Glass Detection in Videos

Student: Harvey Mannering
Supervisor: Gabriel Brostow

MSc Computer Graphics, Vision, and Imaging

September 2022

This report is submitted as part requirement for the MSc Degree in Computer Graphics, Vision and Imaging at University College London. It is substantially the result of my own work except where explicitly indicated in the text.

Department of Computer Science

University College London

Abstract

Many computer vision applications, in particular navigation, struggle when presented with glass. Recent advances in glass detection give promising results, but thus far have only been applied to images. To our knowledge, this project is the first attempt to extend the glass detection problem from images to videos. To this end, we present a new network called GSDNet4Video and find that temporal stability of glass detection can be improved using propagation-based techniques.

Our method involves adding a fourth input channel to an existing glass detection neural network. This extra channel takes in the mask from a previous frame. Through careful training using static images, a dependency between the previous frame's result and the current frame's result can be established. We demonstrate that this increases temporal stability through qualitative analysis. Methods of fine tuning GSDNet4Video are also explored. The effects of heavy data augmentation and active learning are both considered. We develop a novel query strategy based on flicker in between frames, however its efficacy remains uncertain. In the final section we suggest the next steps to be undertaken for anyone wishing to further explore a query strategy based on flickering pixels.

Acknowledgements

Thank you to Gabriel Brostow for pointing me towards many of the ideas fundamental to the project. Thank you to Jamie Watson for his helpful and informative suggestions with regard to training the baseline GSDNet4Video model. And finally, thank you to Mohamed Sayed, Gizem Unlu, and Omiros Pantazis for their help, guidance, and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The presence of glass in real world scenes has proved problematic for many computer vision applications. For instance, large glass elements present challenges and potential dangers to robot navigation systems [34]. Novel view synthesis also struggles with glass. NeRF can manage simple reflections, but not complex ones, like those produced by glass [12]. Windows, due to the material properties of glass, are particularly problematic for 3D reconstruction, causing errors and large scale artifacts [18]. But despite these problem areas and more, glass detection in RGB images has received little attention until recently. Since the publication of the first large-scale glass detection dataset [17] in 2020, glass detection in images has been an active area of research.

Glass is difficult mainly due to its transparency. A picture with and without a sheet of glass in front of it may look virtually identical. To solve this problem we can take inspiration from how humans detect glass. Which images in Figure 1.1 contain glass? And how do we know this? Figure 1.1a does contain glass. We can see this not from the window area itself, but from the frame. Our experience of the real world tells us that window frames usually contain glass. The overall context of an image is therefore immensely important for detecting glass. At a glance Figure 1.1b may look as though it contains glass too. Chrome railings like these are usually filled with glass panes, however, if we examine more closely, we see there is no glass. Non of the visual phenomena (e.g. reflection, condensation, discolouration) exhibited by glass are present in the area where we might expect glass to be. This demonstrates that just relying on the image context is not sufficient for the glass detection problem. This is especially true for Figure 1.1c which is shot through a pane of glass. As a result the perimeter of the glass is not visible at all and the only clue we have is a faint reflection on the glass surface.

Despite it's difficulty, glass detection methods are improving year on year [17, 15, 40], but the problem of glass detection in videos remains unexplored in the literature. Both opportunities and challenges are presented when extending glass detection from image to video. The main opportunity is that videos contain much more information than images. This can hopefully be leveraged to make the glass detection more accurate. One of the key challenges, especially for the purposes of navigation, is speed. For example, inference time on the state of the art glass detection neural network GSDNet [15] is 5.81 seconds when performed on a MacBook Pro (2020) with M1 processor. Yet another challenge is a lack of data. While plenty of datasets now exist for glass detection in images [17, 15, 40, 37], no datasets

are available for glass detection in video. To our knowledge this project is the first attempt to extend the glass detection problem from images to videos.

Figure 1.1: Images for which glass detection is difficult. Image (a) is taken from GSD.

1.1 Overview

This project aims to bring glass detection, a problem that has until now only been attempted in images, into the video domain. As a starting point we take existing models and naively perform glass detection on each frame of a video. Accuracy for these videos is good, but flickering pixels is a problem. To be specific, temporal stability of detected glass regions needs improvement. Objects classified as *glass* can and often do change their classification to *not glass* (or vice versa) throughout videos. Temporal stability will therefore be our focus as we attempt to move glass detection into the video domain.

This project has two key parts corresponding to two key hypotheses. Both revolve around improving the temporal stability of glass detection when performing inference on videos. Taking as our starting point a convolution neural network (CNN) that naively performs glass detection frame by frame, these two hypotheses are:

(1) Temporal stability of glass detection in RGB videos can be improved by adding a fourth channel which takes in the previous frame's mask.

(2) Active learning, with a new query strategy based on flicker between frames, can be used to further improve temporal stability while simultaneously reducing the amount of training data needed.

1.2 GSDNet4Video

The glass detection problem in RGB images can be stated as follows: given an image, classify all pixels as either *glass* or *not glass*. Since this is a binary choice, pixels that are classified as *glass* are labelled with a '1' and pixels that are classified as *not glass* are labelled with a '0'. To train a glass detection model, each image will need a corresponding 2D array of matching size, containing a label for each pixel. This 2D array is called a mask. See Figure 2.4 for examples of images and their corresponding masks. There are variations on this problem, such as transparent object detection [37] or glass detection using depth sensors [35], but they will not be the focus of this project. We only consider glass using only RGB images.

The field of glass detection, as with segmentation [19], is currently dominated by deep learning approaches. After evaluating state of the art models, we decided the best candidate for extending to video would be GSDNet [15]. GSDNet is a CNN which uses ResNeXt-101 as its backbone. Using a technique similar to Atrous Spatial Pyramid Pooling a prediction is made for each pixel. Finally, this prediction is refined using the Reflection-based Refinement Module, a U-Net [24] like network that improves the prediction by looking for reflections characteristic of glass.

Using the propagation-based techniques described in [2], we convert GSDNet into a video model. We call this new model GSDNet4Video. While GSDNet has three input channels (RGB), GSDNet4Video has four. The additional channel takes the previous frame's mask. In most videos, consecutive frames will be similar, and as a result, consecutive masks should also be similar. Therefore, by using the previous frame's mask as an input, we are providing our network with a strong clue as to how the current frame's mask should look. The issue is that we cannot train the network on a previous frame's mask because we do not have that data. There are no video datasets for glass detection. However, by performing minor transformations on the masks that we do have, we can simulate shifts in perspective. These minor transformations match what we would expect to see in a real video. In this way, we can train a video model using a single image dataset.

We show through qualitative analysis that GSDNet4Video has better temporal stability than GSDNet when used naively. Our results suggest that GSDNet outperforms GSDNet4Video in accuracy very slightly. We have therefore shown our first hypothesis to be true. Temporal stability can be improved with the addition of a fourth channel, but this method has resulted in a small decrease the overall accuracy.

A fully labelled glass detection video dataset would open up a myriad of opportunities in this area, but further advances can still be made with the available resources. Our approach only considers the current frame and the previous frame, however, video segmentation techniques exist which can make use of more frames, thereby improving accuracy [21]. By replacing the original ResNeXt-101 backbone with ResNet-18, we have roughly halved inference time, nevertheless, if these models are to be deployed for robot navigation, inference time will need to be further reduced. This can perhaps be achieved using the methods found in [26] or in [32].

1.3 Fine Tuning via Active Learning

The absence of a glass dataset for video is an issue. The techniques described in the previous section approximate video data, but do not reflect the exact characteristics of real videos. For example, occlusion of glass is not accounted for by the methods described in [2]. GSDNet4Video therefore stands to gain a lot by fine tuning on real video data.

Collecting videos of glass can be done quickly and easily, almost any indoor scene contains some glass, but labelling takes a long time [8, 16]. Furthermore, neural networks are very data hungry [23]. Techniques that will learn more with less data are therefore desirable. Here we look to the literature on active learning, a field that aims to reduce the amount of training data needed by only using the most informative data. The method of choosing the most informative data is called the query strategy [25].

In this portion of the project we use active learning to fine tune the baseline GSDNet4Video model already developed. To help the network better generalize, addition data augmentation is done during fine tuning. We draw heavily on the framework used in [27] which can increase segmentation performance by training on just a few pixel labels per image. We also explore new query strategies based on flicker between frames as a means of improving temporal stability. Flicker can be defined as an object changing classification from one frame to the next and can be seen as a type of temporal uncertainty. Optical flow can be used to track individual pixels between two frames. With this we have all the components we need to finds flickering pixels between two frames. In Section 3.2, we detail two query strategies based on flicker, we call these *flickering regions* and *flickering pixels*.

We go on to evaluate the effectiveness of fine tuning, paying particular attention to the impact of learning rate and augmentation. We also examine whether our *flickering regions* and *flickering pixels* query strategies are effective at reducing the amount of training data needed. While there is evidence to suggest that training on flicker can effectively reduce the labelling burden, our findings are not conclusive. We have therefore not verified our second hypothesis. In Section 5.2 we recommend next steps to be taken if this query strategy were to be investigated further.

Chapter 2

Background

In this chapter, we will review the literature surrounding this project. Segmentation is foundational to modern glass detection algorithms, therefore relevant research from this field will be examined first. Building upon this foundation, we will then look at current models and datasets used for glass detection. Finally, we will examine active learning with a focus on how it can be applied to segmentation.

2.1 Segmentation

We want to determine which pixels in an image should be classified as *glass* and which should be classified as *not glass*. We can thus formulate this as a binary segmentation problem and draw upon the literature in this area. Segmentation is a well studied problem with applications in scene understanding, autonomous vehicles and robotic perception [19].

Two main families of segmentation exist, semantic segmentation and instance segmentation. Semantic segmentation aims to classify each pixel in an image with a semantic label. Instance segmentation on the other hand makes distinctions between individual objects [19]. For example, lets say we wanted to separate pixels belonging to clouds from pixels belonging to sky in Figure 2.1. If we ran semantic segmentation, each pixel would receive one of two labels, *sky* or *cloud*. However, if we ran instance segmentation on the same image, each distinct cloud would get its own unique label, which would then be assigned to the associated pixels.

In the case of detecting glass in RGB images, we are not considering individual instances of glass, just whether a pixel is *glass* or *not glass*. This is therefore a semantic segmentation problem.

2.1.1 Segmentation in Images

Classical methods for segmentation include thresholding, region growing, split and merge algorithms, snakes, and Markov Random Fields [5]. Recently, deep learning approaches have achieved state of the art performance on popular benchmarks and now lead the field [19]. What follows is a brief review of popular deep learning segmentation models which are most relevant to this project.

Encoder-decoder networks are one popular approach to segmenting images. These models first compress an image into a latent-space representation using an encoder (typically involving repeated convolution and pooling layers). A decoder can then use this latent-space representation to predict a segmentation map [19].

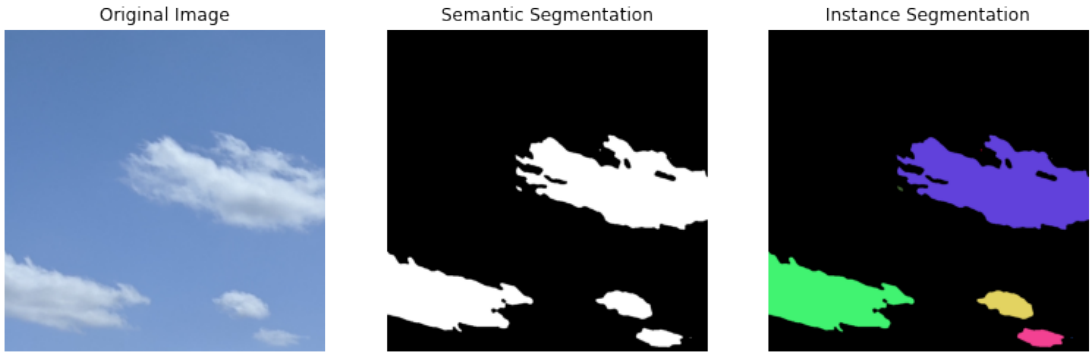

Figure 2.1: An example of semantic and instance segmentation performed on the same image. In the segmented images, distinct colours represent different labels.

U-Net [24] expands on this architecture. It's encoder and decoder (called the contracting and expanding paths in the original paper) are symmetric, meaning that the feature maps from the encoder can be concatenated with its counterpart in the decoder. This architecture, along with considerable data augmentation, achieved state of the art performance on three segmentation datasets.

