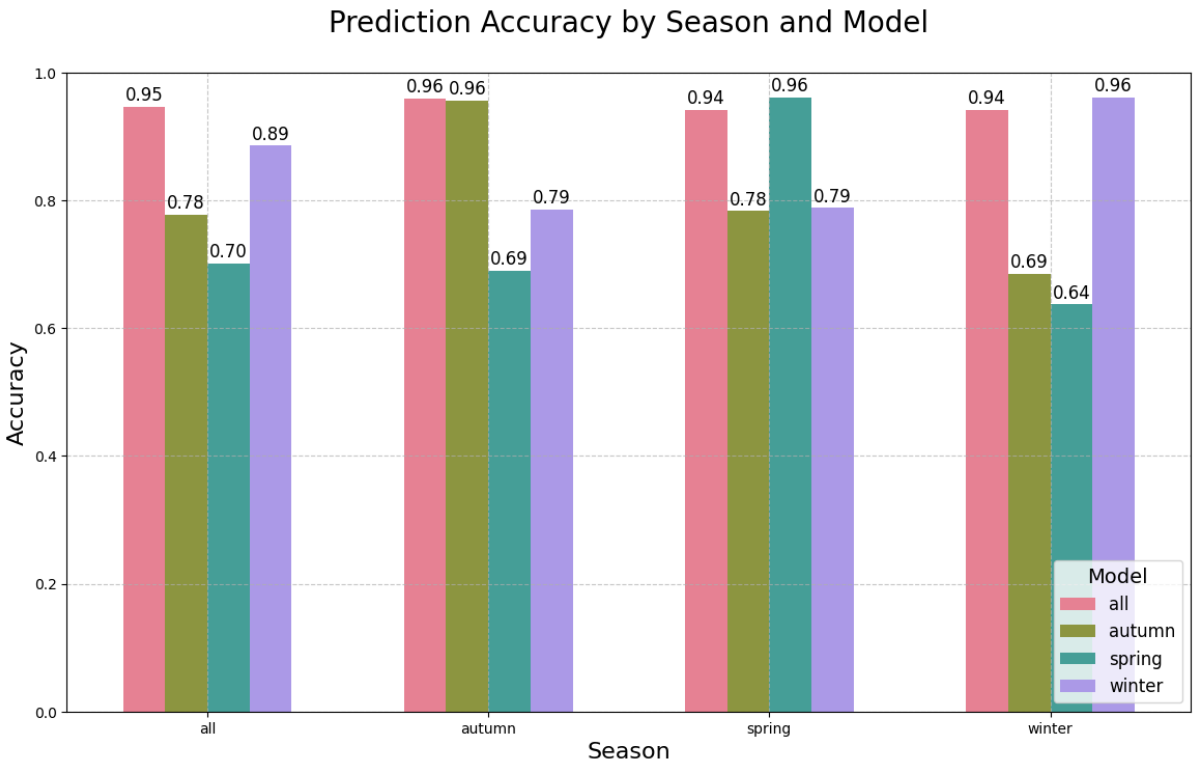


Figure 5.1: Comparison of Models.

data. The model's accuracy further drops to 69% when tested on winter data, indicating a substantial decrease in effectiveness when applied outside its trained season.

5.1.4 Winter Model

Similarly, the winter model achieves 96% accuracy on winter data, demonstrating strong performance within its specific season. However, when tested on other seasonal datasets, its accuracy declines to 89% on the all-seasons dataset, which is 9% lower than its performance on the winter data. The model performs similarly on the autumn and spring data, with accuracy scores of 79%. The performance across seasons is better than the winter or autumn model, but still suggests the winter model is highly specialised for winter data but less effective at generalising to other seasons.

5.1.5 Spring Model

The spring model also exhibits its best performance on its own seasonal data, achieving 96% accuracy, which suggests it is optimised for recognising spring-specific patterns. However, when tested on the all-seasons dataset, its accuracy drops to 70%, a 26% decrease from its performance on the spring data. The performance further declines on the autumn and winter datasets, with accuracy scores of 69% and 64%, respectively. This indicates that the spring model, like the autumn model, struggles to generalise to data from other seasons.

5.1.6 Initial Remarks

These results demonstrate the significant influence of seasonal data on each model's performance, highlighting that a generalised model may not be the optimal choice for all seasons. The observed variations suggest that data drift exists as the models do not perform uniformly across different seasons, likely due to differences in the features that are more relevant or pronounced in each season. This finding is critical for the remainder of this research, as it directs the focus towards understanding the internal workings of the models to identify the specific factors contributing to this data drift.

5.2 Explaining the Seasonal Differences

Understanding seasonal variations is essential for analysing how different factors affect model performance and interpretation throughout the year. In this section, various interpretability techniques - Saliency Mapping, Integrated Gradients, Occlusion Sensitivity and SmoothGrad-CAM++ - are applied to uncover subtleties in model behaviour across winter, spring and autumn.

As the focus is on seasonal differences, the performance of the all-seasons model is not considered.

5.2.1 Saliency Mapping

Saliency mapping provides a pixel-level representation of how the input influences the model's decision-making process. This technique identifies which pixels most significantly affect the model's output.

Autumn Model

Autumn Image 5.2a: The saliency map highlights significant areas around the lower half of the trash screen, particularly focusing on the vegetation and debris present. This suggests that the model effectively detects blockages during the autumn season. Notably, some focus is on leaves that have passed through the trash screen. **Winter Image** 5.2b: The model still identifies some saliency in the lower part of the screen, albeit with slightly less intensity than in the autumn image. The debris above the trash screen is drier than in the autumn image, and some saliency is observed there. **Spring Image** 5.2c: The saliency map for the spring image shows a focus on the same area as in the previous seasons, though with reduced intensity. It is also susceptible to the dry debris located above the trash screen, and there is saliency on the left side wall.

Winter Model

Autumn Image 5.3a: The winter model's saliency map highlights the debris in the lower part of the screen, but it is more concentrated than in the autumn model. There is very little saliency concerning the actual blockage. **Winter Image** 5.3b: The saliency map shows a strong focus on the lower part of the screen, suggesting that the winter model is well-tuned to detect blockages during the winter season. **Spring Image** 5.3c: The saliency map highlights the debris again, with a focus on the debris more on the right side of the trash screen. Unlike the autumn model, there is little saliency around the debris above the trash screen.

(a) Autumn image

(b) Winter image

(c) Spring image

Figure 5.2: Saliency Mapping: Autumn Model

Spring Model

Autumn Image 5.4a: The spring model's saliency map indicates areas of interest similar to those identified by the other models. It appears to identify the built-up debris well, although it also seems to focus on the left side wall. **Winter Image** 5.4b: These results are very similar to those in 5.4a, where there is good focus on the blockage, but also saliency on the left side wall. Unlike the autumn model, the debris above the trash screen does not seem to influence the model. **Spring Image** 5.4c: There is strong saliency around the base of the trash screen, indicating good performance. Again, this model focuses on the left side wall for an unknown reason.

This analysis highlights the significance of each pixel's contribution to the model's decision-

(a) Autumn image

(b) Winter image

(c) Spring image

Figure 5.3: Saliency Mapping: Winter Model

making process. In Figures 5.2, 5.3, and 5.4, it is evident that the models focus on the regions expected to be important. The highlighted areas illustrate the relevance of each pixel's contribution. Notably, sticks and small debris located at the base of the trash screen appear to contribute the most to the model's focus.

(a) Autumn image

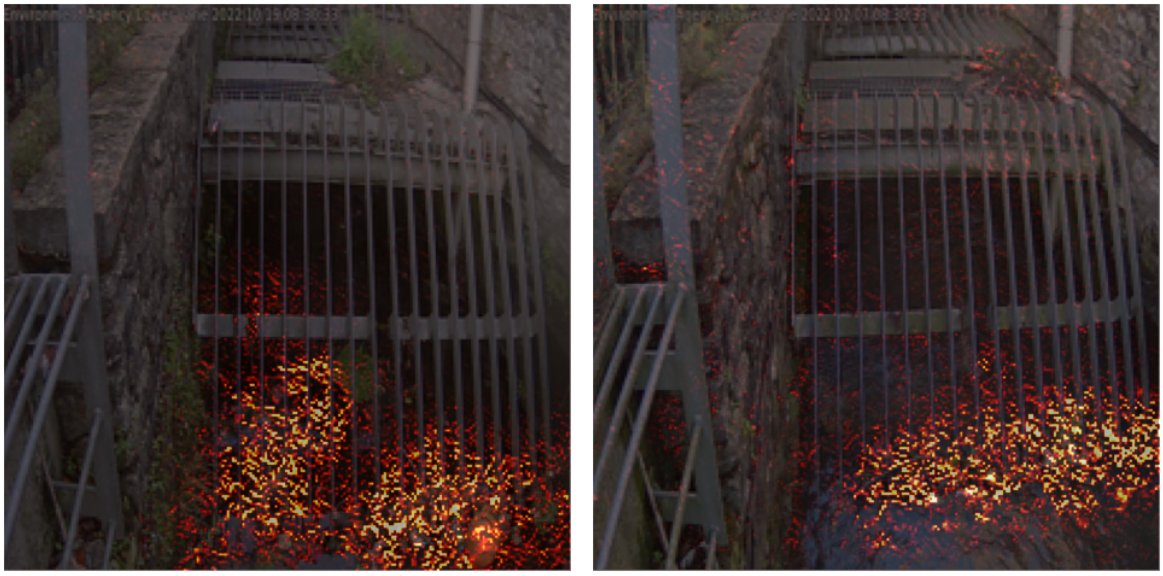
(b) Winter image

(c) Spring image

Figure 5.4: Saliency Mapping: Spring Model

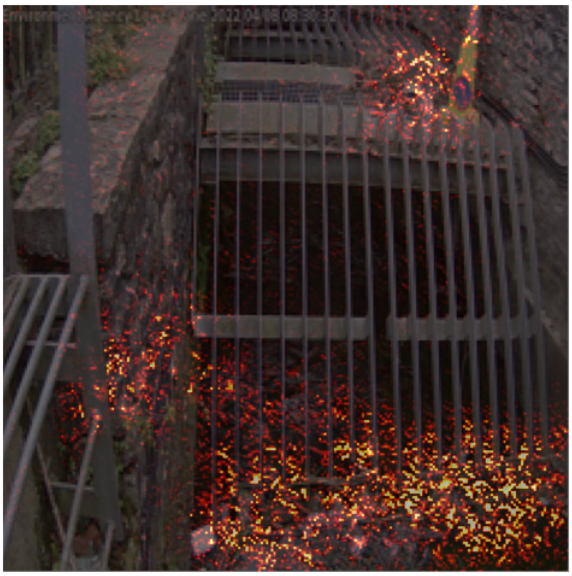
5.2.2 Integrated Gradients

The purpose of using integrated gradients is to extend the saliency mapping analysis by understanding how each feature attribution map contributes through the entire path through the network. At a glance, it can be noted that various parts of the image at some point influence the model. For example, in Figure 5.7a, various areas on the upper left side appear to influence the model. However, as seen in the saliency mapping in Figure 5.4a, these points do not ultimately influence the final decision. The results are therefore broader in the integrated gradient images.



(a) Autumn image

(b) Winter image



(c) Spring image

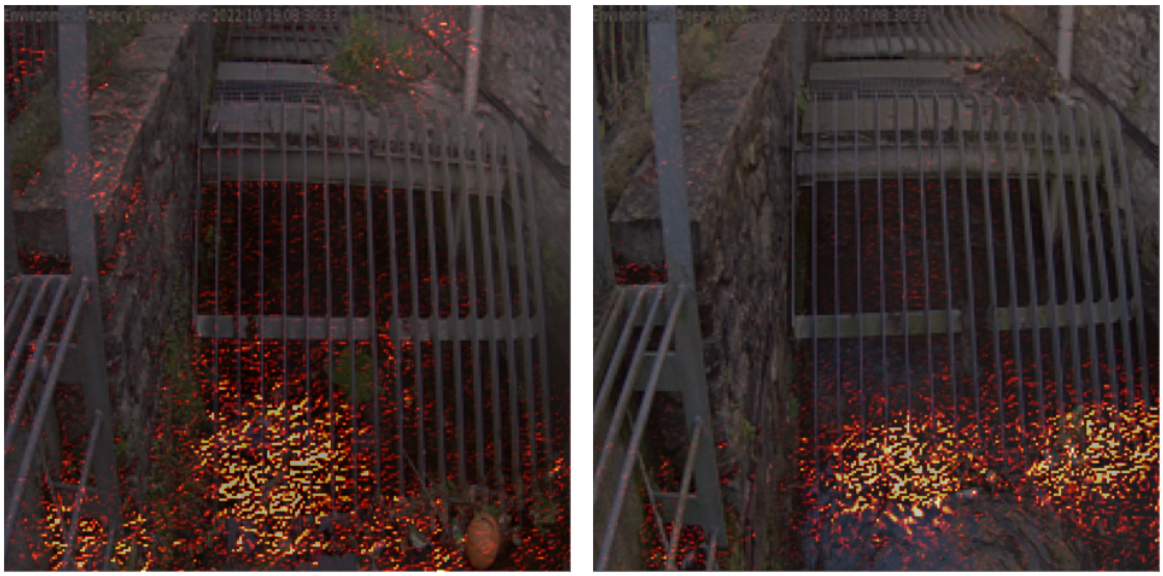
Figure 5.5: Integrated Gradients: Autumn Model

Autumn Model

Autumn Image 5.5a: The autumn model exhibits strong gradient responses primarily concentrated in the lower half of the image, especially around the area where leaves and foliage accumulate at the base of the trash screen. The integrated gradients seem to focus not only on leaves outside the trash screen, but some responses are triggered by leaves located behind the trash screen. **Winter Image** 5.5b: In contrast, the autumn model shows a more subdued gradient response when applied to the winter image. The areas of focus are slightly more dispersed, with most attention given to the lower section, but some attention given to the debris above the trash screen. **Spring Image** 5.5c: The model's response to the spring image is intermediate, with gradients showing concentration in the lower part of the image,

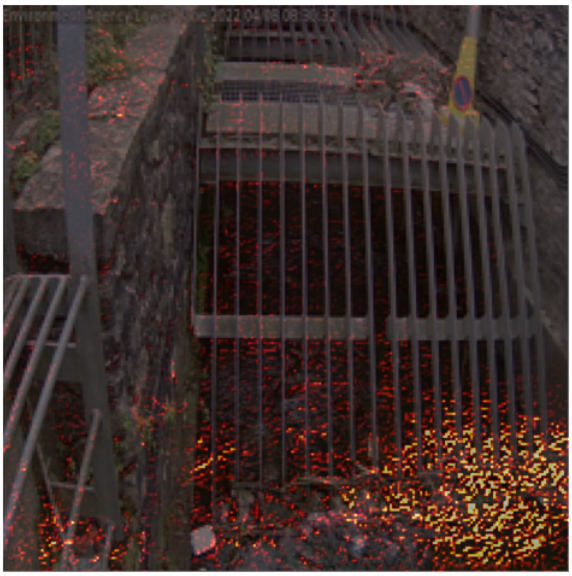
particularly around the base of the structure. The presence of drier debris located above the trash screen has triggered a strong response, suggesting the autumn model may focus on dry debris.