DeepLab [6] improved segmentation using three methods. Firstly, atrous convolution (also known as dilated colvolution), allowed the receptive field to be enlarged while keeping the number of parameters the same. In regular convolution, kernels are applied to groups of neighbouring pixels like in Figure 2.2a, but with atrous convolution, elements of the kernel can be spread out. Figure 2.2b shows atrous convolution with a dilation rate of 2, meaning that every second pixel is used during convolution. Secondly, Atrous Spatial Pyramid Pooling (ASPP) was used to robustly segment objects at multiple scales. ASPP involves performing multiple atrous convolutions in parallel with different dilation rates and then fusing the results. Thirdly, the model combined fully connected conditional random fields with CNNs.

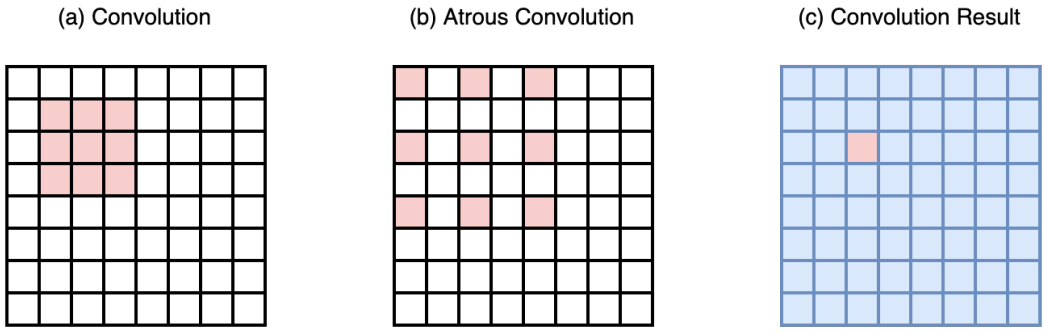

Figure 2.2: A visualization of atrous convolution. (a) shows a pixel grid with a 3x3 kernel overlaid. (b) shows a 3x3 kernel at the same pixel location, but with a dilation rate of 2, meaning this is an atrous convolution. (c) shows the output pixel grid, the red pixel is the output pixel for the convolutions shown in (a) and (b). To ensure the convolution result is the same size as the input image, zero padding can be used.

Attention can also be leveraged to account for long range dependencies between pixels. For ex-

ample, DANet [10] uses both channel and position wise attention (two separate branches of a neural network whose outputs are combined) to improve segmentation. MirrorNet [38], a model that segments mirrors in daily life scenes, uses both atrous convolution and attention in its architecture. To implement attention, it uses the lightweight CBAM module [36], which also works in both channel and position wise dimensions.

2.1.2 Metrics

Numerous metrics can be used to judge the accuracy of a segmentation model. One such metric is the Intersection of Union (IoU), also known as the Jaccard Index [19, 17]. It is calculated by taking the area of intersection between a predicted segmentation map P and the ground truth segmentation map G. This value is then divided by the total area covered by the union of the two maps.

$$\mathrm{IoU}(P, G) = \frac{|P \cap G|}{|P \cup G|} \tag{2.1}$$

Pixel Accuracy (PA) is also used to evaluate segmentation performance. This metric is the percentage of pixels that have been correctly classified. For binary segmentation it is given by

$$\mathrm{PA} = \frac{p_{00} + p_{11}}{p_{00} + p_{11} + p_{01} + p_{10}} \tag{2.2}$$

where p_{ij} is the total number of pixels that belong to class i and were predicted as class j [19].

For any binary classification problem an F-measure can be used, the most generic form of which is the F_β score [17]. This metric uses the number of true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN) to calculate the precision and recall. This is done like so

$$\text{Precision } = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}, \text{ Recall } = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \tag{2.3}$$

these values can then be used to calculate the F_β score as

$$F_\beta = \frac{(1 + \beta^2) Precision \times Recall}{(\beta^2 Precision) + Recall} \tag{2.4}$$

where β is chosen such that recall is β times more important than precision. All three metrics range from 0 to 1, with 1 being best and 0 being worst.

Multi-class segmentation models often use cross-entropy for their loss functions [24, 6, 27], however, Lovász-Softmax loss [4] can be used to directly maximise the mean intersection of union. Both loss functions have their own binary equivalents; binary cross-entropy loss and Lovász hinge loss.

2.1.3 Segmentation in Videos

As with image segmentation, the field of video segmentation has recently seen a huge influx of deep learning based techniques. Video segmentation algorithms fall into one of two categories; Video Object Segmentation (VOS) or Video Semantic Segmentation (VSS). VOS produces a binary foreground/background segmentation map and is not concerned with the exact categories of the segmented objects. Conversely, VSS cares about object categories when segmenting an image and can therefore produce a

multi-class segmentation map [33]. We have already identified glass detection in images as a semantic segmentation problem. Therefore, extending glass detecting into video is a VSS problem with just two classes, *glass* and *not glass*.

The simple solution to VSS would be the naive approach; using an image segmentation model to process each frame in isolation. However, this ignores helpful information, like coherence cues and temporal continuity, that resides in neighbouring frames. This extra information, inherent to videos, can greatly improve segmentation accuracy [33].

One approach is to use the segmentation mask from the previous frame as an extra input channel. That is to say, if a segmentation CNN takes a three channel image (RGB) as its input, then we can easily modify the network by adding an addition mask (M) channel to create an RGBM input (see Figure 2.3). The mask for the previous frame will likely be very similar to the current frame, and thus serves as a strong clue as to how the current frame's mask should look. Using only information from the previous frame is efficient, however, considering multiple previous (or future) frames will likely give better accuracy. One major advantage of this method is that it can be trained on single image/mask pairs without the need for video data. Through heavy data augmentation we can simulate previous frames in a video. For example, in order to simulate a perspective shift, we could simply apply an affine transformation to the ground truth mask. This transformed mask is then used in our M channel. [2].

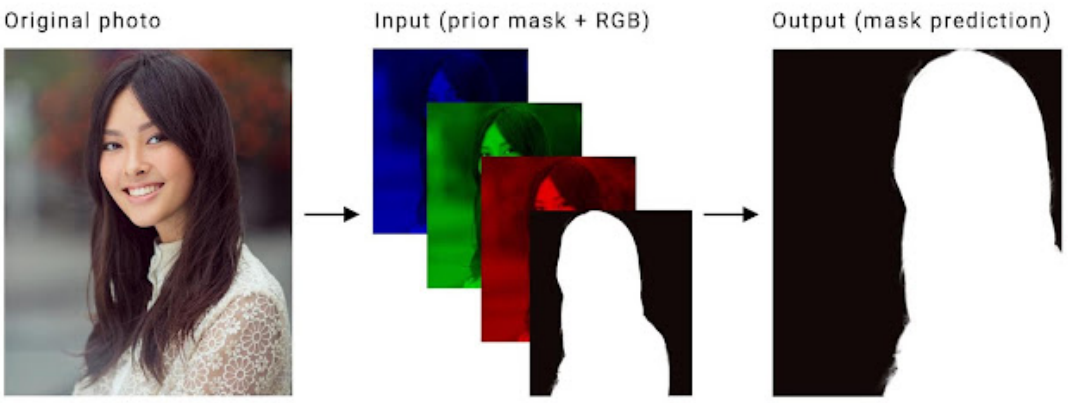

Figure 2.3: Adding a binary segmentation mask as an extra channel to an RGB image. Image taken from Google AI's blog post [2].

MaskTrack [22] uses many of the same techniques to build a VOS model. Using DeepLab [6] as their starting architecture, MaskTrack adds a fourth input channel that is used for a previous frame's mask. This is trained with an offline and an online stage. The offline stage involves training on single images. For the fourth input channel an altered form of the ground truth mask is used. The alterations include affine transformations, non-rigid deformations via thin-plate splines, and dilation of the masks to simulate noise from the previous frame. The online stage involves fine tuning this model to a specific object. This is done by repeatedly augmenting and training on the segment annotation from the first frame of a video. MaskTrack achieved high quality VOS results without needing to consider more than one frame at a time.

Using video data can also be avoided by using unsupervised learning. In [20], camera motion, depth estimation and camera intrinsics are used to provide an extra supervision signal via consistency loss

$$\ell_{L1} = \sum_{x,c} W(x,c) \left\| \hat{L}_{t+1}(x,c) - L_{t+1}(x,c) \right\|_1 \tag{2.5}$$

where x is a pixel, c is a class, L_{t+1} is segmentation based on a single frame, $\hat{L}_{t+1}$ is segmentation based on depth and motion estimation, and W is a weighting function

To integrate information from multiple previous frames, [21] uses an encoder decoder architecture with a memory module in the latent space. Local attention mechanisms are used to incorporate information from the memory module into the current frames prediction.

Accuracy can also be improved through user interaction. One example of this can be found in [7] which allows users to refine and correct segmentation at any frame of a video using scribbles or clicks (which are much faster than densely annotating every pixel in an image). The users correction is captured using the difference in the selected mask before and after interaction. This difference is fed into the fusion module as guidance for improving segmentation accuracy.

Another major area of research in VSS revolves around decreasing compute times of these models [33]. Shelhamer *et al.* [26] use a clock to turn off parts of the network for select frames. Periodically cutting out branches of the network also cuts out large chunks of unnecessary computation and hence speeds up the segmentation model. SwiftNet [32] achieves speed improvements by performing object matching to compress temporal redundancy.

2.2 Glass Detection

Detecting glass in RGB images at first appears to a simple semantic segmentation problem, however, it is complicated by the fact that objects can appear behind glass. Due to glass's transparency, objects behind the glass, not the glass itself, may end up being segmented. Humans when perceiving glass will typically look for context clues in the environment. Frames and glass edges can all be used towards this end. On a glass surface itself, reflections, smudges, discolourations, and condensation can also be used.

In 2020 Haiyang Mei *et al.* [17] published the Glass Detection Dataset (GDD), the first large-scale dataset specifically for glass detection. GDD contains 3916 images of glass and corresponding binary masks. Three example image/mask pairs are shown in Figure 2.4.

The paper also presents the convolutional neural network GDNet, which uses ResNet [13] as its backbone. Features from different depths are passed into their custom Large-field Contextual Feature Integration (LCFI) block and processed in parallel. These blocks produce masks based on high and low level features. The masks are then merged in order to produce the final segmentation. A pixel at the center of a window may have no clues in its immediate neighbourhood that it should be classified as glass, long range dependencies between pixels hence becomes important. As with with MirrorNet [38], the network uses dilated convolutions and attention, via a CBAM module [36], so that long range dependencies can be captured if necessary.

The following year Jiaying Lin *et al.* [15] improved upon both the model and the dataset. GSDNet

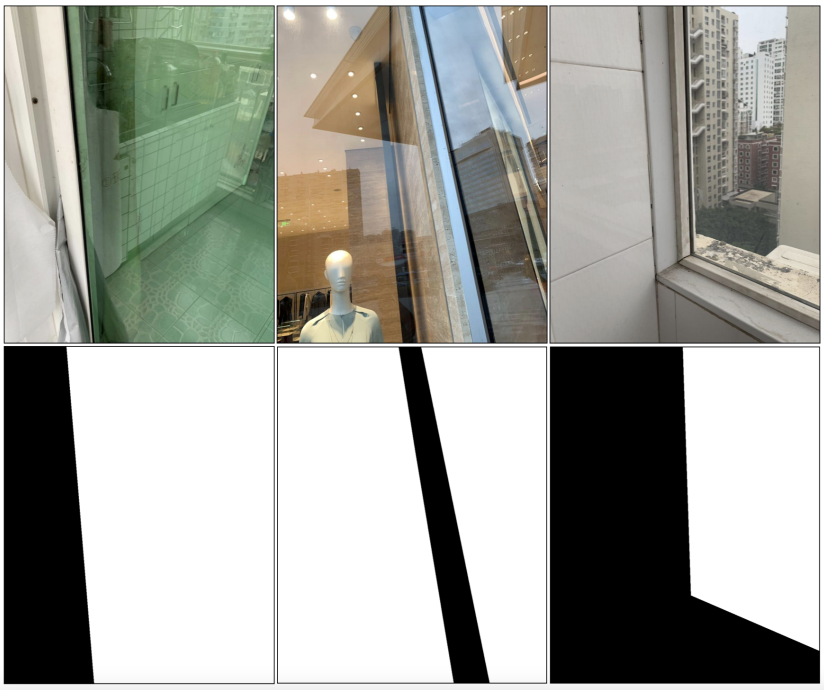

Figure 2.4: Examples from the Glass Detection Dataset (GDD). The top row shows RGB images containing glass. The bottom row shows binary masks which indicate where glass is located in the images.

improves upon GDNet by also taking into account reflections. The two networks share many characteristics; they both use ResNet as a backbone, they both use dilated convolutions, and they both combine masks calculated using features extracted from different levels. However, unlike GDNet, at the end of the network GSDNet has a Reflection-based Refinement Module (RRM), a U-Net [24] like module that takes the current unrefined mask and the original RGB image as input. As it's outputs the RRM produces both a refined mask and an RGB image containing only the isolated reflections of the glass.

GDD mostly contains close up pictures of windows (see Figure 2.4) which meant that glass far in the distance would often be ignored. Glass Surface Dataset (GSD) includes close-up, medium and long shots from a variety of scenes. There are 4012 image/mask pairs. The training set also contains the isolated reflection images of the glass which are needed for back propagation but not for inference. These are generated using the single image reflection removal methods described in [41]. See Figure 2.5 for two image/mask/reflection sets taken from GSD. The separated reflection images are not perfect here because no method completely removes reflections, however multi image methods (using at least three images from the same scene) make the problem more tractable [31]. Videos could exploit this fact to create more accurate reflection images, which could help improve overall accuracy.

Glass detection can be seen as a subset of another more general problem, transparent object detection. The Trans2Seg network and corresponding Trans10K dataset have been used to tackle this problem [37]. The dataset contains more than just glass, however each category of thing is labelled (e.g. window, jar, cup). This also shows that transformers are a viable way of segmenting glass.

Robot navigation needs to detect glass in order to operate safely. This tends to be done not using

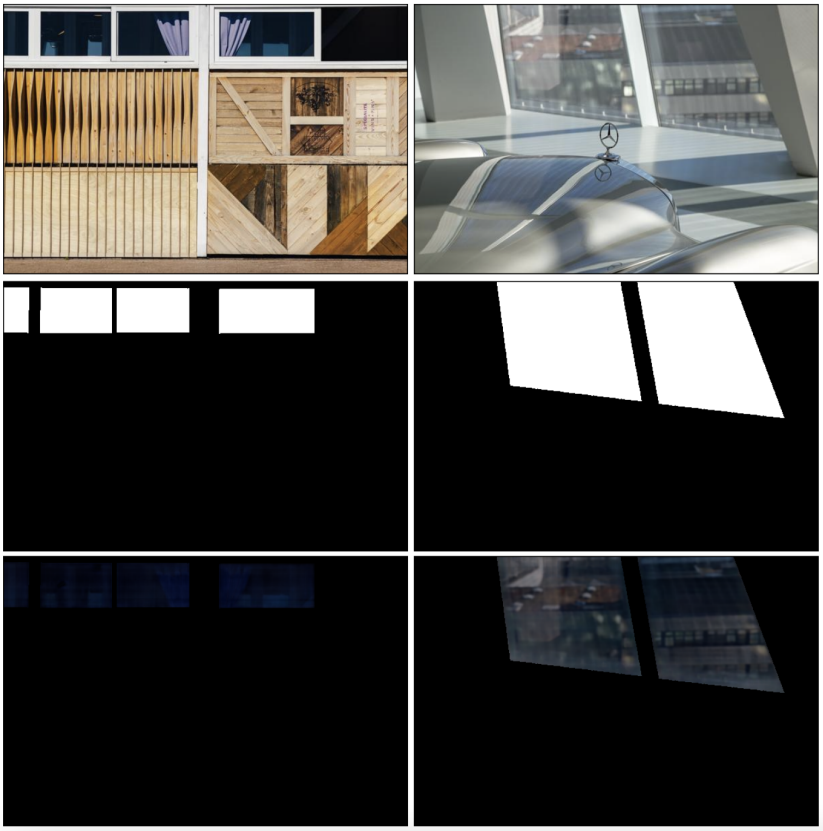

Figure 2.5: Example images taken from the Glass Surface Dataset (GSD). The top row contains RGB images, the second row contains the corresponding binary masks, and the bottom row shows the isolated glass reflections.

RGB images, but via sensors. [34] uses a laser range finder to measure the distance to objects and map out the environment. Glass may be found with a laser range finder because of the unique properties of the reflections that glass produces. [35] detects glass using ultrasound and lidar data. Ultrasonic sensors will find the distance between a robot and glass, whereas lidar sensors will measure the distance to objects behind the glass. The resulting difference in measured distance can be exploited to detect glass. In [14], a navigation system for visually impaired people is built that can detect glass. This is done by comparing the distances found by an RGB-D camera and an ultrasonic sensor. If there is difference between the two measurements then glass is detected. This method correctly detects glass over 90% of the time. So glass detection can be done, if depth sensors are available. If RGB image data is all that is we have, then these methods cannot be used.

PGSNet [40], done contemporaneously to this project, is similar to GDNet but adds two new modules. The Discriminability Enhancement (DE) module purifies features to ensure more discriminative representations. The Focus-and-Explore Based Fusion module (FEBF) highlights commonalities and explores differences between features at various levels. The accompanying dataset, called HSO, contains 4852 image/mask pairs and focuses on home scenes. Unfortunately, at time of writing, the code and dataset have not been made publicly available.

2.3 Active Learning

There are several existing datasets that contain images of glass and corresponding masks. GDD, GSD and Trans10k can all be used to train a model on the glass detection problem for images. However, there are no datasets for glass detection in videos. One way to circumvent the lack of labelled video data is to gather a small pool of videos containing glass and then use active learning to make best use of this data.

Active learning allows our learning algorithm to choose the data from which it will learn. An active learning system will select unlabelled instances for which the model is unsure and then ask an oracle (e.g. a human annotator) to label these instances. By strategically choosing these instances, the model will perform better with less training data. The process by which we choose these instances is called the query strategy [25]. For a toy example of how active learning can be used to learn more with less data, see Figure 2.6. Figure 2.6a shows blue and orange points making up two clusters. An optimal decision boundary separates the clusters. Let say we take these labels away, like in Figure 2.6b, and then ask a human to label ten random points. From these ten labelled points we can then recalculate our decision boundary. The decision boundary for Figure 2.6b is reasonable but quite far from the true decision boundary. In Figure 2.6c we have asked a human to label ten points chosen according to some query strategy. The decision boundary calculated using these ten points is much closer to the true decision boundary. This demonstrates how much the training data effects the quality of the final model. Both Figures 2.6b and 2.6c use the same amount of data, however through the use of an effective query strategy we have obtained a far better result.

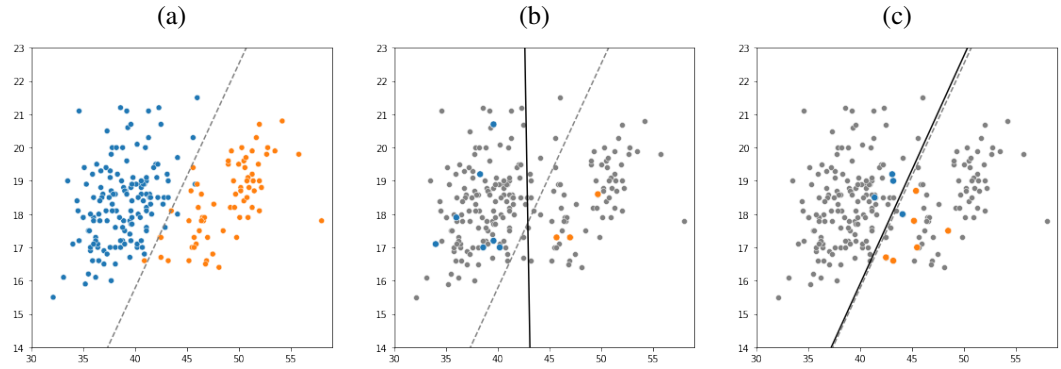

Figure 2.6: A toy example of active learning. Plot (a) shows an orange cluster and a blue cluster. They are separated by a decision boundary (shown in gray) that has been computed using an SVM. Plots (b) and (c) show the same clusters except with the labels taken away. We then choose ten points, get their labels, and recalculate the decision boundary using these points. For the plot (b) points are chosen randomly. For the plot (c) points are chosen by some query strategy. With the same amount of data the plot (c) achieves a more accurate decision boundary.

There are three main active learning scenarios; membership query synthesis, stream-based selective sampling, and pool-based active learning. In membership query synthesis the active learning system generates a new example for the oracle to label. This can have issues if the generated example is ambiguous or has no obvious label. Stream-based selective sampling involves sampling instances from the

distribution we are trying to learn one at a time. The active learning system must then choose to either query or discard this instance. Pool-based active learning is where we have a large pool of unlabelled data. Our model then selects a few points to be labelled by the oracle [25].

Active learning's aim is to reduce the amount of training data needed. This can be particularly relevant to segmentation because labelling every pixel of a image is a time consuming process. For example, producing a single image's segmentation map in the Cityscapes dataset [8] took an average of 1.5 hours. The creation of the Microsoft COCO dataset [16] took 70,000 worker hours to complete. Furthermore, active and deep learning complement each other well. Deep learning models are particularly data hungry and active learning can be used to reduce the amount of training data needed [23].

[30] applied active learning to segmentation by querying the semantic labels of superpixels. A conditional random field (CRF) connects superpixels within and between images. Costs are then updated using expected change (an active learning technique that looks to maximise the training gradient [25]). This technique achieves 97% accuracy of a fully supervised model with only 17% of the superpixel labels.

PixelPick [27] is a pool-based active learning framework that can be used to improve the performance on semantic segmentation models. It has two main steps which are repeated: (i) Retraining the segmentation model on all of the labels that have been provided by the human so far; (ii) Sample new uncertain pixels and get their label from the oracle. Only a few pixels per image are labelled. Labelling can be done very quickly using their custom GUI (one key press assigns one label). In their experiments they found that, in terms of segmentation accuracy, it is better to sparsely annotate many images as opposed to densely labelling a few images. The framework also proved robust to errors caused by annotators. Several query selection functions are suggested, however margin sampling was deemed to be the most effective. Margin sampling chooses pixels which have the smallest difference between the first and second most likely labels. For multiclass semantic segmentation margin sampling is given as:

$$u^*_{MS} = \underset{u \in \Omega}{\operatorname{argmin}} \left(\underset{c_1 \in \{1,\ldots,C\}}{\operatorname{argmax}} \, \hat{y}_u \left(c_1\right) - \underset{c_2 \in \{1,\ldots,C\}}{\operatorname{argmax} 2} \, \hat{y}_u \left(c_2\right) \right) \tag{2.6}$$

where

Ω = unlabelled image

u = pixel

C = number of segmentation classes

$\hat{y}_u$ = prediction for a given pixel

$argmax2$ = second highest

In the case of binary semantic segmentation we only have two classes. Therefore, we can remove the argmax terms by taking the absolute value. Since $\hat{y}_u(c_2) = 1 - \hat{y}_u(c_1)$ the expression can be further simplified to:

$$u^*_{MS} = \underset{u \in \Omega}{\operatorname{argmin}} \left| \hat{y}_u(c_1) - 0.5 \right| \tag{2.7}$$

This equation is essentially looking for predictions that are closest to 0.5, which makes intuitive sense. If *glass* is labelled 1 and *not glass* is labelled 0, then a prediction of 0.5 is where our model is most uncertain. Figure 2.7 shows ten pixels chosen by margin sampling when running a glass detection model. Most of the selected points are on the perimeter of the glass because their predictions are closest to 0.5.