Winter Model



(a) Autumn image

(b) Winter image



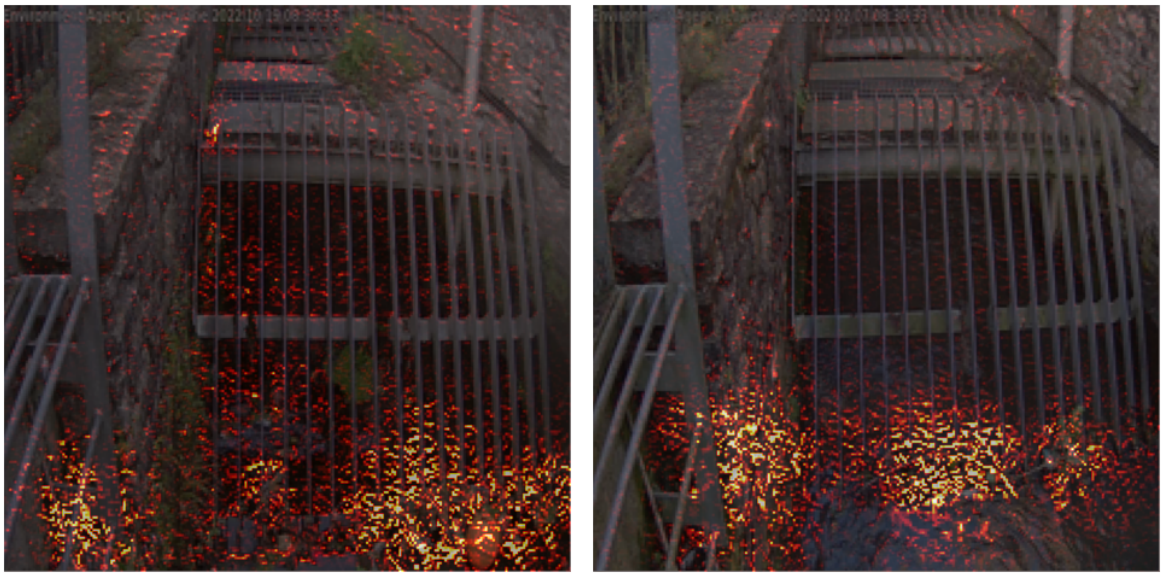
(c) Spring image

Figure 5.6: Integrated Gradients: Winter Model

Autumn Image 5.6a: When the winter model is applied to the autumn image, the integrated gradients indicate a focus on certain textures in the lower part of the image, albeit less concentrated than in the winter image. There is some response above the trash screen. **Winter Image** 5.6b: The winter model shows a robust response when applied to the winter image, with gradients concentrated in the central and lower sections of the image. **Spring**

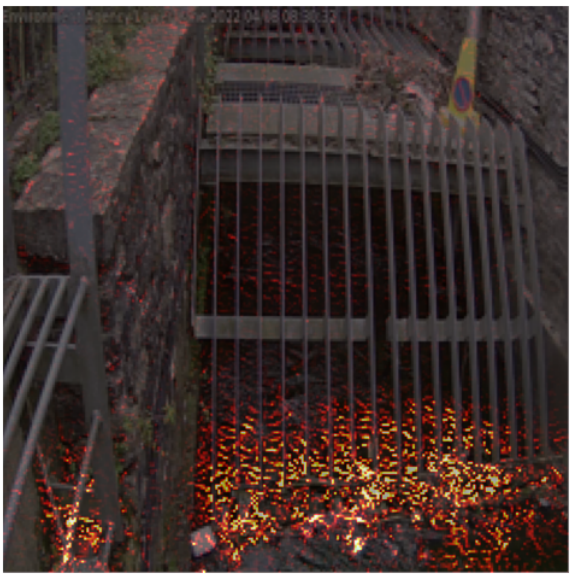
Image 5.6c: The response of the winter model to the spring image is the least intense, with gradients being more dispersed. It is worth noting that, unlike in Figure 5.5c, there is not as much focus on the unnecessary dry debris located above the trash screen.

Spring Model



(a) Autumn image

(b) Winter image



(c) Spring image

Figure 5.7: Integrated Gradients: Spring Model

Autumn Image 5.7a: The integrated gradients for the autumn image appear scattered, with most of the focus on the lower half of the image. Unlike the previous models, there is considerably less focus on the left half of the trash screen, but there is some focus on the left side wall. **Winter Image** 5.7b: For the winter image, the gradients are generally concentrated on the lower half of the image, but with significant focus on the left wall. **Spring Image** 5.7c:

The spring image displays the most concentrated gradients for the spring model. The focus is strongly on the base of the trash screen, and it does not focus on the leaves that have passed through the trash screen. A consistent focus on the left side wall is observed with this model.

All three models appear to perform well, with the majority of the focus concentrated at the base of the trash screen. As observed previously, most of the attention is directed towards sticks and debris.

5.2.3 Occlusion Sensitivity

Occlusion sensitivity generalises saliency mapping by evaluating different regions instead of individual pixels. This technique is useful for understanding broader areas of importance, rather than just specific features.

Autumn Model

Autumn Image 5.8a: In the autumn model (Figure 5.8), varied results can be observed depending on the image. In Figure 5.8a, the model performs well, with strong focus at the base of the trash screen. Unlike Figure 5.2a and Figure 5.5a, there isn't much focus on areas behind the trash screen. This suggests that the features behind the trash screen might be smaller and less influential, as they do not appear prominently in the occlusion sensitivity map. **Winter Image** 5.8b: There appears to be an unusual focus on the upper right side of the image, which is unexplained. However, there is still focus at the base of the trash screen. **Spring Image** 5.8c: The model appears to perform well, with most of the focus on the blockage, though there is also some attention on the debris above the trash screen and the left side wall.

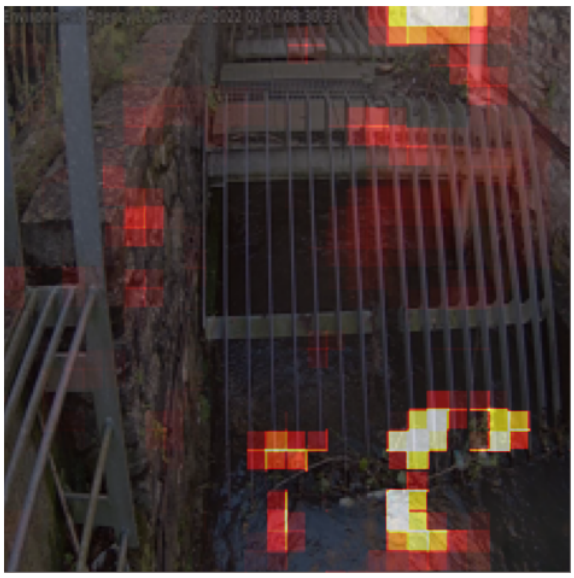
Winter Model

Autumn Image 5.9a: The model shows more focus on the left wall than previously seen, with some attention on the blockage. **Winter Image** 5.9b: Here, the model functions well, with a strong concentration around the blockage. Other parts of the image have little to no influence. **Spring Image** 5.9c: Again, there is focus around the blockage, indicating that the model is functioning well. The focus is less concentrated than in Figure 5.9b, and there is minimal focus on the left side wall.

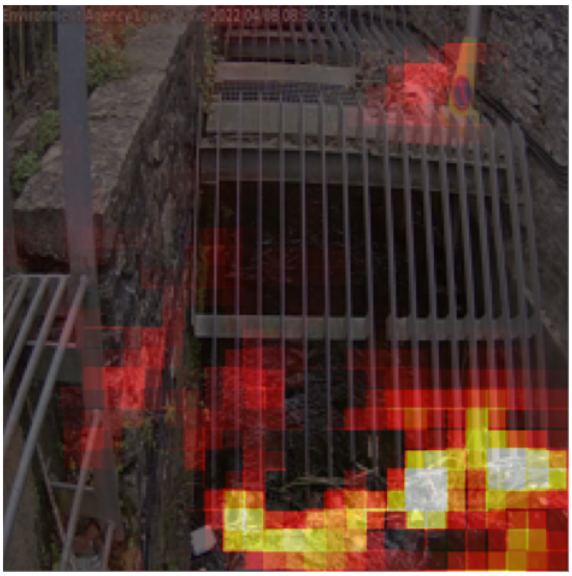
Spring Model

Autumn Image 5.10a: The model shows influence throughout the image, suggesting that the debris behind the trash screen may have more impact than initially thought. This method indicates that if those patches behind the trash screen were not present, it would influence the model's decision, though not significantly. There also appears to be focus on parts of the blockage, rather than the blockage as a whole. **Winter Image** 5.10b: The winter image appears to be performing well, with the focus on the blockage. As noted in Figure 5.4b and Figure 5.7b, there is some focus on the left side wall. **Spring Image** 5.10c: The model works best here, with a strong focus at the base of the image. Unlike in Figure 5.7c, there is much less focus on the left side wall, which may suggest that the feature is small and minimally influences the output.

(a) Autumn image

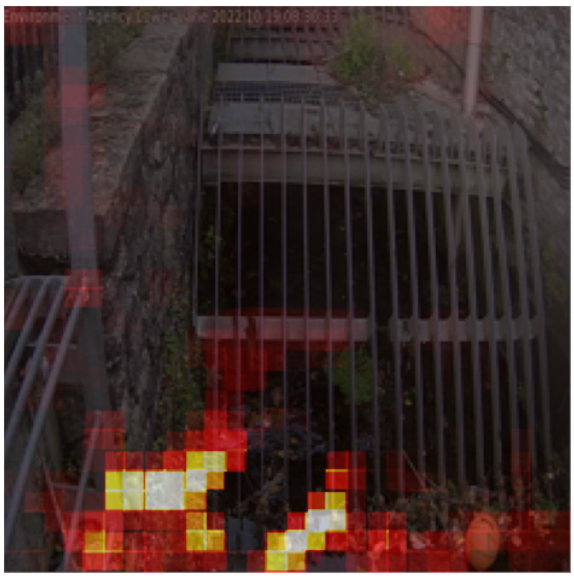


(b) Winter image



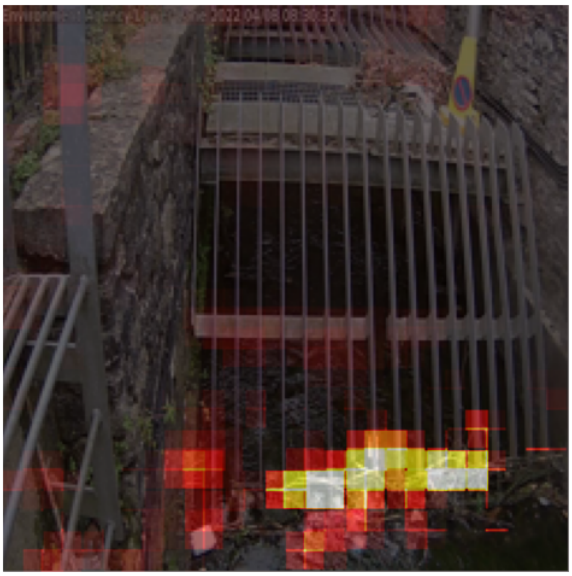
(c) Spring image

Figure 5.8: Occlusion Sensitivity: Autumn Model



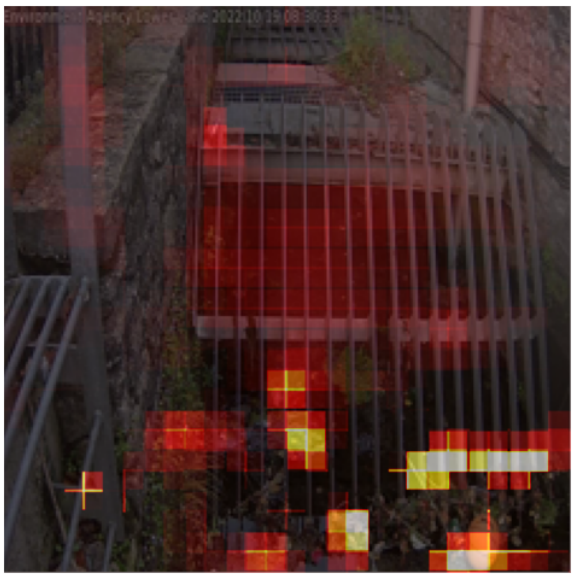
(a) Autumn image

(b) Winter image



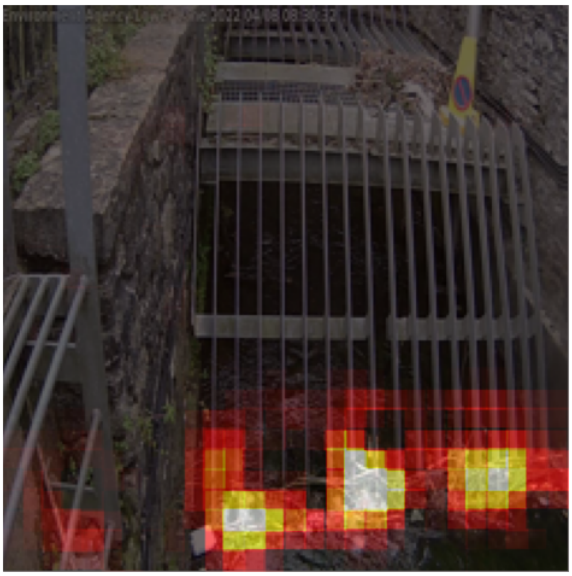
(c) Spring image

Figure 5.9: Occlusion Sensitivity: Winter Model



(a) Autumn image

(b) Winter image



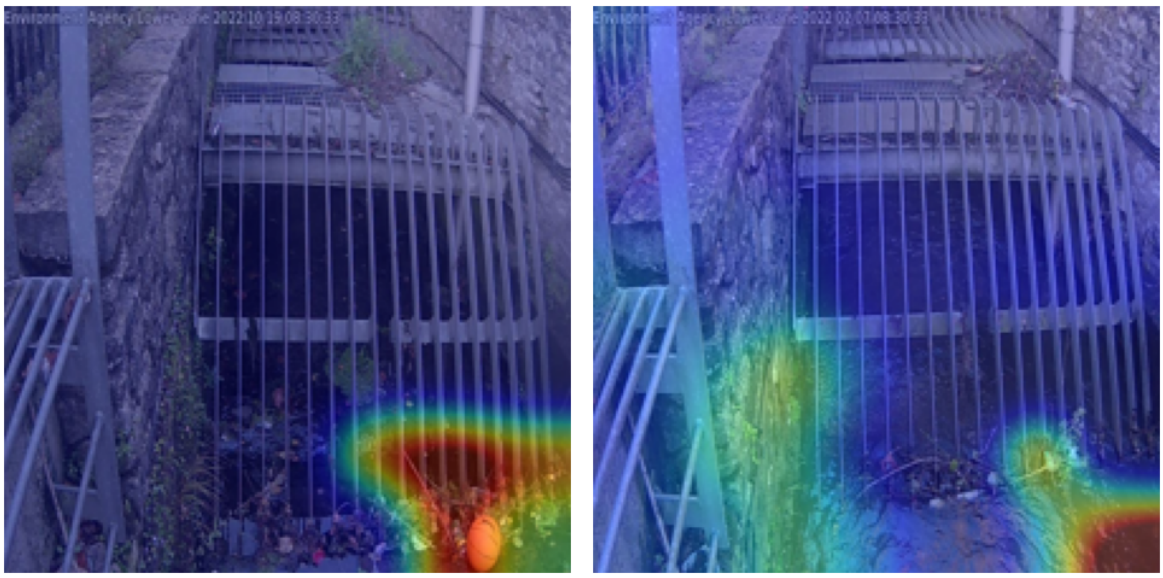
(c) Spring image

Figure 5.10: Occlusion Sensitivity: Spring Model

5.2.4 Smooth Gradient Class Activation Mapping++ (SmoothGrad-CAM++)

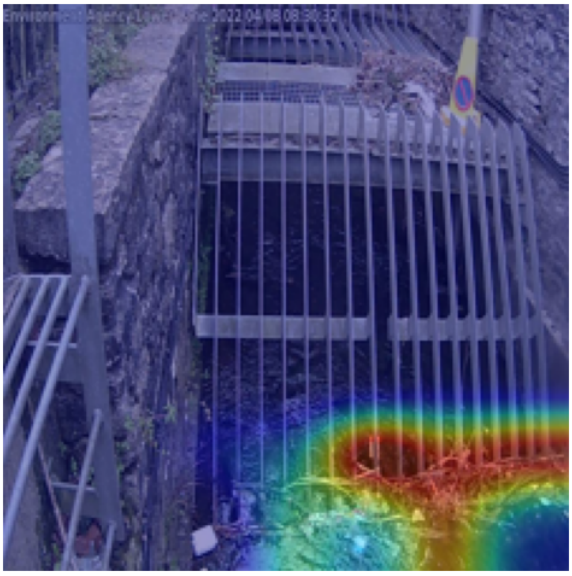
SmoothGrad-CAM++ highlights areas in an image that are important for the network's decision by analysing how the gradients (changes) in the output class score relate to different parts of the image, thus highlighting the spatial region of focus.