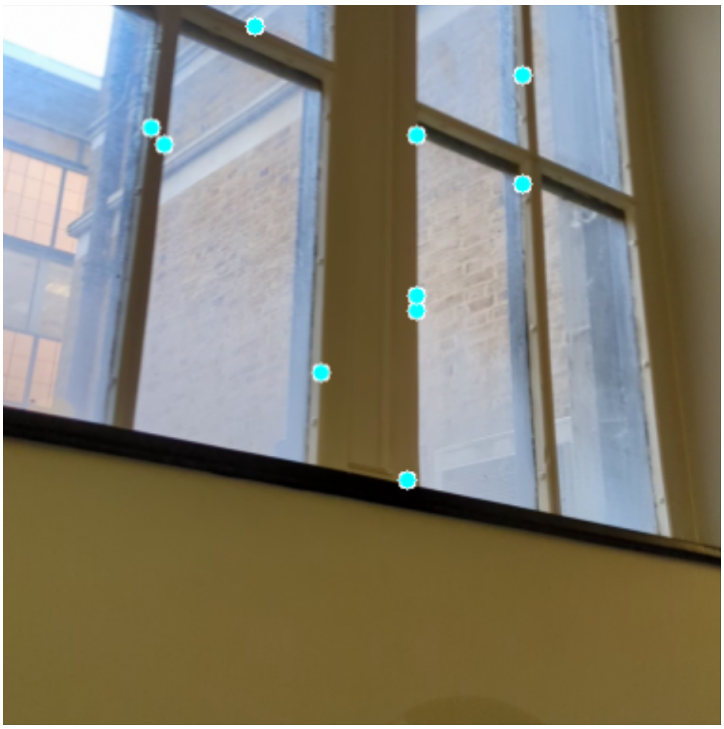

Figure 2.7: Points chosen by running margin sampling on a GSDNet result. The result itself would be a mask, but we have shown the points overlaying the original image. The ten chosen points are highlighted in cyan.

Entropy is another popular query strategy, however in the binary case this also reduces to "which predictions are closest to 0.5". Both entropy and margin sampling are examples of uncertainty sampling, a common type of query strategy. Uncertainty sampling looks to label instances about which a model is least certain [25].

Query strategies are currently a prominent focus for deep active learning researchers [23]. For instance, [39] attaches a loss prediction module onto a CNN which learns to predict the loss for unlabelled images. Higher loss corresponds to a worse prediction, therefore inputs with high predicted loss are good candidates for labelling.

In [11], a new query strategy is proposed for segmenting biomedical images. This is well suited to active learning because providing these labels requires lots of expert time. This method looks to label patches that are both very uncertain and very common throughout the dataset.

Attempts have also been made to apply active learning to video segmentation. [9] again makes use of a graph of superpixels which are connected within and between frames. Active learning based on selecting the most uncertain frames gives better results than selecting random or sequential frames.

Video segmentation is performed in [29] by jointly learning segmentation and optical flow via a neural network. This method also takes into account how uncertain each frame is when performing query selection.

Chapter 3

Implementation

Our method aims at building on the existing GSDNet architecture found in [15]. When the original model processes videos frame by frame, results have a good degree of accuracy, but exhibit flickering behaviour for difficult to classify objects. While GSDNet processes single images, our model, called GSDNet4Video, processes a sequence of frames. The focus in creating this model is primarily on temporal stability. However, accuracy, time, and memory are still relevant factors considered in the design.

The implementation is split into two key sections. In Section 3.1, we detail modifications made to the network and how it was trained. From this we have a baseline model. Section 3.2 describes how we have tried to fine tune this baseline using active learning.

Code for our project and instructions on how to use it can be found in our GitHub repository at `https://github.com/harveymannering/GlassDetectionInVideos`.

3.1 GSDNet4Video

GSDNet [15] was, at the time of this project's commencement, the best performing glass detection neural network for which code was publicly available. Therefore it has been chosen as the network from which we will build. Since it uses reflection removal as part of its training process, and reflection removal is more tractable with multiple images, the network is a natural choice for extending into the video domain. Although we have not leveraged multi image reflection removal in this project, it remains a promising avenue to be explored in future work.

Two main modifications have been made to the GSDNet. Firstly, following [2], a fourth channel has been added to the input, the first three channels being the RGB channels of an image and the fourth channel being the segmentation mask from the previous frame in a video. As with the original model, all input channels are given to ResNet and the RRM. Secondly, the backbone (originally ResNeXt-101) has been replaced with the much lighter weight ResNet-18. This reduces train time, inference time and the number of parameters significantly.

Features from ResNet are extracted at five different depths. The deepest features are given to the Rich Context Aggregation Module (RCAM). RCAM contains four parallel atrous convolution branches with dilation rates of 1, 2, 4, and 8 to cover different scales. The differences between these four sets of features are found, concatenated, and enhanced using context-wise and channel-wise attention. Follow-

ing the RCAM block is a decoder which contains upsampling and convolutional layers. From this we get a mask that is multiplied by the second deepest set of features. This process is then repeated for every set of features from ResNet. By repeatedly integrating previous backbone features we are continually refining the segmentation mask. The result of this process is concatenated with the shallowest set of features, as well as the RGBM input and given to the RRM.

The RRM has a U-Net like architecture. Its purpose is to refine the current prediction using an isolated reflection image (generated prior to training) to guide the process. The final output of the network is a one channel segmentation map and a three channel image displaying just the reflections of the glass. Both outputs have the same height and width as the input. A diagram of the network is shown in Figure 3.1.

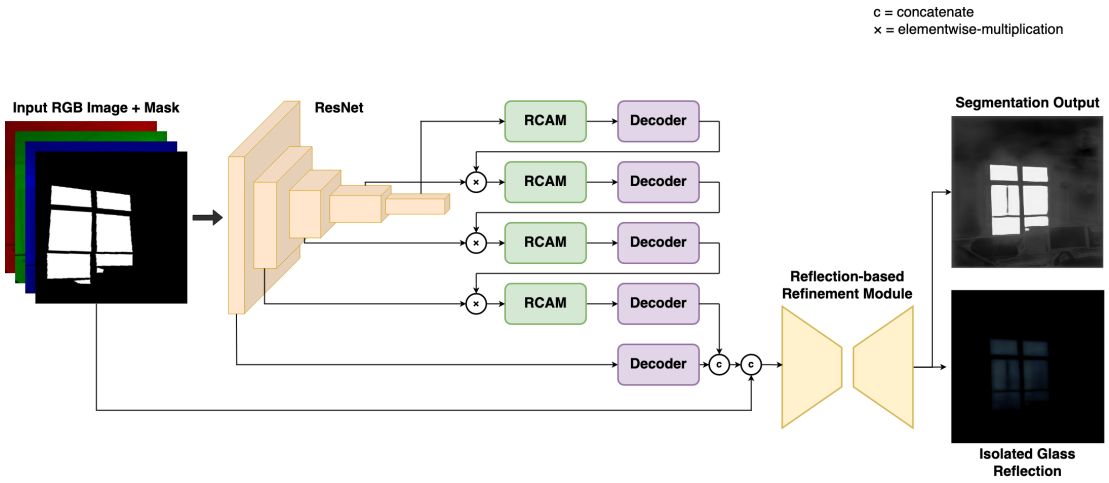

Figure 3.1: Network architecture of GSDNet4Video. Features are extracted from the four channel input and processed using the RCAM and decoder blocks. The RRM is then used to refine the glass prediction. As a side effect, an isolated reflection image is also produced as an output.

3.1.1 Backbone

In order to speed up both training and inference times, the original ResNeXt-101 backbone has been replaced with ResNet-18. This nearly halves the total number of layers in the network. Unless stated otherwise, all experiments use ResNet-18, however a ResNeXt-101 variant of the model also exists. The run times of the two variants will be compared in Chapter 4.

Jiaying Lin *et al.* [15] used ResNeXt-101 trained on ImageNet. Unfortunately, since the feature extractor has been modified to have four input channels, a pretrained backbone cannot be used and the model must be trained from scratch. This means that significantly more epochs must be used before a minimum is reached.

3.1.2 Extending to Video

Converting GSDNet into a video model requires few architecture changes. The essential change is simply to add a fourth channel to the input for a previous frame's mask. The difficulty here lies in the fact that we do not have annotated video data. However, for most videos, consecutive frames will tend to have

similar masks. Two consecutive frames in a given single shot video typically display the same scene from a slightly altered perspective. Therefore, we can fabricate a previous frame's mask by performing a minor transformation on the ground truth mask. The minor transformations we perform are shown in Figure 3.2b and 3.2c. These include cropping to simulate panning, cropping to simulate zooming and affine transforms to simulate perspective shifting.

Using the previous frame's result an input provides a strong clue as to how the current segmentation mask should look and can therefore improve accuracy. However, if previous frames contain misclassifications, errors can persist across multiple future frames. It is therefore important that the network does not rely too heavily on the previous frame's mask. It must learn to discard some masks which are not useful. In this way the network can recover from erroneous classifications. For this, three major transformations have been implemented; mirroring horizontally and vertically, empty masks, and full masks (see Figure 3.2d-f). These are very different from ground truth and are not useful for prediction, as a result the network learns to ignore them. The empty mask (containing only zeros) also trains the network to work correctly on the first frame of a video.

On top of these augmentations, Gaussian blur is applied to half of all masks to simulate noise in the previous frame's result. All prior mentioned augmentations are applied to masks only. Mirroring in the vertical is randomly applied 50% of the time to images, masks, and isolated reflection images in order to increase the variety of the dataset.

3.1.3 Training Procedure

The model was implemented using PyTorch. The overall architecture was based on the original GSDNet code[1]. Our evaluation files also draw from this code base. The implementation of ResNet-18 was taken from an unofficial re-implementation found on GitHub[2]. The loss function uses Lovász hinge loss code that can also be found on GitHub[3]. Code for the dataloader and training was written from scratch.

Hyperparameters SGD was used to optimise the models with a learning rate of 0.00125, momentum of 0.9, and weight decay of 5×10^{-4}. To ensure memory was not overloaded during training, images were resized to 384×384 and a batch size of 4 was used. The frequency of the various augmentations described above should mimic what would be seen in real videos as closely as possible. We randomly chose an augmentation for each training example. We found that using affine transforms 25% of the time, cropping 25% of the time, mirroring horizontally and vertically 25% of the time, and empty/full masks 25% of the time worked well. This trains the model to expect to ignore half of the previous frames' masks. This strikes a good balance between using the mask to increase temporal stability and ignoring the mask when it is not helpful.

Loss We have used the loss function as described in [15]. The loss for the isolated reflection images is calculated as the mean squared error between the true isolated reflection R_{global} and the predicted reflection R_{glass} (with each multiplied by the ground truth segmentation mask G). Lovász hinge loss L_i is used to calculate the loss for a segmentation mask, thus directly optimising for IoU. The overall loss

[1] https://jiaying.link/cvpr2021-gsd/

[2] https://github.com/kshitijkumbar/LearningBasedMethods/blob/45a031df5c1e2fd7f0c6f377c1265f3540736be2/resNet/resnet.py

[3] https://github.com/bermanmaxim/LovaszSoftmax/blob/master/pytorch/lovasz_losses.py

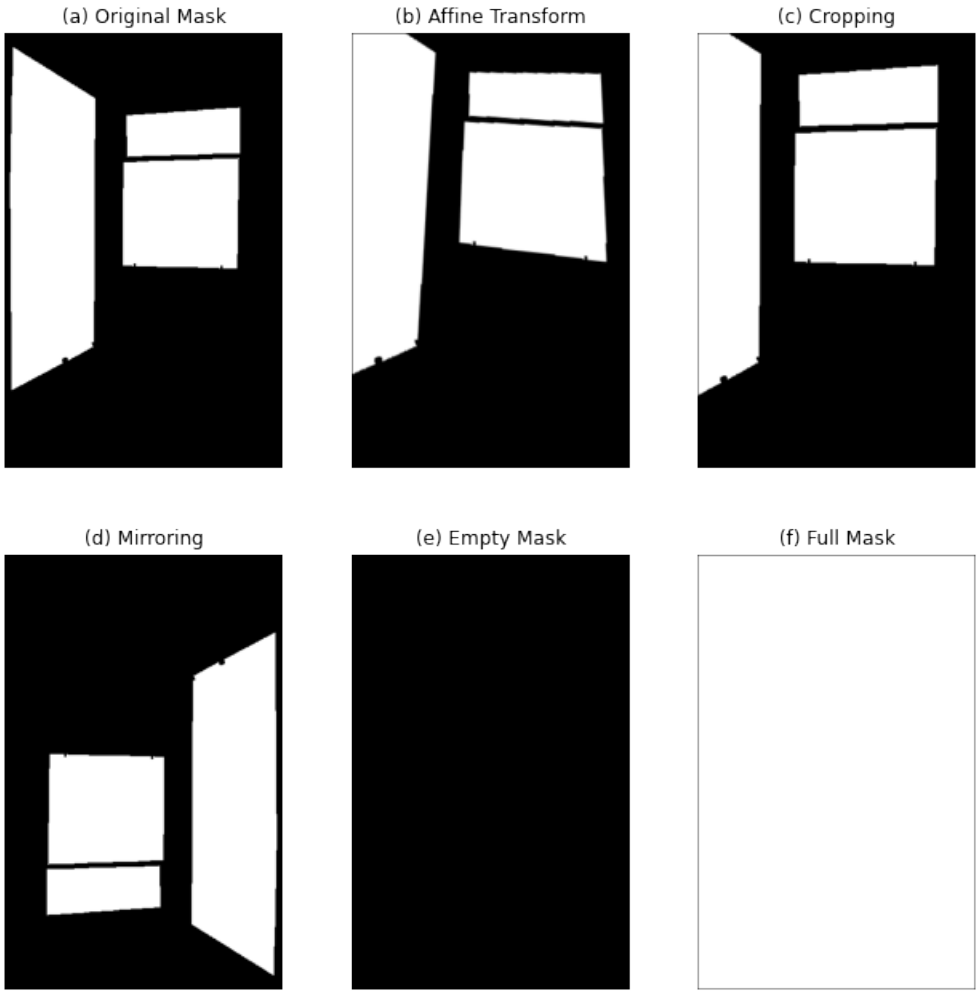

Figure 3.2: Augmentations used to train GSDNet4Video. The black borders around the masks are for display purposes only. (a) is the original ground truth mask. (b) and (c) are examples of the minor transformations. (d) - (f) are major transformations.

is

$$\text{Loss} = \lambda \left\| R_{\text{global}} \otimes G - R_{\text{glass}} \otimes G \right\|^2 + \sum_{i=1}^{N} w_i L_i \tag{3.1}$$

where $\otimes$ is element wise multiplication. Variables w and λ are weights which have all been set to one. N in the number of masks. While we only have one output mask, there are also several intermediate masks which are output after each RCAM+Decoder block (see Figure 3.1). All of these masks can be included in the loss function. In our case $N = 5$.