Autumn Model



(a) Autumn image

(b) Winter image



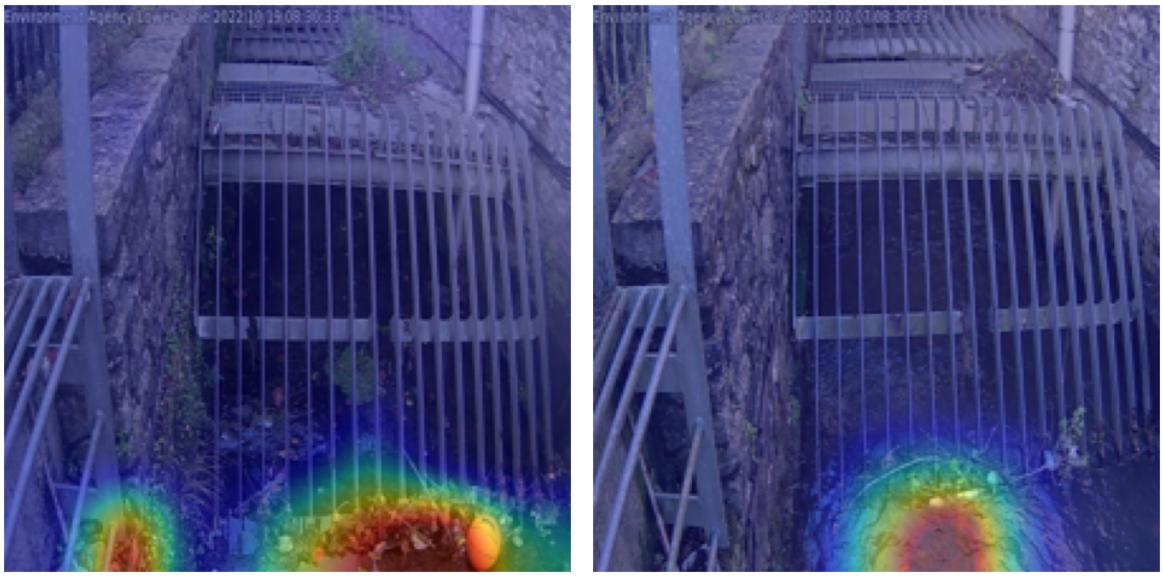
(c) Spring image

Figure 5.11: SmoothGrad-CAM++: Autumn Model

Autumn Image 5.11a: The model strongly focuses on a region in the bottom right. This result aligns with the location of the debris, illustrating that the model is working effectively.

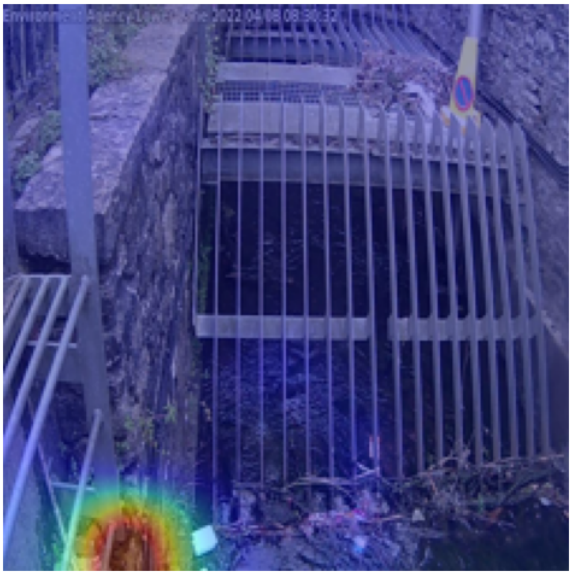
Winter Image 5.11b: The focus appears to be on the left side of the wall and on the water in the lower right part of the image. This result is inconsistent with previous findings from this model. The focal point is on the water with some influence from the surrounding area.
Spring Image 5.11c: This image demonstrates the technique working best, with the focus covering the debris and resembling a similar 'T' shape to that in Figure 5.11a. There is also a focus on the water flowing into the trash screen.

Winter Model



(a) Autumn image

(b) Winter image



(c) Spring image

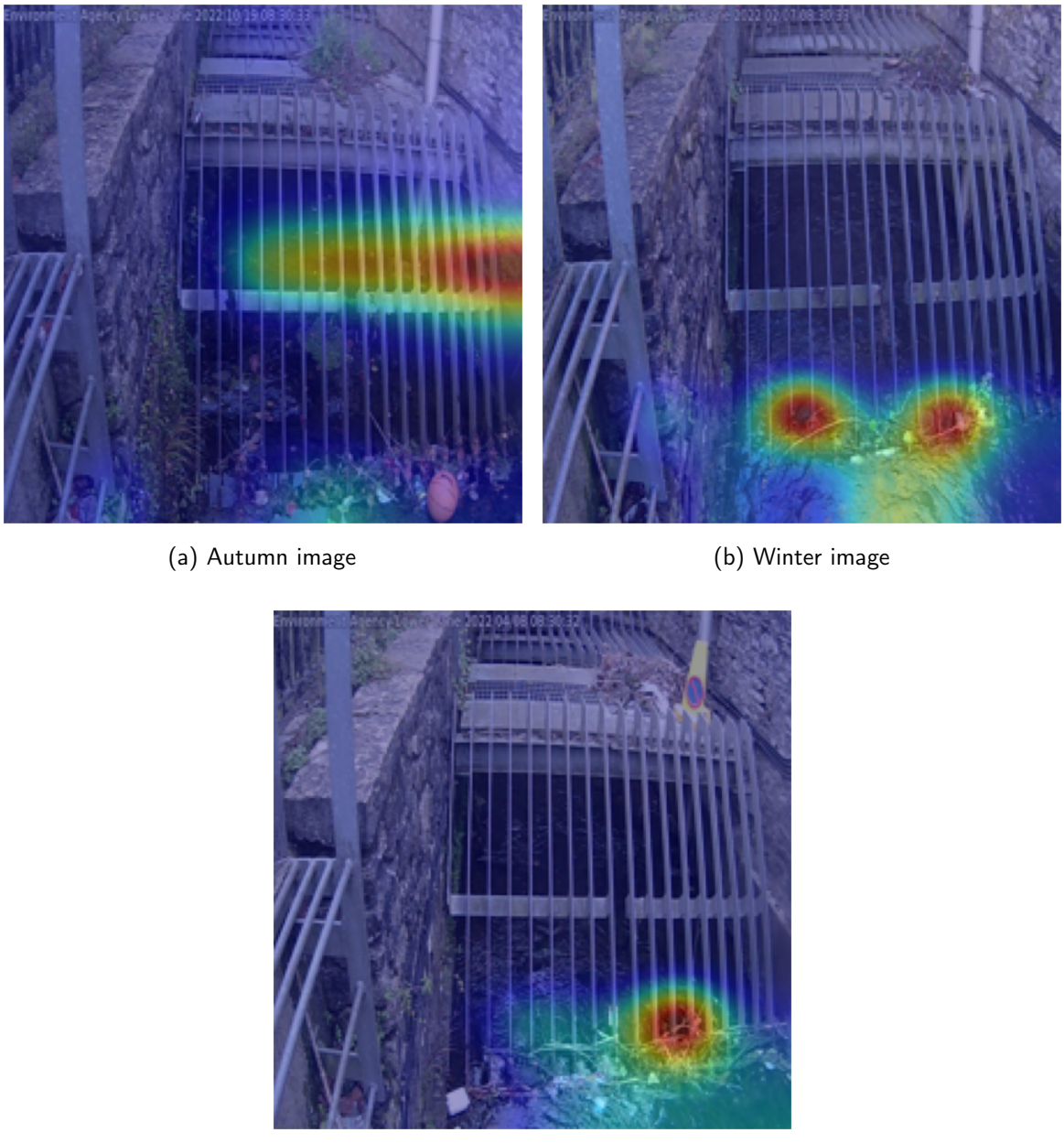
Figure 5.12: SmoothGrad-CAM++: Winter Model

The results from the winter model appear to focus on the lower part of the images.

Autumn Image 5.12a: The model seems to be working effectively with focus at the base

of the trash screen, including the area covered by the water. There is also focus on the wall. **Winter Image** 5.12b: The focus does not seem to be on the blockage, but rather around the base of the trash screen and the water leading into the trash screen. **Spring Image** 5.12c: Here, there is a strong concentration on the left side of the wall with very little influence around the actual blockage.

Spring Model



(a) Autumn image

(b) Winter image

(c) Spring image

Figure 5.13: SmoothGrad-CAM++: Spring Model

Autumn Image 5.13a: The result is somewhat unusual, with the focus halfway up the trash screen. Considering the results from the occlusion sensitivity in Figure 5.10a, this is not entirely surprising. The concentration appears to be more on the right-hand side. **Winter Image**

5.13b: For this image, the model appears to be functioning correctly, with the concentration around the blockage. There are two focal points with some focus on the water. **Spring Image** 5.13c: This image also appears to be functioning correctly, with a focus on the lower part of the trash screen and some influence from the water flowing into the trash screen.

Chapter 6

Discussion

6.1 Summary of Findings

6.1.1 Binary Classifier

The initial phase of this research involved constructing a functional binary classifier to train on subsets of data. The classifier demonstrated strong performance, achieving an accuracy of 95% on the test set. This level of accuracy was deemed sufficient to advance to the subsequent stages of the study.

6.1.2 Seasonal Classifier

Building on the established binary classifier, seasonal models were developed by dividing the data into balanced datasets for each season, excluding images categorised as 'Other'. During the pre-processing stage, it was noted that there were only 66 images for 'Summer', which was considered insufficient for developing a robust model. Consequently, models were developed only for 'Autumn', 'Winter', and 'Spring'.

Some data drift was observed compared to the original binary classifier, as evidenced by the superior performance of season-specific models on the test set for their respective seasons. This finding is significant, offering valuable insights for further investigation.

These results highlight that each model's performance is heavily influenced by the seasonality of the data on which it was trained. While the seasonal models exhibited high accuracy within their respective seasons, their performance declined significantly when applied to data from other seasons. In contrast, the 'all-seasons' model, although not achieving the highest accuracy in Spring and Winter, maintained a more consistent performance across all datasets. This consistency suggests that the all-seasons model has broader applicability and better generalisation capability, making it more robust in scenarios involving multi-seasonal data or where season-specific patterns are less predictable.

6.1.3 Explainability in the Seasonal Classifier

Four different methods were implemented to understand the inner workings of each model, aiming to identify the differences in their performance and explain the varying results.

All three models demonstrated some degree of focus on the actual blockage. This outcome is encouraging as it indicates that, to some extent, the models are correctly detecting the elements in the images associated with blockages.

The key question then arises: where do the models go wrong? Several general themes were identified during the analysis. These include the models' focus on the left-side wall, debris, and the leaves that had passed through the trash screen in the Autumn image.

The left-side wall emerged as a common point of focus across the models, though it is not entirely clear why this occurs. The consistency of vegetation shown or the interaction of smaller vegetation with larger artificial features (resembling blockages) may draw the models' attention. The results are inconclusive, but this misdirected attention detracts from the identification of the actual blockage and should be addressed in future iterations of the model.

Another noteworthy observation is the presence of a traffic cone in the spring image. Across all models and methods, the cone did not appear to significantly influence the models' decisions. This suggests that when debris near the cone affects classification, it is not due to the cone's proximity to the trash screen. Rather, the debris above the screen seems to influence the models based on its shape. In the autumn image, a living plant appeared in the same location as dry debris in the winter and spring images, but it was less influential than the dry debris. This suggests that colour may play a role in the models' decision-making. If shape alone were the determining factor, the plant would have similarly confused the models. This was particularly evident in the autumn model, as shown in Figure 5.2, Figure 5.5, and Figure 5.8.

The presence of leaves passing through the trash screen, as observed in the autumn image, introduced confusion for the models. Since the leaves are transient, flowing away from the camera as they pass the trash screen, this temporary disruption impacts the model's performance. This phenomenon, most notable in Figure 5.13a, is significant in the context of practical implementation. The explainability techniques highlight that temporary confusing data, such as the movement of leaves, underscores that the models are fulfilling their intended role. However, the models are limited in determining whether such objects will remain in the frame or flow away, given that the images represent singular snapshots in time.

Given that the images are captured sequentially, it may be worth exploring methods that consider multiple images to mitigate such temporary confusion. For example, incorporating information from images captured before and after the specific frame in question may reduce the uncertainty caused by transient objects like the leaves.

A final key takeaway is that the models are not perfect. Despite working with seemingly similar images, there is significant variation in the models' perception, warranting further research into their performance and limitations.

6.2 Future Research Directions and Implementation

The most significant insight from this research is the seasonal differences observed in each model. This suggests that if a model is trained solely on images from a particular season, continuous re-evaluation and retraining will be necessary as more images become available.

A second avenue for future research is image cropping. Given the challenges posed by the left-side wall and debris above the trash screen, cropping may be beneficial. However, it is

argued that cropping should not be restricted solely to the trash screen; as demonstrated in subsection 5.2.4, attention should also be given to the water in front of the screen.

An additional area for development is the implementation of such a product in real-world settings. The models' high level of accuracy and the demonstrated focus on relevant areas suggest efforts should shift towards real-world deployment of this approach. As with all machine learning methods, improvements will arise over time with new techniques and architectures. However, the current explainability results indicate that we are at a stage where deployment is feasible, with opportunities to refine and update the models as needed through future research.

6.3 Limitations of the Study

6.3.1 Assumptions

A key assumption made in this study is that each season's data came from the same sites. This assumption was necessary to ensure a sufficient number of images for training. As highlighted in Figure 4.1, the proportion of blocked to clear images varied between seasons. Consequently, some sites may have been excluded from certain seasons if they did not contain any blocked images. Additionally, it was assumed that there were enough overlapping sites across seasons to mitigate this variability.

6.3.2 Time Constraints

This research could have taken several different directions, but due to time constraints, the study primarily focused on explainability. There is, however, room for exploring additional explainability techniques. Furthermore, the backbone used in this study, ResNet-50, was chosen based on its success in previous research. Future work could investigate other backbones to determine which pre-trained network performs best.