Dataset GSD was used as training data for the model. The dataset contains 3285 training images and 812 test images. However, only 2710 training images had corresponding reflection images. As a result, only these 2710 were used for training. In corresponding with the original authors, it was found that the dataset initially included 2710 training images and was then expanded for CVPR 2021. Furthermore, the remaining 575 training images with no corresponding reflection images were deemed suitable for use as a validation set. For training we have therefore used 2710 training images, 575

validation images, and 812 test images. After bringing this to the author's attention, the missing reflection images have now been added to the dataset online.

3.2 Active Learning

For the purposes of fine tuning GSDNet4Video, we propose two new query strategies based on "flicker" between frames. Flicker is where an object changes classification from one frame to the next. If an object's classification changes from *glass* to *not glass*, or vice versa, between frames, then flicker has occurred. If our model is changing its classification with regard to an object in the scene, we can view this as a type of temporal uncertainty. We therefore target these flickering areas with our query strategy in the hope of improving temporal stability. This query strategy has two variants. The one presented in Section 3.2.1 we will call *flickering regions* and the one described in Section 3.2.2 we will call *flickering pixels*.

These techniques were initially applied to real video data, however due to limitations in both data and time, synthetic videos have been used for training instead. How these synthetic videos are generated will be described in Section 3.2.3. That being said, the query strategies and training procedure detailed in this section were built with real videos in mind. It is our hope that if a large enough dataset of videos containing glass could be gathered, these method could be applied to fine tune GSDNet4Video on real video data.

3.2.1 Flicker Detection

We are defining flicker here as an object that changed classification between frames, that is, changes from *glass* to *not glass* or vice versa. Firstly, we must determine how pixels are moving between frames. For real video data, we can use optical flow, RAFT [28] in particular does a good job here but is not perfect. For an example of flicker detection that uses RAFT to track pixels see Figure 3.3. Creating synthetic videos from still images has the advantage of knowing the exact optical flow.

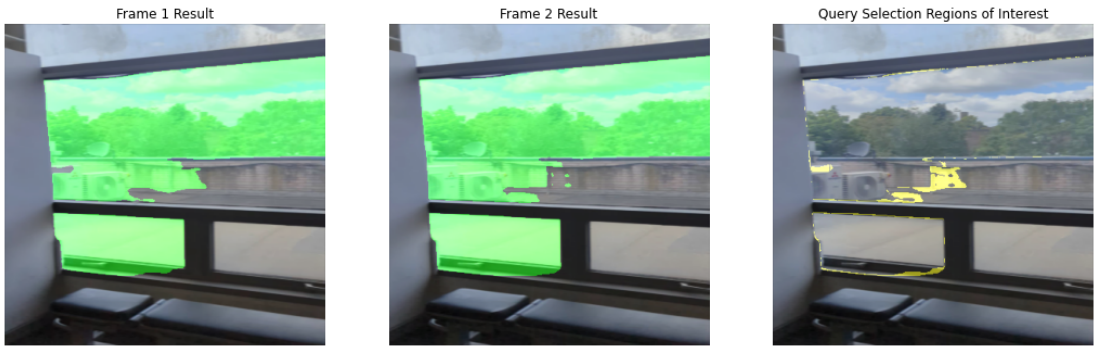

Figure 3.3: Flicker detection in a real video using RAFT. The first two images are two consecutive frames in a video. Highlighted in green are the areas that have been detected as glass. The final image shows the second frame with flickering regions highlighted in yellow.

Figure 3.4 shows the components used to find flicker between two images. The result is a binary image where 1 represents a flickering pixel and 0 represents a pixel that has a consistent classification.

This flicker detection mask has the same height and width as the input images. We call this query strategy *flickering regions*. Given our pretrained GSDNet4Video model, and two consecutive frames in a video, we calculate flicker like so:

(1) Run the two frames through GSDNet4Video, the second frame should take the results of the first frame as its fourth channel. From this we get mask 1 and mask 2.

(2) Calculate the optical flow between the two frames.

(3) Translate pixels in mask 1 to where we would expect them to be in mask 2 based on the result of the optical flow. This will create a predicted mask 2.

(4) Find differences between mask 2 and the predicted mask 2, ignoring pixels for which no optical flow information existed. Set pixels that disagree to 1 and pixels that agree to 0. This results in a final flicker image which highlights any areas that flicker between the two frames.

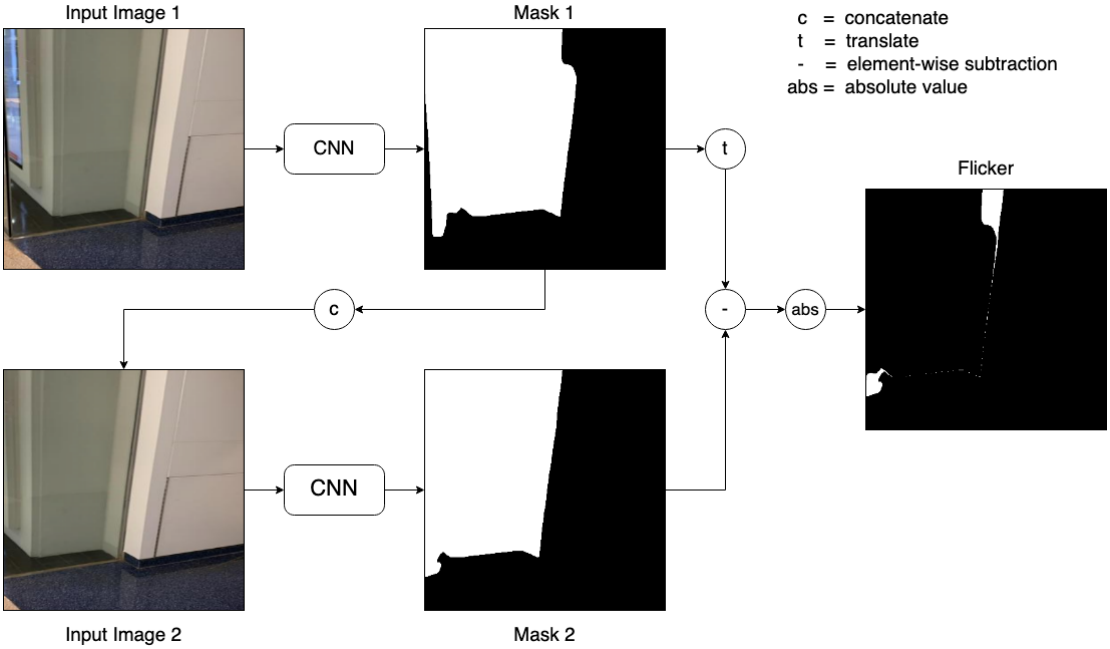

Figure 3.4: This diagram shows how the flicker between two images is found. The CNN block is referring to our GSDNet4Video model. A sequence of two frames is created by cropping images in GSD. The segmentation masks, calculated in order, are also shown. The flicker between the two masks is calculated by finding where pixels in masks 1 and 2 differ.

Once this procedure has been followed we can use the highlighted flickering regions for training. These are the areas that are changing classification between frames meaning our model is uncertain of their true class. We can target these uncertain areas by only training on the pixels highlighted in the flicker mask.

3.2.2 Choosing Pixels

As seen in [27], only a few pixels are needed in order to fine tune segmentation models. Labels for individual pixels are cheap to collect in comparison to masks. No labelled video data is available for this problem, so training on pixels is useful because collecting labels for pixels is relatively fast.

Starting from the flicker masks described in Section 3.2.1, we have developed a method for picking representative pixels. These pixels can then be used in training. We will refer to this query strategy as *flickering pixels*. To choose these pixels we use the following process, each stage of which is shown in Figure 3.5:

(1) Repeatedly perform erosion on flicker mask until all pixels are 0. This is done with a 3×3 structuring elements where all values are set to 1.

(2) Randomly choose a pixel that is valued 1 in the mask immediately before all pixels have been eroded away.

(3) In the original flicker mask, a 30 by 30 box of pixels centered on this chosen point are set to 0 to prevent pixels in this area from being chosen again.

(4) Repeat steps (1)-(3) until the desired number of pixels has been reached.

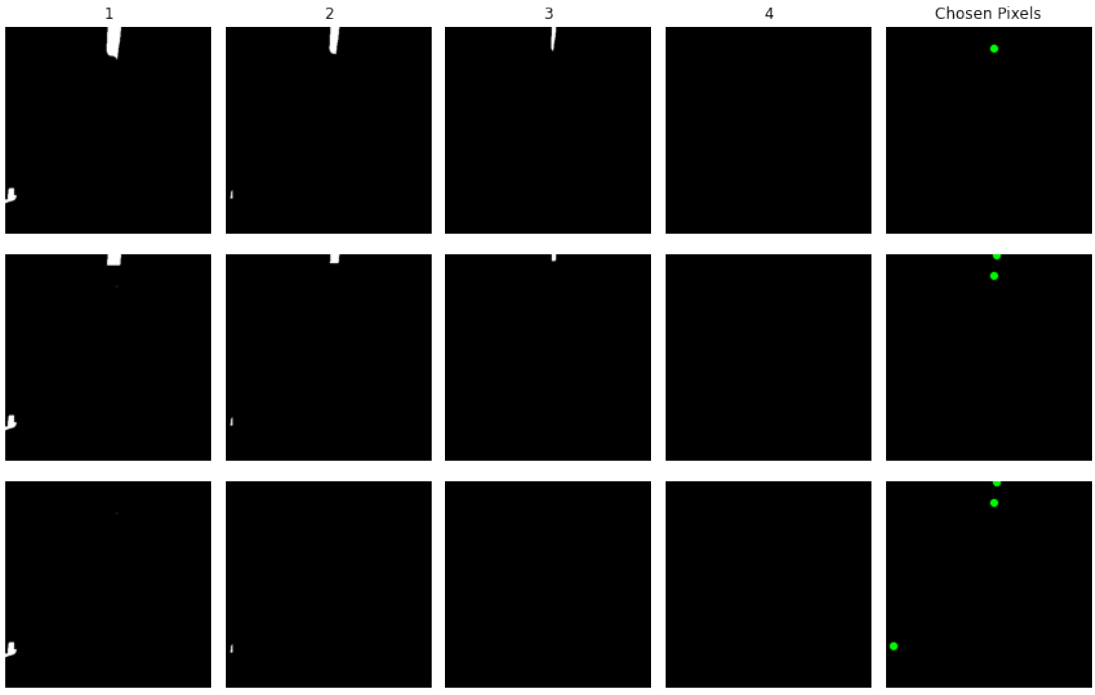

Figure 3.5: The process of choosing pixels from flickering areas. Each row shows the picking of a single pixel. From left to right, the flicker mask is gradually eroded away until there is nothing left in the 4th column. The final column shows which pixels up until that point have been chosen.