Chapter 7

Conclusions

This study set out to explore the challenges posed by seasonal data drift in the application of computer vision models for detecting blockages in culverts. Through the development and analysis of both a binary classifier and season-specific models, the research has highlighted the significant impact that seasonal variations can have on model performance. The findings suggest that while a generalised model may offer consistent performance across multiple seasons, season-specific models tend to outperform the general model when applied to their respective datasets. This highlights the importance of considering seasonal factors in the development and deployment of machine learning models for environmental monitoring.

The use of explainability techniques has provided valuable insights into the decision-making processes of the models, revealing both their strengths and areas where improvements are needed. The models were generally effective at focusing on the relevant features of the images, such as debris at the base of the trash screen, which is critical for accurate blockage detection. However, the occasional focus on irrelevant features, such as the left-side wall or debris located above the screen, indicates that there is still room for refinement.

The implications of this research are clear: to maintain high levels of accuracy and reliability in culvert monitoring systems, it is essential to continuously re-evaluate and retrain models as environmental conditions evolve. Future research should investigate seasonal explainability using image cropping that includes water, explore alternative model backbone architectures, and analyse sequentially dependent images. Additionally, the demonstrated success of computer vision techniques in flood prediction suggests these methods are ready for further development and deployment in real-world scenarios, thereby enhancing their practical utility in flood prevention.

In conclusion, this research has demonstrated the critical importance of addressing seasonal data drift in the application of machine learning to flood prevention. By deepening our understanding of how these models operate and how they are affected by environmental changes, we can develop more reliable and effective solutions for protecting infrastructure and communities from the increasing threats posed by extreme weather events.

Number of words until this point, excluding front matter: 10314.

Bibliography

Barredo Arrieta, A., Díaz-Rodríguez, N., Del Ser, J., Bennetot, A., Tabik, S., Barbado, A., Garcia, S., Gil-Lopez, S., Molina, D., Benjamins, R., Chatila, R. and Herrera, F., 2020. Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. *Information fusion* [Online], 58, pp.82–115. Available from: https://doi.org/https://doi.org/10.1016/j.inffus.2019.12.012.

Blanc, J., Wallerstein, N.P., Arthur, S. and Wright, G.B., 2014. Analysis of the performance of debris screens at culverts. *Proceedings of the institution of civil engineers - water management* [Online], 167(4), pp.219–229. https://doi.org/10.1680/wama.12.00063, Available from: https://doi.org/10.1680/wama.12.00063.

Chattopadhay, A., Sarkar, A., Howlader, P. and Balasubramanian, V.N., 2018. Grad-cam++: Generalized gradient-based visual explanations for deep convolutional networks [Online]. *2018 ieee winter conference on applications of computer vision (wacv)*. IEEE. Available from: https://doi.org/10.1109/wacv.2018.00097.

Clark, A., 2015. *Pillow (pil fork) documentation*. readthedocs. Available from: https://buildmedia.readthedocs.org/media/pdf/pillow/latest/pillow.pdf.

Cornelius Smith, R., Barnes, A., Wang, J., Dooley, S., Rowlatt, C. and Kjeldsen, T., 2023. Cctv image-based classification of blocked trash screens. *Journal of flood risk management*.

Disabato, S. and Roveri, M., 2019. Learning convolutional neural networks in presence of concept drift [Online]. *2019 international joint conference on neural networks (ijcnn)*. pp.1–8. Available from: https://doi.org/10.1109/IJCNN.2019.8851731.

Fernandez, F.G., 2020. *Torchcam: class activation explorer*. GitHub.

Ghosh, A., Sufian, A., Sultana, F., Chakrabarti, A. and De, D., 2020. *Fundamental concepts of convolutional neural network* [Online], pp.519–567. Available from: https://doi.org/10.1007/978-3-030-32644-9_36.

Goel, P. and Ganatra, A., 2023. Unsupervised domain adaptation for image classification and object detection using guided transfer learning approach and js divergence. *Sensors* [Online], 23(9). Available from: https://doi.org/10.3390/s23094436.

Haar, L.V., Elvira, T. and Ochoa, O., 2023. An analysis of explainability methods for convolutional neural networks. *Engineering applications of artificial intelligence* [Online], 117, p.105606. Available from: https://doi.org/https://doi.org/10.1016/j.engappai.2022.105606.

Harris, C.R., Millman, K.J., Walt, S.J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J., Kern, R., Picus, M., Hoyer, S., Kerkwijk,

M.H. van, Brett, M., Haldane, A., Río, J.F. del, Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C. and Oliphant, T.E., 2020. *Array programming with NumPy*. 7825. Available from: https://doi.org/10.1038/s41586-020-2649-2.

He, K., Zhang, X., Ren, S. and Sun, J., 2015. Deep residual learning for image recognition [Online]. 1512.03385, Available from: https://arxiv.org/abs/1512.03385.

Holzinger, A., Biemann, C., Pattichis, C.S. and Kell, D.B., 2017. What do we need to build explainable ai systems for the medical domain? [Online]. 1712.09923, Available from: https://arxiv.org/abs/1712.09923.

Hunter, J.D., 2007. *Matplotlib: A 2d graphics environment*. 3. Available from: https://doi.org/10.1109/MCSE.2007.55.

Iqbal, U., Barthélemy, J. and Perez, P., 2022. Prediction of hydraulic blockage at culverts from a single image using deep learning. *Neural computing and applications* [Online], 34, pp.1–17. Available from: https://doi.org/10.1007/s00521-022-07593-8.

Iqbal, U., Barthélemy, J. and Perez, P., 2023. Visual blockage assessment at culverts using synthetic images to mitigate blockage-originated floods. *Journal of hydroinformatics* [Online], 25. Available from: https://doi.org/10.2166/hydro.2023.068.

Iqbal, U., Riaz, M.Z.B., Barthelemy, J., Hutchison, N. and Perez, P., 2022. Floodborne objects type recognition using computer vision to mitigate blockage originated floods. *Water* [Online], 14(17). Available from: https://doi.org/10.3390/w14172605.

Kingma, D.P. and Ba, J., 2017. Adam: A method for stochastic optimization [Online]. 1412.6980, Available from: https://arxiv.org/abs/1412.6980.

Kokhlikyan, N., Miglani, V., Martin, M., Wang, E., Reynolds, J., Melnikov, A., Lunova, N. and Reblitz-Richardson, O., 2019. *Pytorch captum*. GitHub.

Kore, A., Abbasi Bavil, E., Subasri, V., Abdalla, M., Fine, B., Dolatabadi, E. and Abdalla, M., 2024. Empirical data drift detection experiments on real-world medical imaging data. *Nature communications* [Online], 15(1), p.1887. Available from: https://doi.org/10.1038/s41467-024-46142-w.

Lemley, J., Bazrafkan, S. and Corcoran, P., 2017. Transfer learning of temporal information for driver action classification. *Conference: 28th modern artificial intelligence and cognitive science conference*.

maintainers, T. and contributors, 2016. *Torchvision: Pytorch's computer vision library*. GitHub.

McKinney Wes, 2010. *Data Structures for Statistical Computing in Python*. Available from: https://doi.org/10.25080/Majora-92bf1922-00a.

Miller, J.D. and Hutchins, M., 2017. The impacts of urbanisation and climate change on urban flooding and urban water quality: A review of the evidence concerning the united kingdom. *Journal of hydrology: Regional studies* [Online], 12, pp.345–362. Available from: https://doi.org/https://doi.org/10.1016/j.ejrh.2017.06.006.

Mohammadi, P., Sherafat, B. and Rashidi, A., 2023. A risk-based framework for optimizing inspection planning of utah culverts. *United states. department of transportation* [Online]. Available from: https://rosap.ntl.bts.gov/view/dot/68528.

Omeiza, D., Speakman, S., Cintas, C. and Weldermariam, K., 2019. Smooth grad-cam++: An enhanced inference level visualization technique for deep convolutional neural network models [Online]. 1908.01224, Available from: https://arxiv.org/abs/1908.01224.

O'Shea, K. and Nash, R., 2015. An introduction to convolutional neural networks. *Corr* [Online], abs/1511.08458. 1511.08458, Available from: http://arxiv.org/abs/1511.08458.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. and Chintala, S., 2019. *Pytorch: An imperative style, high-performance deep learning library*. Curran Associates, Inc. Available from: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V. et al., 2011. *Scikit-learn: Machine learning in python*. Oct.

Phung, V.H. and Rhee, E.J., 2019. A high-accuracy model average ensemble of convolutional neural networks for classification of cloud image patches on small datasets. *Applied sciences* [Online], 9, p.4500. Available from: https://doi.org/10.3390/app9214500.

Roschewitz, M., Khara, G., Yearsley, J., Sharma, N., James, J.J., Ambrózay, É., Heroux, A., Kecskemethy, P., Rijken, T. and Glocker, B., 2023. Automatic correction of performance drift under acquisition shift in medical image classification. *Nature communications* [Online], 14(1), p.6608. Available from: https://doi.org/10.1038/s41467-023-42396-y.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C. and Fei-Fei, L., 2015. Imagenet large scale visual recognition challenge. *International journal of computer vision* [Online], 115(3), pp.211–252. Available from: https://doi.org/10.1007/s11263-015-0816-y.

Senarathna, D., Tragoudas, S., Gowda, K.N. and Schmit, M., 2023. Detection and quantization of data drift in image classification neural networks [Online]. *2023 ieee 24th international conference on high performance switching and routing (hpsr)*. pp.38–42. Available from: https://doi.org/10.1109/HPSR57248.2023.10147972.

Simonyan, K., Vedaldi, A. and Zisserman, A., 2014. Deep inside convolutional networks: Visualising image classification models and saliency maps [Online]. 1312.6034, Available from: https://arxiv.org/abs/1312.6034.

Smilkov, D., Thorat, N., Kim, B., Viégas, F. and Wattenberg, M., 2017. Smoothgrad: removing noise by adding noise [Online]. 1706.03825, Available from: https://arxiv.org/abs/1706.03825.

Streftaris, G., Wallerstein, N., Gibson, G. and Arthur, S., 2013. Modeling probability of blockage at culvert trash screens using bayesian approach. *Journal of hydraulic engineering* [Online], 139, pp.716–726. Available from: https://doi.org/10.1061/(ASCE)HY.1943-7900.0000723.

Sundararajan, M., Taly, A. and Yan, Q., 2017. Axiomatic attribution for deep networks [Online]. 1703.01365, Available from: https://arxiv.org/abs/1703.01365.

Vandaele, R., 2023. Trash screen blockage detection using cameras and deep learning: code and dataset [Online]. Available from: https://researchdata.reading.ac.uk/498/.

Vandaele, R., Dance, S.L. and Ojha, V., 2024. Deep learning for automated trash screen blockage detection using cameras: Actionable information for flood risk management. *Journal of hydroinformatics* [Online], 26(4), pp.889–903. https://iwaponline.com/jh/article-pdf/26/4/889/1407447/jh0260889.pdf, Available from: https://doi.org/10.2166/hydro.2024.013.

Waskom, M.L., 2021. *seaborn: statistical data visualization*. 60. Available from: https://doi.org/10.21105/joss.03021.

Yamashita, R., Nishio, M., Do, R.K.G. and Togashi, K., 2018. Convolutional neural networks: an overview and application in radiology. *Insights into imaging* [Online], 9(4), pp.611–629. Available from: https://doi.org/10.1007/s13244-018-0639-9.

Yazdi, J., 2018. Improving urban drainage systems resiliency against unexpected blockages: A probabilistic approach. *Water resources management* [Online], 32. Available from: https://doi.org/10.1007/s11269-018-2069-3.

Zeiler, M.D. and Fergus, R., 2013. Visualizing and understanding convolutional networks [Online]. 1311.2901, Available from: https://arxiv.org/abs/1311.2901.

Zhang, Z., Duan, F., Solé-Casals, J., Dinarès-Ferran, J., Cichocki, A., Yang, Z. and Sun, Z., 2019. A novel deep learning approach with data augmentation to classify motor imagery signals. *Ieee access* [Online], 7, pp.15945–15954. Available from: https://doi.org/10.1109/ACCESS.2019.2895133.

Appendix A

Design Diagrams

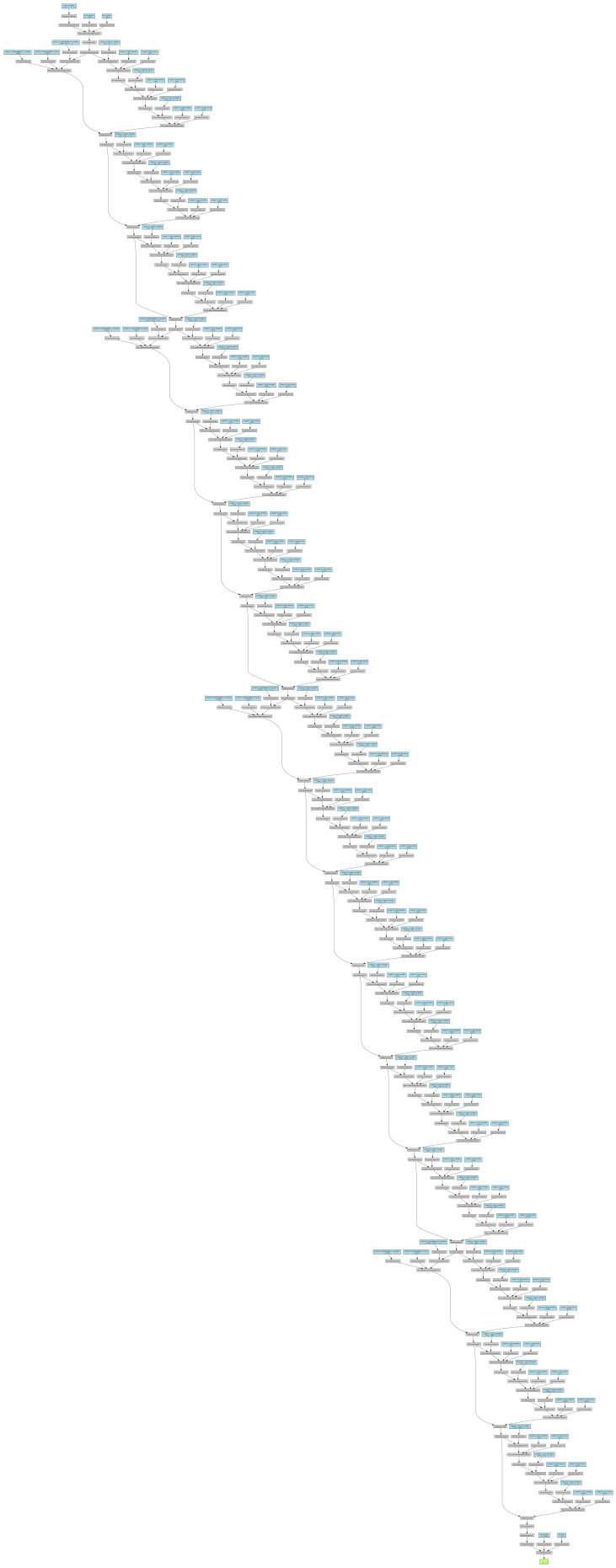


Figure A.1: Diagram of the overall model architecture, showcasing the ResNet-50 backbone and its integration into a binary classification output.

Appendix B

User Documentation

Each script within this directory can be executed independently, provided that there exists a 'Data' folder within the same directory. This 'Data' folder should contain the 'blockagedetection_dataset' as sourced from Vandaele (2023).