Pixels central to flickering regions are chosen because they are likely to be representative of the whole area of flicker. Practically, these pixels are also likely to be easier to label than pixels near the

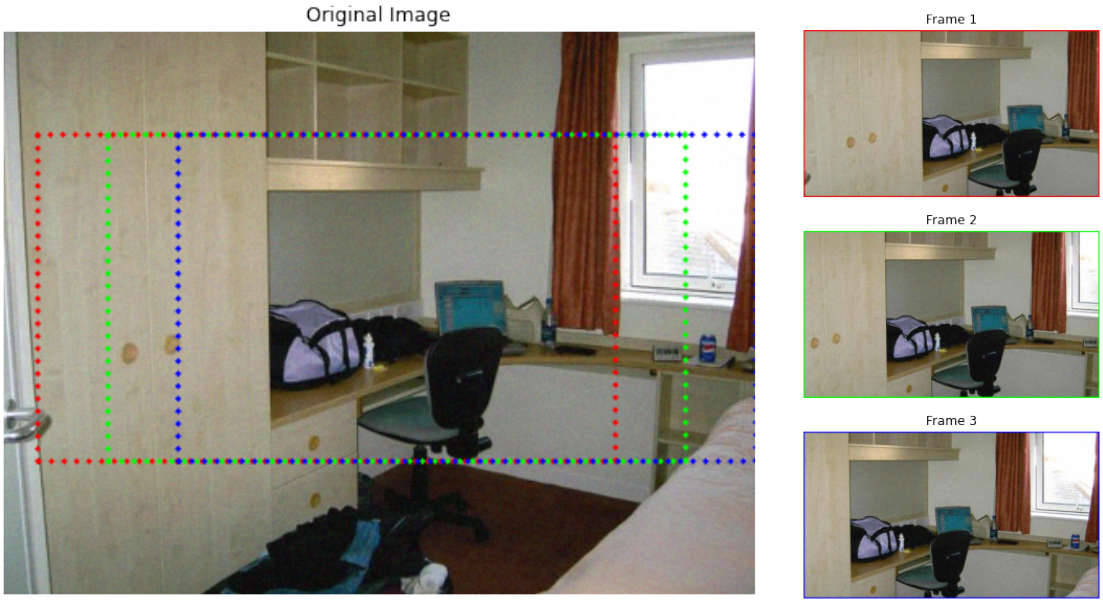

Figure 3.6: Generation of synthetic video. Original image is taken from GSD. By cropping this image in three different positions we create 3 frames that simulate a camera panning sideways across a scene.

edge of flickering areas (which often cover the perimeter of a glass region).

The process of zeroing out a 30 by 30 block of pixels eliminates the possibility that pixels from that region can be picked again. As mentioned in [3], the effectiveness of uncertainty based strategies can suffer if the sampled points are not diverse. Since neighbouring pixels tend to be similar, diversity here can be read as spatially diverse. This is why we do not want to chose pixels too close to each other.

3.2.3 Synthetic Videos

In order to make use of these flickering pixels we will be generating synthetic videos using images taken from GSD. This has the following advantages over using real videos. Firstly, we have the ground truth labels for every image, meaning that we have an omniscient oracle. We do not need a human oracle to label these flickering pixels because the oracle already knows their ground truth values. Having this omniscient oracle both speeds up the active learning process and removes the risk of labelling mistakes being made. Secondly, we know the exact optical flow between frames and therefore do not need to rely on imperfect optical flow models like RAFT.

Figure 3.6 shows how a synthetic video is created. We start with an image from GSD. A random rectangular crop is then calculated which forms the first frame of the video. A random direction is then chosen; up, down, left or right. The cropping window is moved in this direction half way to the edge of the image. We take another crop at this position which corresponds to our second frame. The cropping window is then moved all the way to the edge of the image and our third frame is captured. The ground truth segmentation masks for each frame can be created by cropping the ground truth in the same way. With these crops we have a short, three frame, fully labelled video.

In order to increase variety within the training data, we randomly augment the image before crop-

Figure 3.7: Three augmentations applied to an image taken from GSD. These augmentations are colour jittering (adding noise), converting the images to grey scale, and blurring. These augmentations are only applied during fine tuning.

ping. These augmentations follow [27]. They include colour jittering, mirroring vertically (must also be applied to the mask), blurring, and converting to grey scale. These augmentations help the model to better generalise. Figure 3.7 shows these augmentations minus mirroring. All augmentations shown maintain the position of glass in the image, as a result the corresponding mask will stay the same regardless of the augmentation. Colour jittering adds salt and pepper noise to one or more channels in the image. This changes pixel values significantly while maintaining the general structure of the image. Grey scaling is achieved by averaging all three RGB channels into one. This averaged result is then copied three times back into each of the RGB channels, producing a grey image. Blurring here is done using a Gaussian kernel. The amount of augmentation applied is random.

3.2.4 Training Procedure

Fine tuning is done using the synthetic videos described above. Each frame is run in sequence on GSDNet4Video, with the predicted mask from frame 1 being used as the fourth channel input for frame 2, and the predicted mask from frame 2 being used as the fourth channel input for frame 3. An empty mask (all 0s) is used for the fourth channel input of the first frame because no prior information is available. From this we get masks from three consecutive frames. These can be used to fine tune our model.

How is this different from the baseline training described in Section 3.1? While the dataset is the same, it is augmented in different ways. Also important is the fact that input masks used during baseline training and fine tuning will look very different. Compare how the masks look in Figure 3.2 and Figure 3.4 to see this difference. Masks used to train the baseline are transformed versions of ground truth masks, as a result, edges are well defined and shapes tend to be regular. This is how ground truth looks. However, the real output of the model has imperfections as can be seen in Figure 3.4. Shapes are not

always regular, corners are rounder and the masks generally have a blotchier look. These imperfections will not have been seen during the baseline training described in Section 3.1.

Our query strategy has two varieties. Training on the entire flickering region and training on just a few central pixels in the flickering regions. We can calculate the flicker from frame 1 to 2 and from frame 2 to 3. We are able to calculate the loss just on these flickering regions and thereby train on just the pixels that flicker. From these flickering regions we can choose pixels as described in 3.2.2. Again, we can calculate loss for only these few flickering pixels and ignore everything else. Both versions of the query strategy, *flickering regions* and *flickering pixels*, significantly reduce the amount of labels needed.

To prevent our GSDNet4Video from forgetting what it has learnt during the baseline training we interweave fine tuning training with baseline training. This means that each epoch contains two stages: (1) fine tune using the synthetic videos (2) perform training as described in Section 3.1.3 all be it with a lower learning rate. This duel training prevents the model from drifting too far from the minimum found for our baseline.

Following [27], after any label has been used for training it is saved into a database. Future epochs can then use this entire database while training. In the framework of [27] we can think of each epoch as a round. Every round, a query strategy (*flickering regions/pixels* in our case) chooses a number of pixels and sends them to an oracle for labelling. Labelled pixels are saved into the database. At the end of the round, our model is trained on the entire database. This framework ensures that we make the best use of the labels given to us by the oracle.

Hyperparameters Two separate optimizers are used, one is used for fine tuning on synthetic videos, the other is used for baseline training (see the two stage training described above). For both we use SGD with a learning rate of 1×10^{-5}, weight decay of 5×10^{-4} and momentum of 0.9. Images are again resized to be 384×384 and a batch size of 1 is used.

Loss These query strategies look to train on just binary *glass/not glass* labels and not isolated reflection images. This is because labelling a pixel as *glass* or *not glass* is a simple and intuitive process. Labelling the colour and amount of light that can be attributed to reflection for a given pixel is a much more complex and time consuming process. To make the query strategies more applicable to a real world scenario, we only use binary *glass/not glass* labels during training and not isolated reflection images. The loss function is thus simplified to

$$\text{Loss} = \sum_{i=1}^{N} w_i L_i \tag{3.2}$$

This is another reason for the dual training as described above. Without the baseline training, the RRM would quickly forget how to isolate reflections. Again, all weights w_i are set to 1.

Chapter 4

Evaluation

In this chapter we will perform experiments to test the performance of GSDNet4Video. We will also examine how effective our flicker based query strategies are in reducing the amount of training data needed for fine tuning. Section 4.1 will briefly go over real video data we have collected and how it was used to evaluate our methods. In Section 4.2 we examine the performance of the baseline GSDNet4Video model. Since no fully labelled videos exist for this problem the most important subsection is 4.2.3, which qualitatively evaluates 20 videos. Section 4.3 details our exploration of the fine tuning procedure, examining specifically the effect of learning rates, augmentations, and query strategies. All videos referenced in this section can be found at `https://harveymannering.github.io/GlassDetectionInVideos/`

4.1 Glass Video Dataset

For the purposes of evaluation, a small dataset of 20 videos containing glass has been collected. 11 clips were taken ourselves, mostly collected in the south of England on a Google Pixel 6 smartphone. The other 9 clips have been randomly selected from the Youtube-8M dataset [1]. All video clips are between 2.3 and 28.7 seconds. The mean length of a clip is 10.5 seconds. Glass type, lighting condition, scene diversity, and camera movements were all considered when constructing the dataset. This dataset contains sequences of RGB images and no segmentation masks. However, as will be described next, a small sample of pixels have been labelled.

4.1.1 Pixel Labels

As these videos are initially unlabelled, they can only be evaluated qualitatively. In order to add a quantitative element to the evaluation of these videos, random pixels have been chosen and labelled. When these videos are evaluated by a model, we can then compare the predicted labels with the ground truth labels and calculate the pixel accuracy for the few labels we have. This measure is clearly incomplete, however, it also allows us to numerically compare performance on real videos.

To sample pixels in this video dataset we use the quasi-random Halton sampling pattern. Figure 4.1 shows the advantage of using this pattern over a completely random sequence. When completely random points are used, clusters and empty spaces can appear. The Halton sequence on the other hand guarantees that points are evenly spread throughout the image. Using this sampling pattern ensures that the few

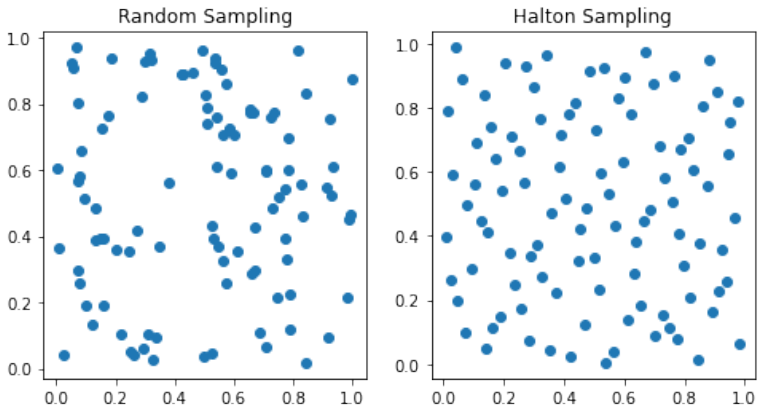

Figure 4.1: Comparison of random and Halton sampling. 100 points are shown for each method. Notice that the random sampling has more clusters and empty spaces.

pixels that are labelled in the videos are spatially diverse.

The first ten frames of every video are included. Ten pixels per frame are chosen according to the Halton sequence and subsequently labelled. In total 2000 pixels are labelled.

Figure 4.2: Modified version of PixelPick's GUI as presented in [27].

Figure 4.2 shows how pixels, once chosen, were labelled. The GUI shown is adapted from the annotation tool given in [27]. This tool allows a human to quickly label pixels with a key press. The pixel in question is highlighted in red. On the right of GUI is a box which contains our changes. The original semantic classes have now been replaced with "Glass" and "Not Glass". To assign the *glass* label to a point, a user would press "G" and to assign *not glass* they press "N". The other change here is

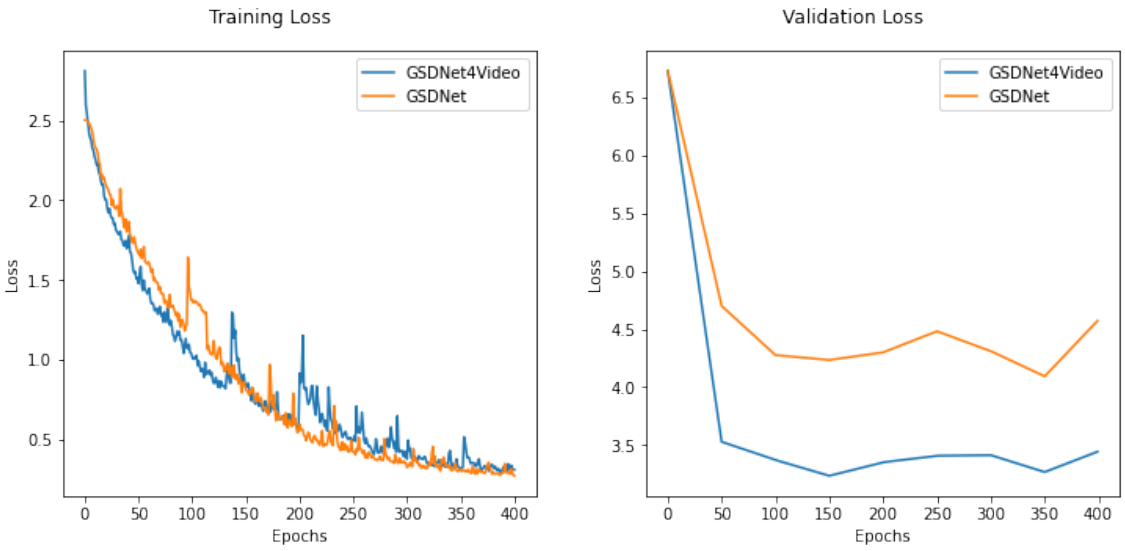

Figure 4.3: Training and validation loss curves for GSDNet and GSDNet4Video, both using ResNet-18 as their backbone.

a small window of 25 by 25 pixels showing a zoomed in area immediately around the chosen point. In practice, we found this window necessary for determining the class of pixels that live on the perimeters of glass. Once all labels for a particular image are assigned, they are saved to a database and the next image is shown. Our code for this was adapted from the original PixelPick project's code, which can be found on GitHub[1].

4.2 GSDNet4Video

We will be considering four models. The first model is the original GSDNet model given in [15]. The second model is a GSDNet4Video, which has an extra input channel for a previous frame's mask. These models have ResNeXt-101 as their backbone, so we also have variants of GSDNet and GSDNet4Video with ResNet-18 as their backbone.

Important to note is that these models have long training times. Since a four channel input was needed, pretrained weights could not be used for the backbone. Therefore all models were trained from scratch meaning many epochs were required before they converged to a minimum. After profiling the code we found that training was slowed down primarily by the data augmentation, which is performed using the ImgAug and NumPy libraries. Further development on these models should make speeding up the data augmentation a priority as these long train times are a bottleneck to development.

Both GSDNet (3 channels) and GSDNet4Video (4 channels) are trained for 400 epochs. Checkpoints of the models are saved every 50 epochs. Validation loss was later calculated for each checkpoint. To avoid over fitting the checkpoint that minimises the validation loss is taken as our final model. This means that our final model for GSDNet has been run for 350 epochs and our final model for GSDNet4Video has been run for 150 epochs. Training and validation loss curves are shown in Figure 4.3.

[1] https://github.com/NoelShin/PixelPick

Model	Parameters	Inference Time (seconds)
GSDNet (ResNeXt-101)	83,715,784	6.11
GSDNet4Video (ResNeXt-101)	83,719,607	6.02
GSDNet (ResNet-18)	14,365,848	2.79
GSDNet4Video (ResNet-18)	14,369,560	2.81

Table 4.1: The second column shows the total number of parameters for each variant of the model. The third column shows the time in seconds to run a prediction for one image (run on a MacBook Pro (2020) with M1 processor.)

4.2.1 Network Size

As can be seen in Table 4.1 changing the backbone to ResNet-18 reduces the number of parameters in the network by over 80%. The architectural changes involved in converting the GSDNet to a video model add just under 4,000 new parameters.

Average inference time for individual images is also shown in Table 4.1. These results were computed on a MacBook Pro (2020) with M1 processor and 8GB of memory. We see that changing the backbone from ResNeXt-101 to ResNet-18 roughly halves inference time.

4.2.2 Accuracy on Single Images

In Table 4.2 different variants of GSDNet are evaluated. For this, Equations 2.1, 2.2, and 2.4 have been used. These are intersection of union (IoU), pixel accuracy (PA), and F-score (F_β). For the F-score we have set $\beta = 0.3$ following [17, 15].

We have included in the top row of Table 4.2 the architecture and pretrained weights provided for the original GSDNet model. This model scores higher than ours because our training procedure has deviated in two major ways. Firstly, we have used a validation set which was taken out of the original training data. Ultimately, our training set has decreased in size by 17.5%. Secondly, due to the addition of a fourth channel, pre-trained weights could not be used for the backbone and the network needed to be trained from scratch for many more epochs. For a fair comparison, our three channel model was also trained from scratch.

GSDNet and GSDNet4Video are two very similar networks, however the data taken as input to each is different. While GSDNet takes a single RGB image, GSDNet4Video takes in that same image plus an additional channel containing binary segmentation masks. This makes comparing the two networks difficult because GSDNet4Video is given extra information to makes its predictions with. For a fair comparison, all metrics for GSDNet4Video in Table 4.2 have been calculated using an empty mask in the fourth channel. Having the fourth channel empty mimics the situation where there is no prior segmentation data available, as in the first frame of a video.

The original network and weights performed best on all metrics for the reasons stated above. The two models with ResNet-18 achieved similar performance, with GSDNet slightly better than GSDNet4Video. While GSDNet4Video is less accurate than GSDNet, it has the advantage of being more

Model	IoU	PA	F_β
GSDNet with Original Weights (ResNeXt-101)	0.8434	0.9502	0.9218
GSDNet (ResNet-18)	0.7346	0.9091	0.8516
GSDNet4Video (ResNet-18)	0.7139	0.8986	0.8458

Table 4.2: Mean Intersection of Union (IoU), Pixel Accuracy (PA), and F-score (F_β) for three models when evaluated on the GSD test set. For a fair comparison an empty mask is used in the fourth channel of GSDNet4Video. Included are both models we have trained; GSDNet (ResNet-18) and GSDNet4Video (ResNet-18). Also included is the original GSDNet using pretrained weights provided by the authors.

temporally stable, as will be discussed next.

4.2.3 Analysis on Real Videos

This section will compare two models; GSDNet and GSDNet4Video. Both have been trained ourselves from scratch on a restricted dataset of 2710 images and both have a backbone of ResNet-18. The videos are taken from the glass video dataset described in Section 4.1. There are 20 clips included. The comparison video can be found at `https://youtu.be/SYh0NOeJ81w`. The results from GSDNet and GSDNet4Video are played side by side simultaneously. GSDNet is on the left and GSDNet4Video is on the right. Glass detection is highlighted in green.

The outputs of these models will be evaluated qualitatively against each other, with two factors considered; temporal stability and accuracy. Improving temporal stability has been the motivation for GSDNet4Video, however it is also important to consider how these changes have affected the overall accuracy of the model, if at all.

Table 4.3 shows our qualitative assessment of the 20 clips. Each model has their own columns for accuracy and temporal stability. A tick has been placed in a the column of the best performing model for each video clip. Where the models perform just as well as each other an equals sign as been placed in both columns. Note that an equal sign does not mean that the outputs for both models were the same, just that each produces a similar number of mistakes.

For all videos temporal stability for GSDNet4Video was as good as or better than GSDNet. For 17 videos temporal stability was improved and for 3 it was roughly equal. From this we conclude that the inclusion of a fourth channel containing a previous frame's mask successfully improves temporal stability for glass detection. Accuracy is much less clear. GSDNet4Video performs better for 5 videos, roughly equal for 9 videos and worse for 6 videos. Accuracy may be unaffected by the inclusion of a fourth channel as the number of better or worse videos is similar. Alternatively, there may be a trade off between accuracy and temporal stability, since GSDNet recorded more wins in the accuracy column. GSDNet4Video has only 0.02% more parameters than its three channel counter part, yet it is being asked to do a lot more. Not only must it perform glass detection, but it must also decide if the previous frame's mask is relevant, and if so, somehow integrate it into the current prediction. It is most likely the case that

Video Number	GSDNet			GSDNet4Video		
	PA	**Acc.**	**TS**	**PA**	**Acc.**	**TS**
1	0.99	=		0.98	=	✓
2	0.83	✓	=	0.72		=
3	0.94	=	=	0.98	=	=
4	0.83	✓		0.85		✓
5	0.98			0.99	✓	✓
6	0.73	=	=	0.81	=	=
7	0.84	✓		0.91		✓
8	0.44	=		0.78	=	✓
9	0.71			0.80	✓	✓
10	0.89			0.87	✓	✓
11	0.96			0.90	✓	✓
12	0.96	=		0.99	=	✓
13	0.98			0.95	✓	✓
14	0.96	✓		0.88		✓
15	0.99	✓		0.98		✓
16	1.00	✓		0.99		✓
17	0.95	=		0.88	=	✓
18	0.79	=		0.91	=	✓
19	0.80	=		0.39	=	✓
20	0.95	=		0.93	=	✓

Table 4.3: Accuracy (Acc.) and temporal stability (TS) are evaluated qualitatively for 20 video and on both GSDNet (ResNet-18) and GSDNet4Video (ResNet-18). Ticks indicate the model which performed better in either accuracy or temporal stability. Were the models have performed roughly as well as each other, an equals sign is used. Pixel Accuracy (PA) for 100 random pixels in the first 10 frames of each video is also shown for each model.

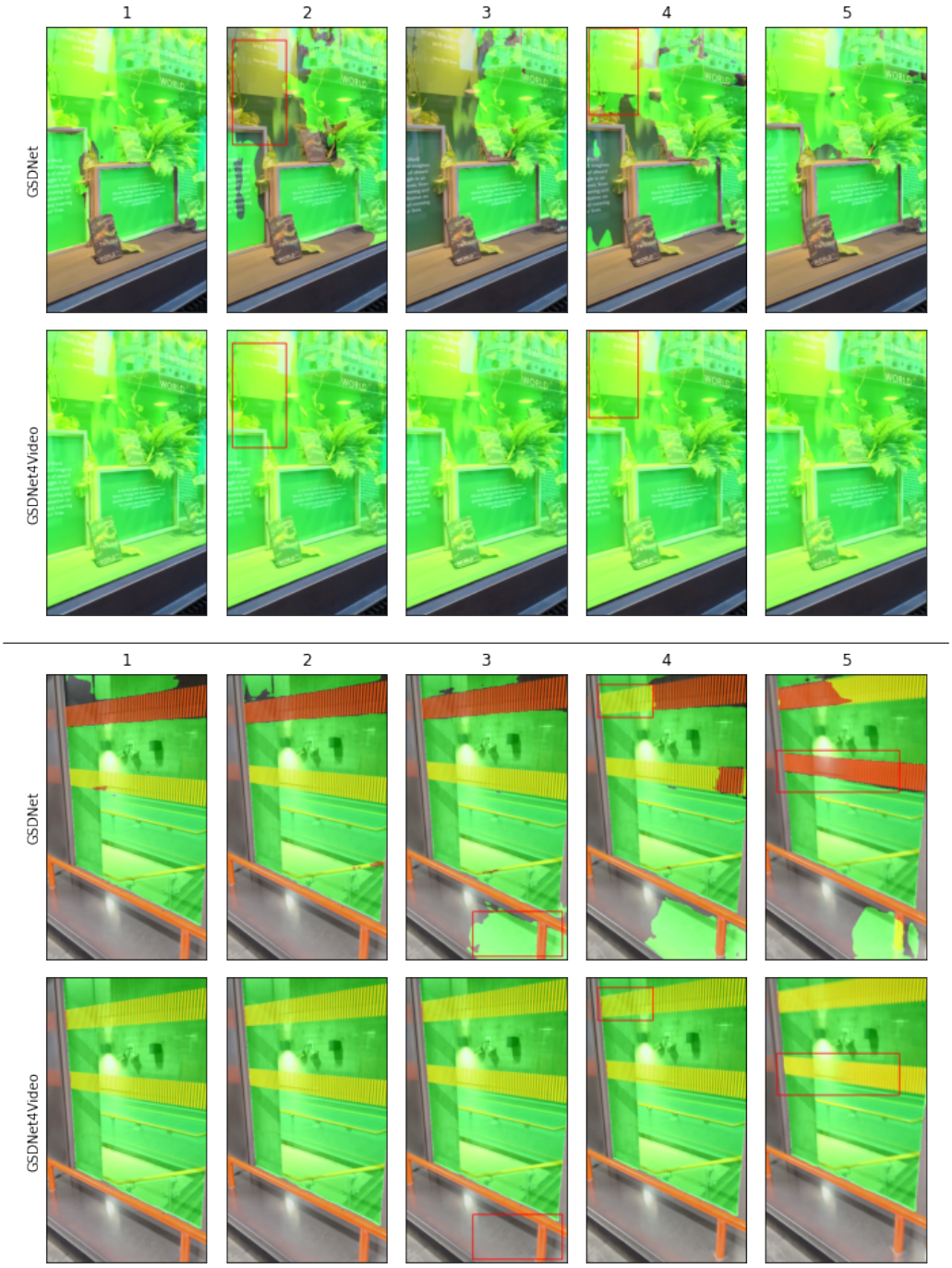

Figure 4.4: Two clips (from videos 7 and 10) made up of five frames. Each picture shows glass detection in green. The glass detection results are shown for both GSDNet and GSDNet4Video so they can be compared. Flickering areas are highlighted with red boxes.

some of GSDNet4Video's capacity is used in processing the previous frame's mask, whereas GSDNet can use its entire capacity on glass detection, thus making it slightly more accurate. This is a less clear result, hence, all that we can conclude is that accuracy has probably not been increased by adding a fourth channel.