- **eda.py**: This script performs a straightforward exploratory data analysis, generating a visualisation of the entire dataset. It is designed to be easily customisable to include summary statistics or other modifications as required.
- **seasonal_data_split.py**: This script is responsible for splitting the dataset by season and balancing it according to site. It is a foundational script that is utilised by all other scripts in this project, and hence, must be located in the same directory as the others.
- **train_seasonal.py**: This script handles the training of models on a seasonal basis. Users need to manually select the model type they wish to train. The script includes an option to save the model weights and the best validation accuracy achieved during training.
- **classification_network.py**: Adapted from the script provided by Vandaele (2023), this script is used to load a trained model and classify images from a test set. It outputs a CSV file in the working directory containing all the predictions. This script only needs to be run once to generate results for all models.
- **seasonal_plot.py**: This script utilises the CSV files generated by 'classification_network.py' to create bar plots. These plots are featured in Figure 5.1.
- **saliency_mapping.py**: This script generates saliency maps for visualising model predictions. Users must manually select the classifier to be used.
- **grad-cam.py**: This script produces smoothgrad-CAM visualisations. Similar to the saliency mapping script, the user must manually select the classifier.
- **occlusion.py**: This script generates occlusion sensitivity maps, which help to identify which parts of the input data contribute most to the model's decisions. Again, users must manually select the classifier to use.
- **integrated_gradients.py**: This script creates integrated gradient maps to highlight the features that are most influential in the model's decision-making process. As with the other visualisation scripts, the classifier must be manually selected.

Appendix C

Raw Results Output

C.1 Saliency Mapping

Figure C.1: Additional Saliency Mapping Autumn Model Results

Figure C.2: Additional Saliency Mapping Winter Model Results

Figure C.3: Additional Saliency Mapping Spring Model Results

C.2 Integrated Gradients

Figure C.4: Additional Integrated Gradients Autumn Model Results

Figure C.5: Additional Integrated Gradients Winter Model Results

Figure C.6: Additional Integrated Gradients Spring Model Results

C.3 Occlusion Sensitivity

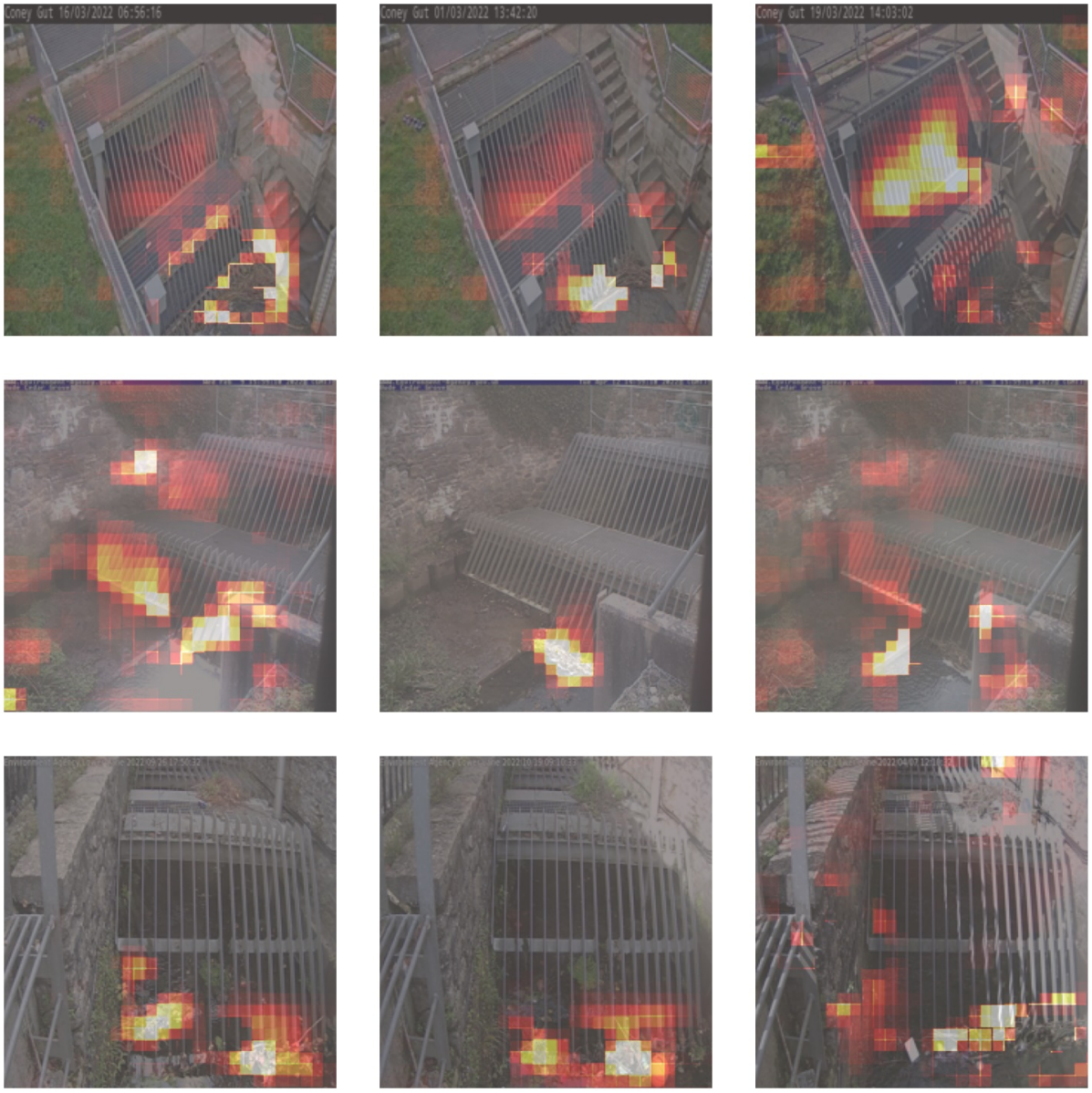


Figure C.7: Additional Occlusion Sensitivity Autumn Model Results

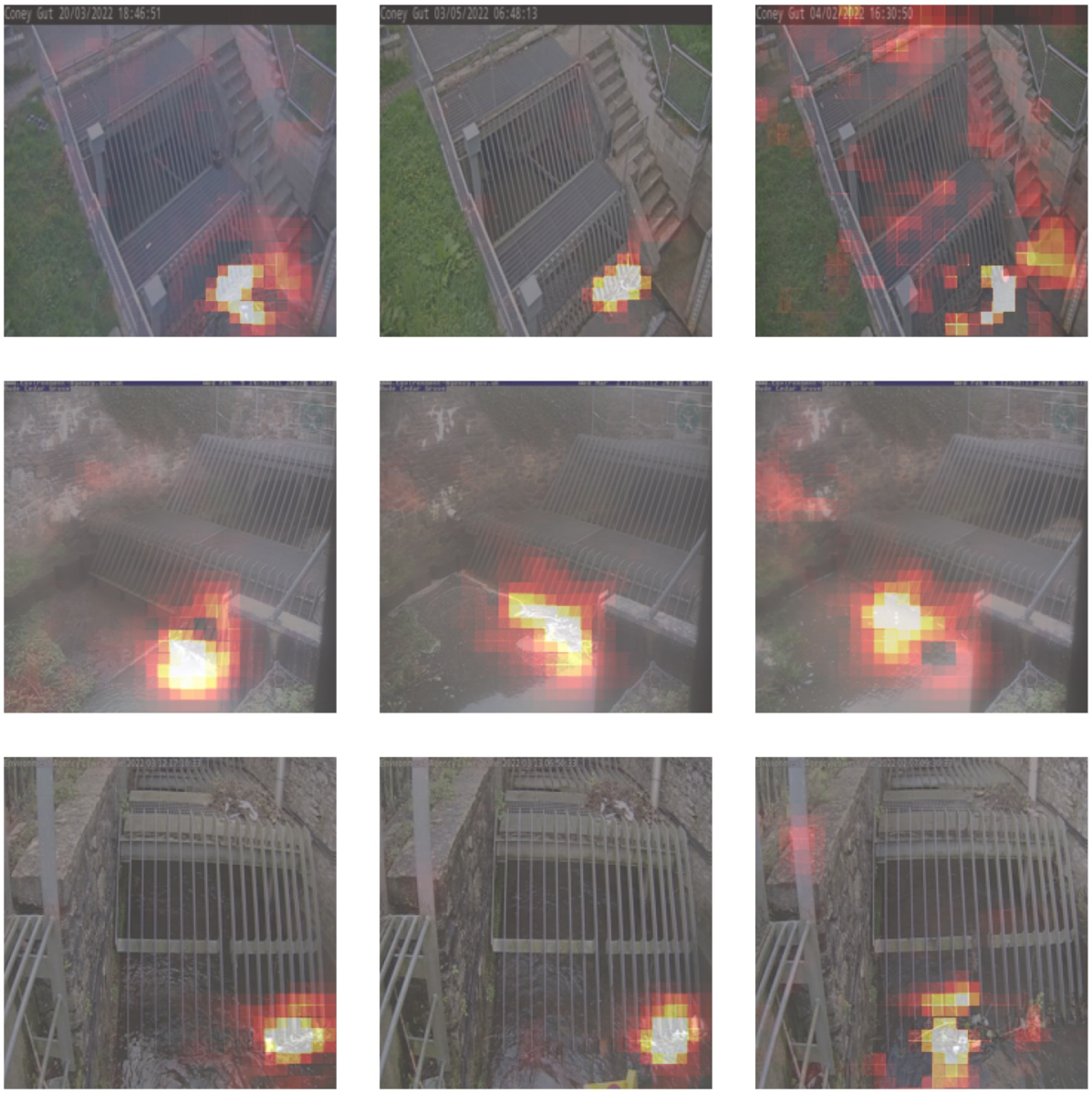


Figure C.8: Additional Occlusion Sensitivity Winter Model Results

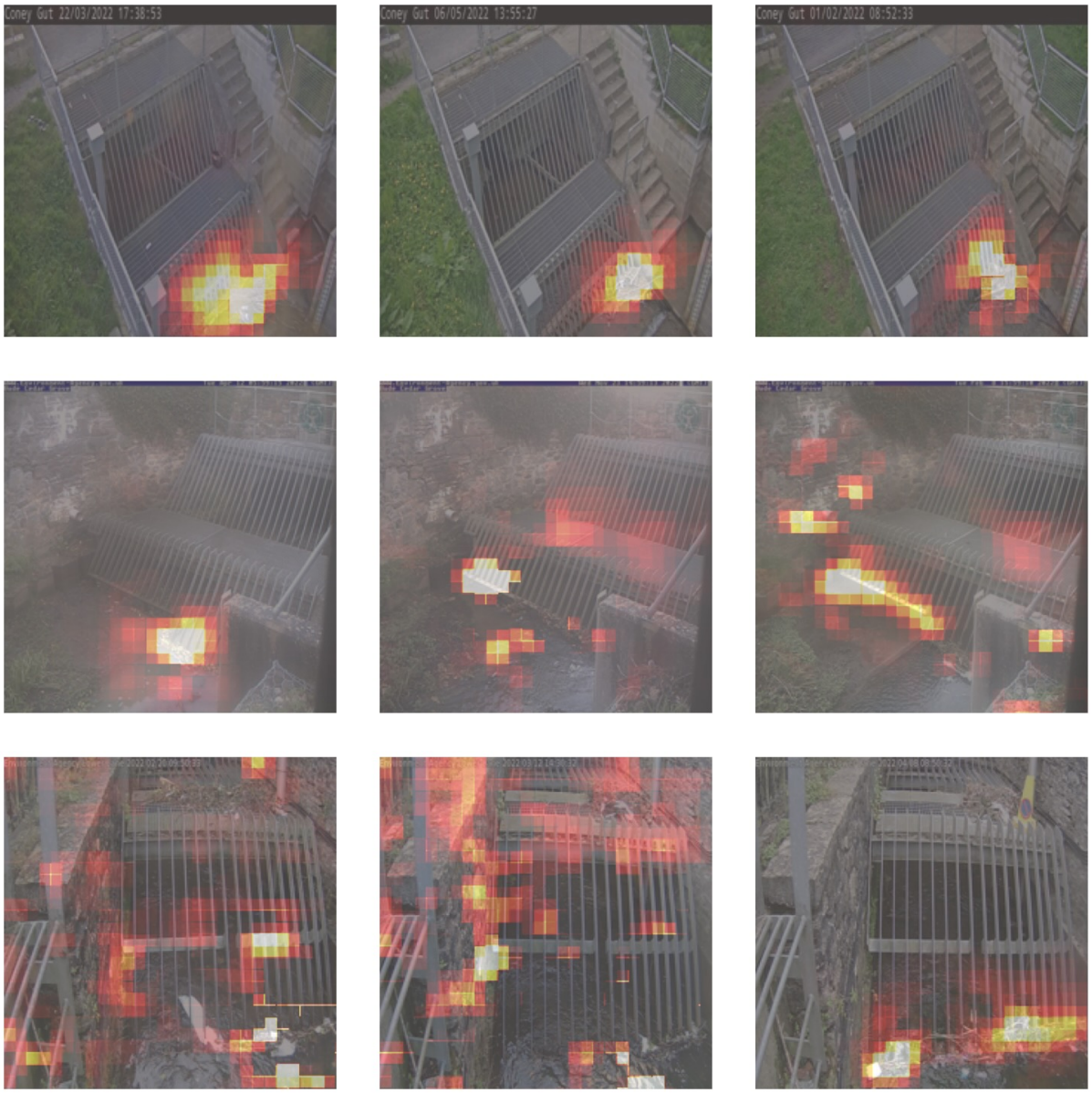


Figure C.9: Additional Occlusion Sensitivity Spring Model Results

C.4 SmoothGrad-CAM++

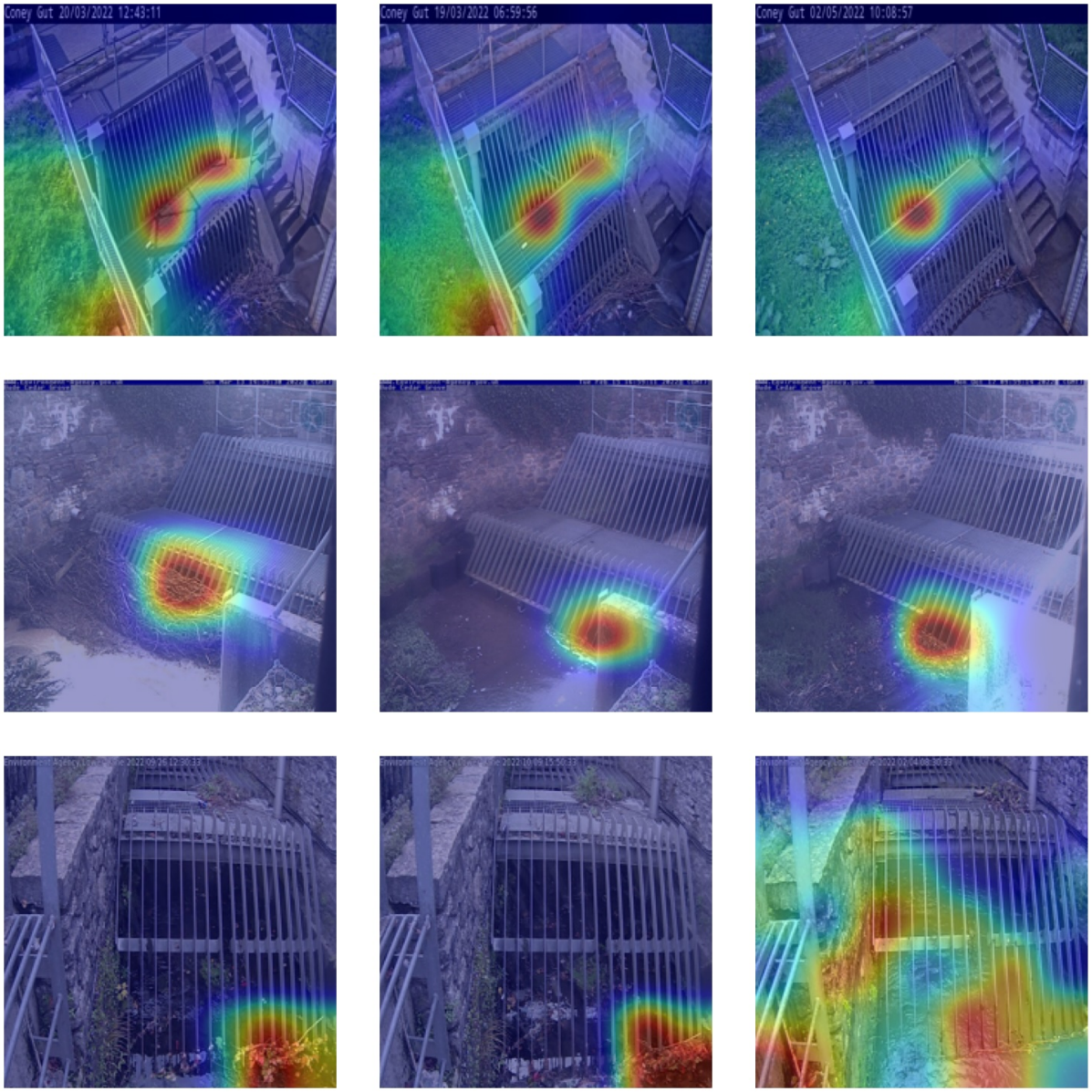


Figure C.10: Additional SmoothGrad-CAM++ Autumn Model Results

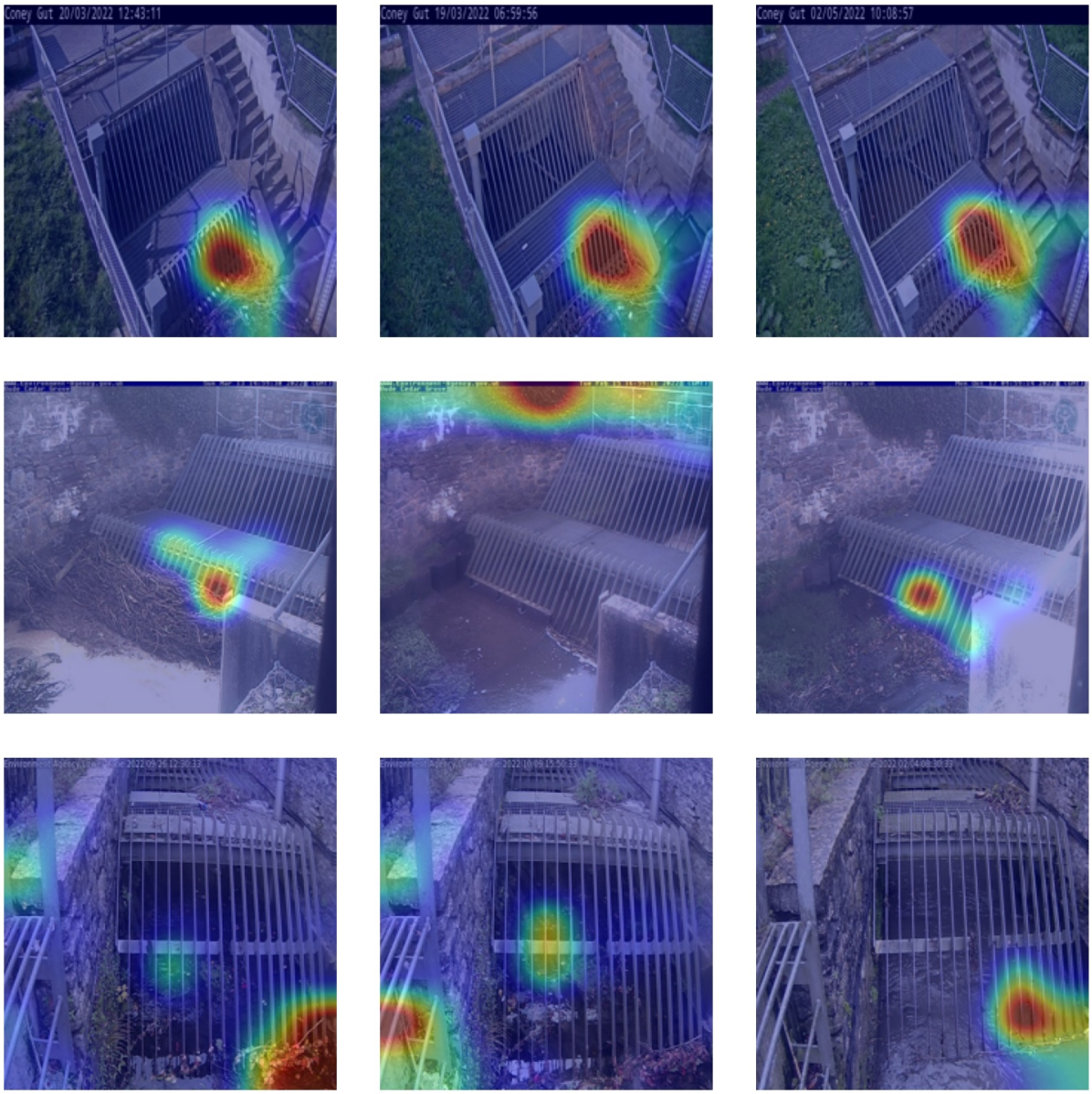


Figure C.11: Additional SmoothGrad-CAM++ Winter Model Results

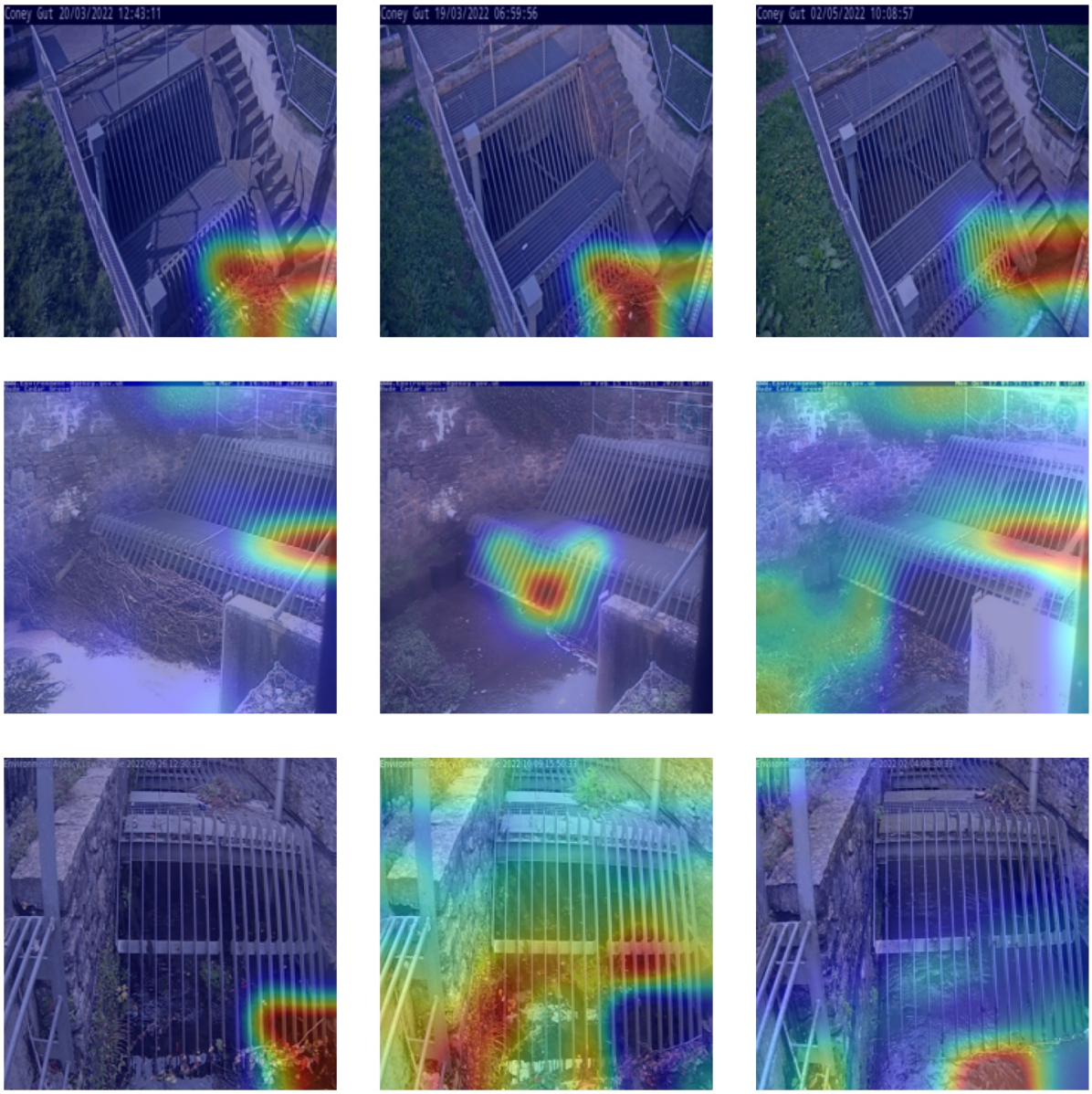


Figure C.12: Additional SmoothGrad-CAM++ Spring Model Results

Appendix D

Code

D.1 File: eda.py

```
import os
import pandas as pd
import matplotlib.pyplot as plt
from datetime import datetime


def label_images(base_dir):
    image_data = []

    # Traverse the base directory to find all site directories
    for site in os.listdir(base_dir):
        site_path = os.path.join(base_dir, site)
        if os.path.isdir(site_path):
            subdirs = ['blocked', 'clear', 'other']

            # Traverse each subdirectory
            for subdir in subdirs:
                subdir_path = os.path.join(site_path, subdir)
                if os.path.exists(subdir_path):
                    for root, _, files in os.walk(subdir_path):
                        for file in files:
                            if file.lower().endswith(('png',
                                'jpg', 'jpeg')):
                                # Full file path
                                file_path = os.path.join(root,
                                    file)
                                image_data.append({
                                    'file_path': file_path,
                                    'site': site,
                                    'label': subdir
                                })

    # Create a DataFrame from the collected data
    df = pd.DataFrame(image_data)
    return df


def assign_season(date):
    year = date.year
    seasons = {
        'Spring': (datetime(year, 3, 20), datetime(year, 6, 21)),
        'Summer': (datetime(year, 6, 21), datetime(year, 9, 23)),
        'Autumn': (datetime(year, 9, 23), datetime(year, 12, 22)),
        'Winter_1': (datetime(year, 1, 1), datetime(year, 3, 20)),
        'Winter_2': (datetime(year, 12, 22), datetime(year, 12,
            31))
    }
    for season, (start, end) in seasons.items():
        if start <= date <= end:
            return 'Winter' if season in ['Winter_1', 'Winter_2']
                else season


# Determine the base directory dynamically
script_dir = os.path.dirname(os.path.abspath(__file__))
base_dir = os.path.join(script_dir,
    'Data/blockagedetection_dataset/images')
df_images = label_images(base_dir)

# Extract the datetime string from the filenames by removing the
    extension
df_images['datetime_str'] = df_images['file_path'].apply(
    lambda x:
        '_'.join(os.path.basename(x).split('.')[0].split('_')[:5])
)

# Convert the datetime string to datetime objects
df_images['date'] = pd.to_datetime(df_images['datetime_str'],
    format='%Y_%m_%d_%H_%M')

# Drop the intermediate datetime string column
df_images.drop(columns=['datetime_str'], inplace=True)

# Apply the function to assign seasons
df_images['season'] = df_images['date'].apply(assign_season)


missing_dates = df_images['date'].isna().sum()
print(f"Number␣of␣images␣with␣missing␣or␣invalid␣dates:␣
    {missing_dates}")

missing_seasons = df_images[df_images['season'].isna()]
print(f"Number␣of␣images␣without␣an␣assigned␣season:␣
    {len(missing_seasons)}")
print(missing_seasons[['site', 'file_path']])

# Group by season and label, then count the images
season_summary = df_images.groupby(['season',
    'label']).size().unstack(fill_value=0)

# Plotting the bar plot with stacked sections for each label
season_summary.plot(kind='bar', stacked=True, figsize=(10, 7),
```

```
    color=['blue', 'green', 'red'])
plt.title('Distribution␣of␣Blocked,␣Clear,␣and␣Other␣Images␣by␣
    Season')
plt.xlabel('Season')
plt.ylabel('Number␣of␣Images')
plt.tight_layout()
plt.savefig(os.path.join(script_dir,
    'plots/season_distribution.png'))
plt.show()

# Print the summary for reference
print("Image␣Distribution␣by␣Season␣and␣Label:")
print(season_summary)
```

D.2 File: seasonal_data_split.py

```
import os
import pandas as pd
import matplotlib.pyplot as plt
from datetime import datetime
from sklearn.model_selection import train_test_split


def label_images(base_dir):
    image_data = []

    # Traverse the base directory to find all site directories
    for site in os.listdir(base_dir):
        site_path = os.path.join(base_dir, site)
        if os.path.isdir(site_path):
            subdirs = ['blocked', 'clear', 'other']

            # Traverse each subdirectory
            for subdir in subdirs:
                subdir_path = os.path.join(site_path, subdir)
                if os.path.exists(subdir_path):
                    for root, _, files in os.walk(subdir_path):
                        for file in files:
                            if file.lower().endswith(('png',
                                'jpg', 'jpeg')):
                                # Full file path
                                file_path = os.path.join(root,
                                    file)
                                image_data.append({
                                    'file_path': file_path,
                                    'site': site,
                                    'label': subdir
                                })

    # Create a DataFrame from the collected data
    df = pd.DataFrame(image_data)
    return df

# Determine the base directory dynamically
script_dir = os.path.dirname(os.path.abspath(__file__))
base_dir = os.path.join(script_dir,
    'Data/blockagedetection_dataset/images')
df_images = label_images(base_dir)

print(len(df_images))

# Filter out the 'other' label
df_filtered = df_images[df_images['label'].isin(['blocked',
    'clear'])]

# Group by site and label, then count the images
summary = df_filtered.groupby(['site',
    'label']).size().unstack(fill_value=0)

# Determine the minimum count between blocked and clear labels for
    each site
summary['balanced'] = summary[['blocked', 'clear']].min(axis=1)

# Extract the datetime string from the filenames by removing the
    extension
df_images['datetime_str'] = df_images['file_path'].apply(
    lambda x:
        '_'.join(os.path.basename(x).split('.')[0].split('_')[:5])
)

# Convert the datetime string to datetime objects
df_images['date'] = pd.to_datetime(df_images['datetime_str'],
    format='%Y_%m_%d_%H_%M')

# Drop the intermediate datetime string column
df_images.drop(columns=['datetime_str'], inplace=True)


def assign_season(date):
    year = date.year
    seasons = {
        'Spring': (datetime(year, 3, 20), datetime(year, 6, 21)),
        'Summer': (datetime(year, 6, 21), datetime(year, 9, 23)),
        'Autumn': (datetime(year, 9, 23), datetime(year, 12, 22)),
        'Winter_1': (datetime(year, 1, 1), datetime(year, 3, 20)),
        'Winter_2': (datetime(year, 12, 22), datetime(year, 12,
            31))
    }
    for season, (start, end) in seasons.items():
        if start <= date <= end:
            return 'Winter' if season in ['Winter_1', 'Winter_2']
                else season


# Apply the function to assign seasons
df_images['season'] = df_images['date'].apply(assign_season)

# Separate the data into different seasons
```

```
winter_data = df_images[df_images['season'] == 'Winter']
spring_data = df_images[df_images['season'] == 'Spring']
summer_data = df_images[df_images['season'] == 'Summer']
autumn_data = df_images[df_images['season'] == 'Autumn']

# Plotting histogram and saving the figure
plt.figure(figsize=(12, 6))
df_images['date'].hist(bins=50, edgecolor='black')
plt.xlabel('Date')
plt.ylabel('Number of Images')
plt.title('Histogram of Image Count by Date')
plt.xticks(rotation=45)
plt.tight_layout()
plt.savefig(os.path.join(script_dir, 'plots/data_dates.png'))
plt.show()

# Define top sites
"""top_sites = [
    'sites_corshamaqueduct_cam1', 'Cornwall_BudeCedarGrove',
    'Cornwall_Crinnis',
    'Devon_BarnstapleConeyGut_Scree',
    'Cornwall_Mevagissey_PreScree',
    'sites_sheptonmallet_cam2', 'Cornwall_PenzanceCC'
]"""


# Function to filter and balance data within each group
def filter_and_balance(data):
    sampled_dfs = []
    for site, group in data.groupby('site'):
        blocked_df = group[group['label'] == 'blocked']
        clear_df = group[group['label'] == 'clear']
        if not blocked_df.empty and not clear_df.empty:
            sample_size = min(len(blocked_df), len(clear_df))
            blocked_sample = blocked_df.sample(n=sample_size,
                random_state=1)
            clear_sample = clear_df.sample(n=sample_size,
                random_state=1)
            sampled_df = pd.concat([blocked_sample, clear_sample])
            sampled_dfs.append(sampled_df)

    return pd.concat(sampled_dfs).sample(frac=1,
        random_state=1).reset_index(
        drop=True) if sampled_dfs else pd.DataFrame()


# Apply the function to each season's data
balanced_winter = filter_and_balance(winter_data)
balanced_spring = filter_and_balance(spring_data)
balanced_summer = filter_and_balance(summer_data)
balanced_autumn = filter_and_balance(autumn_data)

# Print results
print("Balanced Winter Data:", balanced_winter, sep="\n")
print("\nBalanced Spring Data:", balanced_spring, sep="\n")
print("\nBalanced Summer Data:", balanced_summer, sep="\n")
print("\nBalanced Autumn Data:", balanced_autumn, sep="\n")


# Function to split dataset into training/validation and test sets
def initial_split(season_data, test_size=0.2, random_state=42):
    image_filenames = season_data['file_path'].tolist()
    labels = season_data['label'].apply(lambda x: 1 if x ==
        'blocked' else 0).tolist()
    train_val_filenames, test_filenames, train_val_labels,
        test_labels = train_test_split(
        image_filenames, labels, test_size=test_size,
        random_state=random_state)
    return train_val_filenames, test_filenames, train_val_labels,
        test_labels


# Function to further split the training/validation set into
    training and validation sets
def train_val_split(train_val_filenames, train_val_labels,
    val_size=0.2, random_state=42):
    train_filenames, val_filenames, train_labels, val_labels =
        train_test_split(
        train_val_filenames, train_val_labels, test_size=val_size,
        random_state=random_state)
    return train_filenames, val_filenames, train_labels, val_labels


# Splitting data for each season
autumn_train_val, autumn_test, autumn_train_val_labels,
    autumn_test_labels = initial_split(balanced_autumn)
winter_train_val, winter_test, winter_train_val_labels,
    winter_test_labels = initial_split(balanced_winter)
spring_train_val, spring_test, spring_train_val_labels,
    spring_test_labels = initial_split(balanced_spring)

# Further split the training/validation set into training and
    validation sets
autumn_train, autumn_val, autumn_train_labels, autumn_val_labels =
```

```
    train_val_split(autumn_train_val,
                    autumn_train_val_labels)
winter_train, winter_val, winter_train_labels, winter_val_labels =
    train_val_split(winter_train_val,
                    winter_train_val_labels)
spring_train, spring_val, spring_train_labels, spring_val_labels =
    train_val_split(spring_train_val,
                    spring_train_val_labels)
```

D.3 File: train_seasonal.py

```
import torch
from PIL import Image
from torchvision import transforms, models
from torchvision.models import ResNet50_Weights
import torch.nn as nn
import torch.optim as optim
from seasonal_data_split import winter_train, winter_val,
    winter_train_labels, winter_val_labels, spring_train, \
    spring_val, spring_train_labels, spring_val_labels,
        autumn_train, autumn_val, autumn_train_labels,
        autumn_val_labels
import matplotlib.pyplot as plt

device = torch.device("cuda" if torch.cuda.is_available() else
                          ("mps" if
                              torch.backends.mps.is_available()
                              else "cpu"))
print(f"Using device: {device}")

# Define the custom dataset class
class ScreenDataset(torch.utils.data.Dataset):
    def __init__(self, filenames, labels, xmin=-1, xmax=-1,
        ymin=-1, ymax=-1):
        self.preprocess = transforms.Compose([
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406],
                std=[0.229, 0.224, 0.225]),
        ])
        self.xmin = xmin
        self.xmax = xmax
        self.ymin = ymin
        self.ymax = ymax
        self.filenames = filenames
        self.labels = labels

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

    def __getitem__(self, item):
        img = Image.open(self.filenames[item]).convert('RGB')
        if self.xmin > 0 and self.xmax > 0 and self.ymin > 0 and
            self.ymax > 0:
            img = img.crop((self.xmin, self.ymin, self.xmax,
                self.ymax))
        label = self.labels[item]
        return self.preprocess(img), label


# Function to train the model (from your provided code)
def train_model(model, dataloaders, criterion, optimizer,
    scheduler=None, num_epochs=25, min_delta=0.01, patience=5):
    model.to(device)
    best_model_wts = None
    best_acc = 0.0
    epochs_no_improve = 0

    train_losses = []
    train_accuracies = []
    val_losses = []
    val_accuracies = []

    best_train_acc = 0.0
    best_val_acc = 0.0

    for epoch in range(num_epochs):
        print(f'Epoch {epoch}/{num_epochs - 1}')
        print('-' * 10)

        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()
            else:
                model.eval()

            running_loss = 0.0
            running_corrects = 0

            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)

                optimizer.zero_grad()

                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    if phase == 'train':
                        loss.backward()
                        optimizer.step()
```

```
            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)

        epoch_loss = running_loss /
            len(dataloaders[phase].dataset)
        epoch_acc = running_corrects.float() /
            len(dataloaders[phase].dataset)

        print(f'{phase} Loss: {epoch_loss:.4f} Acc:
            {epoch_acc:.4f}')

        if phase == 'train':
            train_losses.append(epoch_loss)
            train_accuracies.append(epoch_acc.cpu().numpy())
                # Convert to CPU and then to numpy
            best_train_acc = max(best_train_acc, epoch_acc)
            if scheduler:
                scheduler.step()
        else:
            val_losses.append(epoch_loss)
            val_accuracies.append(epoch_acc.cpu().numpy())  #
                Convert to CPU and then to numpy
            best_val_acc = max(best_val_acc, epoch_acc)
            # Early stopping
            if epoch_acc - best_acc > min_delta:
                best_acc = epoch_acc
                best_model_wts = model.state_dict()
                epochs_no_improve = 0
            else:
                epochs_no_improve += 1

            if epochs_no_improve == patience:
                print("Early stopping")
                if best_model_wts is not None:
                    model.load_state_dict(best_model_wts)
                return model, (train_losses, val_losses,
                    train_accuracies, val_accuracies),
                    best_train_acc, best_val_acc

    print(f'Best val Acc: {best_acc:.4f}')
    if best_model_wts is not None:
        model.load_state_dict(best_model_wts)
    return model, (train_losses, val_losses, train_accuracies,
        val_accuracies), best_train_acc, best_val_acc

# Choose the season to train on
season = 'winter'

# Use a dictionary to map the season name to the corresponding
    data variable
season_data_dict = {
    'winter': (winter_train, winter_val, winter_train_labels,
        winter_val_labels),
    'spring': (spring_train, spring_val, spring_train_labels,
        spring_val_labels),
    'autumn': (autumn_train, autumn_val, autumn_train_labels,
        autumn_val_labels),
    'all': (
        autumn_train + winter_train + spring_train,
        autumn_val + winter_val + spring_val,
        autumn_train_labels + winter_train_labels +
            spring_train_labels,
        autumn_val_labels + winter_val_labels + spring_val_labels
    )
}

if season == 'all':
    train_filenames, val_filenames, train_labels, val_labels =
        season_data_dict['all']
else:
    # Handle individual seasons
    train_filenames, val_filenames, train_labels, val_labels =
        season_data_dict[season]

# Coordinates for cropping (change if needed)
xmin, xmax, ymin, ymax = -1, -1, -1, -1

# Create datasets
train_dataset = ScreenDataset(train_filenames, train_labels, xmin,
    xmax, ymin, ymax)
val_dataset = ScreenDataset(val_filenames, val_labels, xmin, xmax,
    ymin, ymax)

# Create dataloaders
batch_size = 64
dataloaders = {
    'train': torch.utils.data.DataLoader(train_dataset,
        batch_size=batch_size, shuffle=True, num_workers=0),
    'val': torch.utils.data.DataLoader(val_dataset,
        batch_size=batch_size, shuffle=True, num_workers=0)
}

# Load ResNet50 model
```

```
print("Loading␣ResNet50␣model...")
model = models.resnet50(weights=ResNet50_Weights.IMAGENET1K_V1)

# Modify the final layer for binary classification
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, 2)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Train the model
model, history, best_train_acc, best_val_acc = train_model(model,
    dataloaders, criterion, optimizer, num_epochs=25,
    min_delta=0.01, patience=5)

# Plot training graph
train_losses, val_losses, train_accuracies, val_accuracies =
    history

plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(train_losses, label='Training␣Loss')
plt.plot(val_losses, label='Validation␣Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training␣and␣Validation␣Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(train_accuracies, label='Training␣Accuracy')
plt.plot(val_accuracies, label='Validation␣Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Training␣and␣Validation␣Accuracy')
plt.legend()

# Save the plot
plot_filepath = f'plots/{season}_training_validation_plot.png'
plt.savefig(plot_filepath)
print(f"Plot␣saved␣to␣{plot_filepath}")

# Save best scores to a file
scores_filepath = f'{season}_best_scores.txt'
with open(scores_filepath, 'w') as f:
    f.write(f'Best␣Training␣Accuracy:␣{best_train_acc:.4f}\n')
    f.write(f'Best␣Validation␣Accuracy:␣{best_val_acc:.4f}\n')
print(f"Best␣scores␣saved␣to␣{scores_filepath}")

# Confirmation prompt before saving the model
save_model = input("Do␣you␣want␣to␣save␣the␣model?␣(yes/no):␣
    ").strip().lower()
if save_model == 'yes':
    # Save the model
    model_filepath = f'weights/{season}_classifier.pth'
    torch.save(model.state_dict(), model_filepath)
    print(f"Model␣saved␣to␣{model_filepath}")
else:
    print("Model␣not␣saved.")
```

D.4 File: classification_network.py

```
import torch
from PIL import Image
from torchvision import transforms
from torchvision.models import resnet50
import torch.nn as nn
import pandas as pd
from sklearn.model_selection import train_test_split

from seasonal_data_split import winter_test, winter_test_labels,
    spring_test, spring_test_labels, autumn_test,
    autumn_test_labels


class ScreenDataset(torch.utils.data.Dataset):
    def __init__(self, filenames, xmin=-1, xmax=-1, ymin=-1,
        ymax=-1):
        self.preprocess = transforms.Compose([
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406],
                std=[0.229, 0.224, 0.225]),
        ])
        self.xmin = xmin
        self.xmax = xmax
        self.ymin = ymin
        self.ymax = ymax
        self.filenames = filenames

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

    def __getitem__(self, item):
        img = Image.open(self.filenames[item])
        if self.xmin > 0 and self.xmax > 0 and self.ymin > 0 and
            self.ymax > 0:
            img = img.crop((self.xmin, self.ymin, self.xmax,
                self.ymax))
        return self.filenames[item], self.preprocess(img)


def split_season_data(season_data, test_size=0.2, random_state=42):
    image_filenames = season_data['file_path'].tolist()
    labels = season_data['label'].apply(lambda x: 1 if x ==
        'blocked' else 0).tolist()
    _, val_filenames, _, val_labels =
        train_test_split(image_filenames, labels,
        test_size=test_size,
                                                    random_state=random_state)
    return val_filenames, val_labels


def predict_for_dataset(season, model, device, threshold=0.5):
    # Use a dictionary to map the season name to the corresponding
        data variable
    season_data_dict = {
        'winter': winter_test,
        'spring': spring_test,
        'autumn': autumn_test,
        'all': winter_test + spring_test + autumn_test
    }

    season_label_dict = {
        'winter': winter_test_labels,
        'spring': spring_test_labels,
        'autumn': autumn_test_labels,
        'all': winter_test_labels + spring_test_labels +
            autumn_test_labels
    }

    val_filenames = season_data_dict[season]
    val_labels =  season_label_dict[season]

    xmin, xmax, ymin, ymax = -1, -1, -1, -1  # Adjust these as
        needed

    screen_dataset = ScreenDataset(val_filenames, xmin, xmax,
        ymin, ymax)
    dataloader = torch.utils.data.DataLoader(screen_dataset,
        batch_size=64, shuffle=False)

    softmax = nn.Softmax(dim=1)

    # Create a DataFrame to store predictions for the validation
        set
    val_df = pd.DataFrame({'file_path': val_filenames, 'label':
        val_labels})
    val_df['pred'] = None

    for filenames, images in dataloader:
        images = images.to(device)
        predictions = softmax(model(images)).detach()
        for i in range(len(filenames)):
```

```
            prediction = "blocked" if predictions[i, 1].item() >
                threshold else "clear"
            val_df.loc[val_df['file_path'] == filenames[i],
                'pred'] = prediction

    # Save the validation set DataFrame with predictions
    val_df.to_csv(f'csvs/{season}_pred_{model_season}_model.csv',
        index=False)
    print(f"Predictions saved for {season} test set.")


if __name__ == "__main__":
    for model_season in ["winter", "spring", "autumn", "all"]: #
        Iterate over model seasons
        model_filepath = f'weights/{model_season}_classifier.pth'

        device = torch.device("cuda" if torch.cuda.is_available()
            else
                              ("mps" if
                                  torch.backends.mps.is_available()
                                  else "cpu"))
print(f"Using device: {device}")

# Load and prepare the model
model = resnet50()
num_ftrs = model.fc.in_features
model.fc = nn.Linear(num_ftrs, 2)
model.to(device)
model.load_state_dict(torch.load(model_filepath,
    map_location=device))
model.eval()

# Predict for each dataset (winter, spring, autumn, all)
for season in ["winter", "spring", "autumn", "all"]:
    predict_for_dataset(season, model, device)
```

D.5 File: seasonal_plot.py

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import glob

# File paths
file_paths = glob.glob('csvs/*_pred_*_model.csv')

# Create an empty list to hold dataframes
dataframes = []

# Load all CSV files and add a column to identify the model and
    season
for file_path in file_paths:
    df = pd.read_csv(file_path)
    season = file_path.split('_pred_')[0].split('/')[-1]
    model = file_path.split('_pred_')[1].split('_model')[0]
    df['model'] = model
    df['season'] = season

    # Convert predictions to numeric form to match labels
    df['pred'] = df['pred'].map({'clear': 0, 'blocked': 1})

    dataframes.append(df)

    print(f'Model: {model}, Season: {season}')
    print(df.head())  # Inspect the first few rows of each
        DataFrame

# Combine all data into a single DataFrame
df = pd.concat(dataframes, ignore_index=True)
print(df.head())  # Inspect the combined DataFrame

# Calculate the accuracy for each model and season
accuracy_df = df.groupby(['model', 'season']).apply(lambda x:
    (x['label'] == x['pred']).mean()).reset_index(
    name='accuracy')
print(accuracy_df)  # Inspect the accuracy DataFrame

# Set up the plot with a larger font size and colour-blind
    friendly palette
plt.figure(figsize=(14, 8))
sns.set_palette("colorblind")  # Ensure this line is in place
sns.barplot(data=accuracy_df, x='season', y='accuracy',
    hue='model', width=0.6)  # Adjusted width for spacing

# Add the title and labels with larger font sizes
plt.title('Prediction Accuracy by Season and Model', fontsize=20,
    y=1.05)  # Moved the title further up
plt.xlabel('Season', fontsize=16)
plt.ylabel('Accuracy', fontsize=16)
plt.ylim(0, 1)  # Accuracy ranges from 0 to 1

# Add gridlines for better readability
plt.grid(True, linestyle='--', alpha=0.7)

# Improve legend placement (inside bottom-right)
plt.legend(title='Model', fontsize=12, title_fontsize=14,
    loc='lower right')

# Annotate the bars with accuracy values, ensuring consistent
    decimal places
for p in plt.gca().patches:
    height = p.get_height()
    # Only annotate if height is greater than a very small
        threshold
    if height > 0.001:
        plt.gca().annotate(f'{height:.2f}', (p.get_x() +
            p.get_width() / 2., height),
                           ha='center', va='center', fontsize=12,
                               color='black', xytext=(0, 8),
                           textcoords='offset points')

# Save the plot to a file
plt.savefig('plots/accuracy_comparison_all_models.png',
    bbox_inches='tight')

# Show the plot
plt.show()
```

D.6 File: saliency_mapping.py

```
import torch
from torchvision import models, transforms
from torchvision.models import ResNet50_Weights
import torch.nn as nn
import matplotlib.pyplot as plt
from PIL import Image
import numpy as np

def load_model(model_path, device):
    model = models.resnet50(weights=ResNet50_Weights.IMAGENET1K_V1)
    num_ftrs = model.fc.in_features
    model.fc = nn.Linear(num_ftrs, 2)
    model.load_state_dict(torch.load(model_path,
        map_location=device))
    model.to(device)
    model.eval()
    return model

def preprocess_image(image_path):
    preprocess = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225]),
    ])
    try:
        img = Image.open(image_path).convert('RGB')
        return preprocess(img).unsqueeze(0)
    except Exception as e:
        print(f"Error␣loading␣image␣{image_path}:␣{e}")
        return None

def generate_saliency_map(model, img_tensor, class_idx):
    img_tensor.requires_grad = True
    output = model(img_tensor)
    model.zero_grad()
    output[0, class_idx].backward()
    saliency, _ = torch.max(img_tensor.grad.data.abs(), dim=1)
    saliency = (saliency - saliency.min()) / (saliency.max() -
        saliency.min())
    return saliency.squeeze().cpu().numpy()

def process_image(model, image_path, device):
    img_tensor = preprocess_image(image_path)
    if img_tensor is None:
        return None, None
    img_tensor = img_tensor.to(device)
    img_pil = Image.open(image_path).convert('RGB').resize((224,
        224))

    output = model(img_tensor)
    class_idx = torch.argmax(output).item()

    saliency_map = generate_saliency_map(model, img_tensor,
        class_idx)
    return saliency_map, img_pil

if __name__ == "__main__":
    device = torch.device('mps' if
        torch.backends.mps.is_available() else 'cpu')
    classifier = 'spring'  # Use a single classifier
    model_path = f'weights/{classifier}_classifier.pth'
    image_paths = [
        'Data/blockagedetection_dataset/images/sites_sheptonmallet_cam2/blocked/20
        'Data/blockagedetection_dataset/images/sites_sheptonmallet_cam2/blocked/20
        'Data/blockagedetection_dataset/images/sites_sheptonmallet_cam2/blocked/20
    ]

    # Load the model once
    model = load_model(model_path, device)

    for idx, image_path in enumerate(image_paths):
        saliency_map, img_pil = process_image(model, image_path,
            device)
        if saliency_map is not None:
            saliency_map_resized =
                Image.fromarray(np.uint8(saliency_map *
                255)).resize((224, 224), resample=Image.BILINEAR)

            plt.figure(figsize=(5, 5))
            plt.imshow(img_pil, alpha=0.6)
            plt.imshow(saliency_map_resized, cmap='hot', alpha=0.4)
            #plt.title(f'Saliecy␣Mapping␣using␣{classifier}␣
                classifier')
            plt.axis('off')
            plt.tight_layout()
            plt.savefig(f'plots/saliency_{classifier}_{idx␣+␣
                1}.png', bbox_inches='tight')
            plt.show()
```

D.7 File: integrated_gradients.py

```
import torch
from torchvision import models, transforms
from torchvision.models import ResNet50_Weights
import torch.nn as nn
import matplotlib.pyplot as plt
from PIL import Image
import numpy as np
from captum.attr import IntegratedGradients

def load_model(model_path):
    model = models.resnet50(weights=ResNet50_Weights.IMAGENET1K_V1)
    num_ftrs = model.fc.in_features
    model.fc = nn.Linear(num_ftrs, 2)
    model.load_state_dict(torch.load(model_path,
        map_location=torch.device('mps')))
    model.eval()
    return model

def preprocess_image(image_path):
    preprocess = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225]),
    ])
    try:
        img = Image.open(image_path).convert('RGB')
        return preprocess(img).unsqueeze(0)
    except Exception as e:
        print(f"Error loading image {image_path}: {e}")
        return None

def integrated_gradients(model, img_tensor, class_idx):
    ig = IntegratedGradients(model)
    attributions = ig.attribute(img_tensor, target=class_idx,
        baselines=img_tensor * 0)
    return attributions

def visualize_attributions(attributions, img_pil, ax, method_name):
    img_pil_resized = img_pil.resize((224, 224))

    attributions = attributions.squeeze().cpu().detach().numpy()
    attributions = np.transpose(attributions, (1, 2, 0))

    attributions = np.sum(attributions, axis=2)

    attributions = np.clip(attributions, 0,
        np.percentile(attributions, 99))

    attributions = (attributions - attributions.min()) /
        (attributions.max() - attributions.min())

    ax.imshow(img_pil_resized)

    height, width = img_pil_resized.size
    ax.imshow(attributions, cmap='hot', alpha=0.6, extent=(0,
        width, height, 0))

    #ax.set_title(f'{method_name}')
    ax.axis('off')

def process_image(model_path, image_path):
    model = load_model(model_path)
    img_tensor = preprocess_image(image_path)
    if img_tensor is None:
        return None, None
    img_pil = Image.open(image_path).convert('RGB').resize((224,
        224))

    output = model(img_tensor)
    class_idx = torch.argmax(output).item()

    ig_attributions = integrated_gradients(model, img_tensor,
        class_idx)
    return ig_attributions, img_pil


if __name__ == "__main__":
    device = torch.device('mps' if
        torch.backends.mps.is_available() else 'cpu')

    # Define the single classifier to be used
    classifier = 'winter'
    model_path = f'weights/{classifier}_classifier.pth'

    # List of different image paths
    image_paths = [
        'Data/blockagedetection_dataset/images/sites_sheptonmallet_cam2/blocked/20
        'Data/blockagedetection_dataset/images/sites_sheptonmallet_cam2/blocked/20
        'Data/blockagedetection_dataset/images/sites_sheptonmallet_cam2/blocked/20
    ]
```

```
# Process each image and save the result individually
for idx, image_path in enumerate(image_paths):
    attributions, img_pil = process_image(model_path,
        image_path)
    if attributions is not None:
        plt.figure(figsize=(5, 5))
        ax = plt.gca()  # Get the current axis
        visualize_attributions(attributions, img_pil, ax,
            f"Image {idx + 1} - {classifier.capitalize()}")
        plt.axis('off')
        plt.tight_layout()
        plt.savefig(f'plots/integrated_{classifier}_{idx +
            1}.png', bbox_inches='tight')
        plt.show()
```

D.8 File: occlusion.py

```
import torch
from torchvision import models, transforms
from torchvision.models import ResNet50_Weights
import torch.nn as nn
import matplotlib.pyplot as plt
from PIL import Image
import numpy as np
from captum.attr import Occlusion

def load_model(model_path, device):
    model = models.resnet50(weights=ResNet50_Weights.IMAGENET1K_V1)
    num_ftrs = model.fc.in_features
    model.fc = nn.Linear(num_ftrs, 2)
    model.load_state_dict(torch.load(model_path,
        map_location=device))
    model.to(device)
    model.eval()
    return model

def preprocess_image(image_path):
    preprocess = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225]),
    ])
    try:
        img = Image.open(image_path).convert('RGB')
        return preprocess(img).unsqueeze(0)
    except Exception as e:
        print(f"Error␣loading␣image␣{image_path}:␣{e}")
        return None

def compute_occlusion(model, img_tensor, class_idx):
    occlusion = Occlusion(model)
    attributions = occlusion.attribute(img_tensor,
        target=class_idx, strides=(3, 8, 8),
        sliding_window_shapes=(3, 15, 15))
    return attributions

def process_image(model_path, image_path, device):
    model = load_model(model_path, device)
    img_tensor = preprocess_image(image_path)
    if img_tensor is None:
        return None, None
    img_tensor = img_tensor.to(device)
    img_pil = Image.open(image_path).convert('RGB').resize((224,
        224))

    output = model(img_tensor)
    class_idx = torch.argmax(output).item()

    occlusion_attributions = compute_occlusion(model, img_tensor,
        class_idx)
    return occlusion_attributions, img_pil

def visualize_image(image_path, model_path, model_name, device,
    save_path):
    attributions, img_pil = process_image(model_path, image_path,
        device)
    if attributions is not None and img_pil is not None:
        attributions =
            attributions.squeeze().cpu().detach().numpy()
        attributions = np.transpose(attributions, (1, 2, 0))
        attributions = np.sum(attributions, axis=2)
        attributions = np.clip(attributions, 0,
            np.percentile(attributions, 99))
        attributions = (attributions - attributions.min()) /
            (attributions.max() - attributions.min())

        img_pil_resized = img_pil.resize((224, 224))
        width, height = img_pil_resized.size

        plt.figure(figsize=(5, 5))
        plt.imshow(img_pil_resized)
        plt.imshow(attributions, cmap='hot', alpha=0.6, extent=(0,
            width, height, 0))
        #plt.title(f'{model_name}')
        plt.axis('off')
        plt.tight_layout()
        plt.savefig(save_path, bbox_inches='tight')
        plt.show()

if __name__ == "__main__":
    device = torch.device('mps' if
        torch.backends.mps.is_available() else 'cpu')
    classifier = 'spring'
    model_path = f'weights/{classifier}_classifier.pth'

    image_paths = [
        'Data/blockagedetection_dataset/images/sites_sheptonmallet_cam2/blocked/20
        'Data/blockagedetection_dataset/images/sites_sheptonmallet_cam2/blocked/20
```

```
        'Data/blockagedetection_dataset/images/sites_sheptonmallet_cam2/blocked/2022_04_08_08_30.jpg'
    ]

    """image_paths = [
        'Data/blockagedetection_dataset/images/Cornwall_Portreath/blocked/2022_11_26_12_59_59.jpg',
        'Data/blockagedetection_dataset/images/sites_sheptonmallet_cam2/blocked/2022_04_08_08_30.jpg',
        'Data/blockagedetection_dataset/images/sites_corshamaqueduct_cam1/blocked/2022_04_08_08_30.jpg'
    ]"""

    for idx, image_path in enumerate(image_paths):
        visualize_image(image_path, model_path, classifier,
                        device, f'plots/occlusion_{classifier}_{idx + 1}.png')
```

D.9 File: grad-cam.py

```
import torch
from torchvision import models, transforms
from torchcam.methods import SmoothGradCAMpp
from torchcam.utils import overlay_mask
from torchvision.models import ResNet50_Weights
import torch.nn as nn
import matplotlib.pyplot as plt
from PIL import Image

def load_model(model_path, device):
    model = models.resnet50(weights=ResNet50_Weights.IMAGENET1K_V1)
    num_ftrs = model.fc.in_features
    model.fc = nn.Linear(num_ftrs, 2)
    model.load_state_dict(torch.load(model_path,
        map_location=device))
    model.to(device)
    model.eval()
    return model

def preprocess_image(image_path):
    preprocess = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225]),
    ])
    try:
        img = Image.open(image_path).convert('RGB')
        return preprocess(img).unsqueeze(0)
    except Exception as e:
        print(f"Error␣loading␣image␣{image_path}:␣{e}")
        return None

def visualize_cam(model, cam_extractor, img_tensor, img_pil,
    class_idx):
    activation_maps = cam_extractor(class_idx, model(img_tensor))

    for activation_map in activation_maps:
        activation_map = activation_map.squeeze(0)
        activation_map_pil =
            transforms.functional.to_pil_image(activation_map,
            mode='F')
        result = overlay_mask(img_pil, activation_map_pil,
            alpha=0.6)

        return result

def process_image(model_path, image_path, device):
    model = load_model(model_path, device)

    cam_extractor = SmoothGradCAMpp(model)

    img_tensor = preprocess_image(image_path)
    if img_tensor is None:
        return None
    img_tensor = img_tensor.to(device)
    img_pil = Image.open(image_path).convert('RGB').resize((224,
        224))

    output = model(img_tensor)
    class_idx = torch.argmax(output).item()

    return visualize_cam(model, cam_extractor, img_tensor,
        img_pil, class_idx)

if __name__ == "__main__":
    device = torch.device('mps' if
        torch.backends.mps.is_available() else 'cpu')

    # Define the seasons and corresponding model paths
    classifier = 'spring'  # Single classifier to be used
    model_path = f'weights/{classifier}_classifier.pth'

    # List of different image paths
    image_paths = [
        'Data/blockagedetection_dataset/images/sites_sheptonmallet_cam2/blocked/20
        'Data/blockagedetection_dataset/images/sites_sheptonmallet_cam2/blocked/20
        'Data/blockagedetection_dataset/images/sites_sheptonmallet_cam2/blocked/20
    ]

    # Process each image and save the result individually
    for idx, image_path in enumerate(image_paths):
        result = process_image(model_path, image_path, device)
        if result is not None:
            plt.figure(figsize=(5, 5))
            plt.imshow(result)
            plt.axis('off')
            plt.tight_layout()
            plt.savefig(f'plots/gradcam_{classifier}_{idx␣+␣
                1}.png', bbox_inches='tight')
            plt.show()
```