Pixel Accuracy (PA) is also shown in Table 4.3. This is referring to the pixel labels described in Section 4.1.1. That is to say, this value does not represent the pixel accuracy for the entire video. This metric only considers ten random pixels in each of the first ten frames for each video. It only gives us a rough guide as to how well each model performed on the first ten frames of each video. According to the PA metric exactly half the videos performed better on GSDNet and half performed better on GSDNet4Video. The average PA for GSDNet was 0.880. The average PA for GSDNet4Video was 0.875. This reinforces the conclusions drawn from the qualitative analysis regarding accuracy.

A couple of common artifacts are noteworthy in these videos. Firstly, shiny surfaces that are not glass fairly regularly get classified as glass. This is most likely because the network is primed to look for reflections via the RRM. This behaviour presents itself in both GSDNet and GSDNet4Video. For examples see the marble wall at the beginning of clip 6 and the polished wood in clip 16. Specifically for GSDNet4Video, there is sometimes an issue with false positives lingering at the edge of an image after glass has exited the frame. For examples of this see clips 10 and 11.

Clips 15, 16, and 17 are particularly interesting for their length. An obvious problem with the method described in [2] is that errors propagate forward in time. Any false positives or negatives are likely to be present in the next frame too. These longer clips (all approximately 25 seconds) let us see how well GSDNet4Video recovers from misclassifications. Generally, errors will persist for as long as a shot lingers on a specific location.

Figure 4.4 shows two short sequences (5 consecutive frames). Outputs from both GSDNet and GSDNet4Video are shown. For these clips in particular we can see that temporal stability has be improved. While GSDNet has large areas of flicker, GSDNet4Video's classification stays constant.

4.3 Fine Tuning

This section will attempt to evaluate the effectiveness of our method for fine tuning GSDNet4Video. It will also detail our results from an exploration of query strategies based on flicker.

4.3.1 Learning Rate

A comparison of learning rates used for fine tuning was undertaken during development. The learning rate used to train the baseline GSDNet4Video model was 1×10^{-3}. A decent minimum had already been found during baseline training, the goal of fine tuning is to tweak the model's parameters such that temporal stability (and perhaps accuracy) can be improved. We therefore use smaller learning rates for fine tuning. For optimisation we use SGD with momentum.

The loss curves for this comparison are plotted in Figure 4.5. We chose learning rates of 1×10^{-5}, 1×10^{-6}, and 1×10^{-7}. The loss for both the training and validation sets are used here so that it is known when we are over fitting. An important caveat here is that this experiment took place during

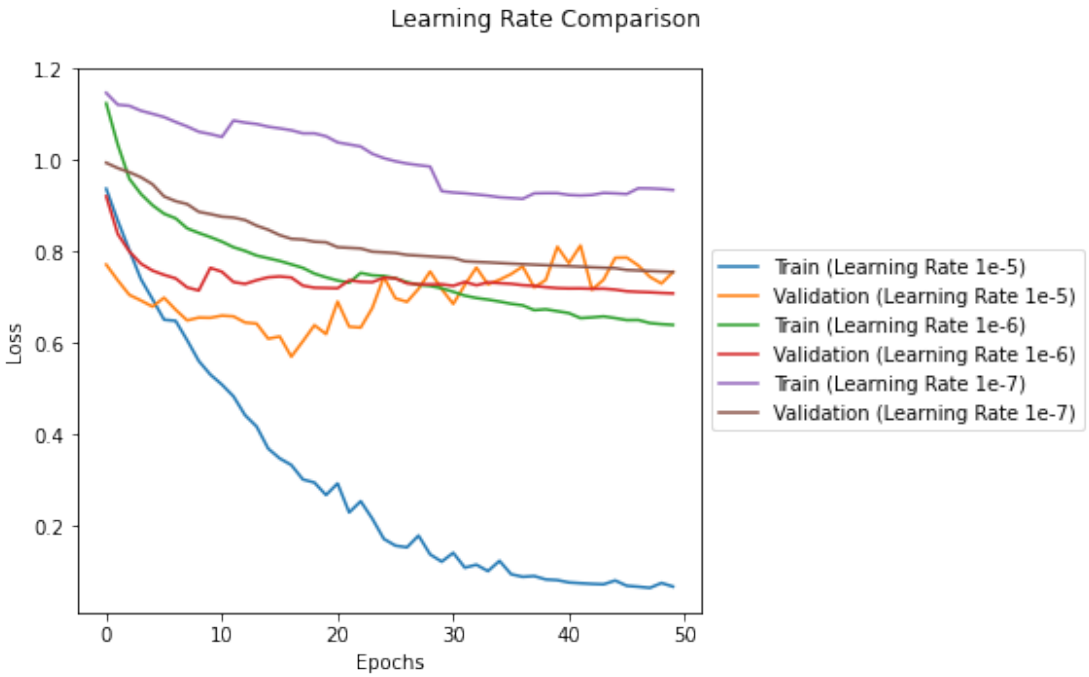

Figure 4.5: Comparison of learning rates used for fine tuning GSDNet4Video. Three learning rates are compared: 1×10^{-5}, 1×10^{-6}, and 1×10^{-7}. Both the training and validation loss are shown.

development, as a result the training process has been tweaked since it took place. We include this experiment because its results were used to determine several hyperparameters. The loss curves show a clear winner for having the learning rate set to 1×10^{-5}. For both the training and validation loss, a learning rate of 1×10^{-5} achieved the lowest minimum score. At epoch 16 we see the validation loss start to increase, this indicates over fitting. While the minimum for the training set may be around 0.05, this graph demonstrates that the model stops generalising many epochs prior, when the training loss is only around 0.35. Over fitting can also be observed when the learning rate is set to 1×10^{-6}, although it's effect is more subtle. Using a learning rate of 1×10^{-7} is impractical because it would take too many epochs to complete.

From this experiment we settle on a learning rate of 1×10^{-5} and train for 20 epochs. It is also interesting to note that learning rate effects the minimum validation loss. Validation curves for 1×10^{-5} and 1×10^{-6} both show over fitting, but a much lower minimum is ultimately found for 1×10^{-5}. To summarize, learning rate can have a significant impact on the effectiveness of fine tuning.

4.3.2 Augmentation

Throughout development we found that the augmentations applied in [27] (colour jittering, converting to grey scale, blurring, and mirroring in the vertical) helped our model to generalize. This additional layer of augmentation, which is not applied during baseline training, helps the fine tuned models achieve better general performance.

We demonstrate this in Figure 4.6. This plot was generated by running our fine tuning procedure on all pixels (no query strategy used). The experiment was run five times with augmentation and five times

without. Figure 4.6 shows the average validation loss for these five runs. The light shading behind the lines denote plus and minus one standard deviation for the graph on the left, and the interquartile range for the graph on the right. When evaluating the validation set, the augmentations are not applied. This is in contrast to Figure 4.7 where augmentation was applied to the validation set, hence why the validation loss values are slightly different.

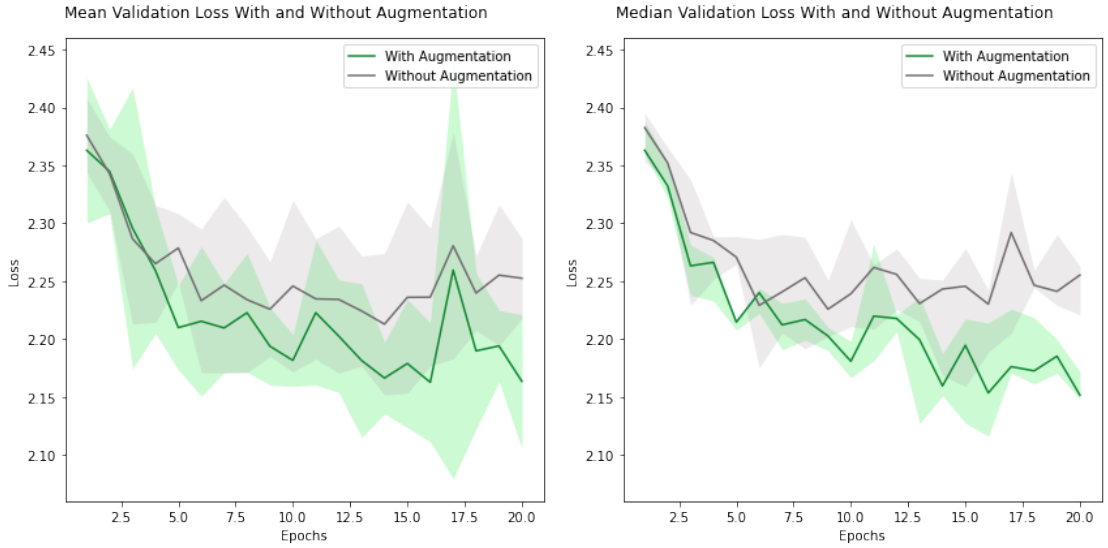

Figure 4.6: Validation loss for fine tuning using all pixels from synthetic videos. We perform this experiment ten times, five times with augmentation applied to the training data and five time without augmentation applied to the training data. The "With Augmentation" and "Without Augmentation" series show the averages of these runs. Due to large outliers sometimes appearing in training, both the mean and medians of the five runs are plotted. Mean loss is shown on the left and median loss is shown on the right. The lighter shading behind the lines show $\pm$ one standard deviation for the graph on the left, and the interquartile range for the graph on the right.

While this loss plot is noisy, a few clear patterns emerge. Mean validation loss when augmentation is used during training consistently ends up lower than when it is not used. Out of the two, training with augmentation also ends up with a better overall minimum. It seems also to be the case that augmentation results in more epochs being needed before a minimum is reached. One especially large outlier was record for one of the "With Augmentation" runs at epoch 17. It had a loss of 2.58 and has sharply spiked the mean validation loss. In order to exclude outliers like this, we have also included the median validation loss. The median validation loss shows the same tends as the mean validation loss.

From these observed trends we conclude that the augmentations described in [27] help our model to better generalize to unseen data. As a result, these augmentations have been applied to all of our fine tuning procedures.

4.3.3 Performance

To evaluate the performance of our query strategies we run five variations of the training procedure described in Section 3.2.4. To summarize this section, we fine tune GSDNet4Video on three frame synthetic videos with extra augmentation. We also run baseline training in parallel to prevent the model from forgetting what was learnt before fine tuning. The five variants are based on our flicker based query strategies, they are:

(1) All Pixels - No pixels are ignored during training. Entire binary masks are used for calculating the loss.

(2) Flickering Regions - Training is done on just pixels that flicker. All pixels that are not flickering are ignored and not used in calculating the loss. See Section 3.2.1 for how flickering regions are defined.

(3) Random Regions - Trains of random pixels within the video. The total number of random pixels will be the same as the number of flickering pixels in the video (≈ 2500 pixels per image).

(4) 5 Flickering Pixels - Training on just up to 5 pixels per image central to the flickering regions. See Section 3.2.2 for how flickering pixels are chosen.

(5) 5 Random Pixels - Train on five random pixels in each image. All other pixels are ignored and not used to calculate the loss.

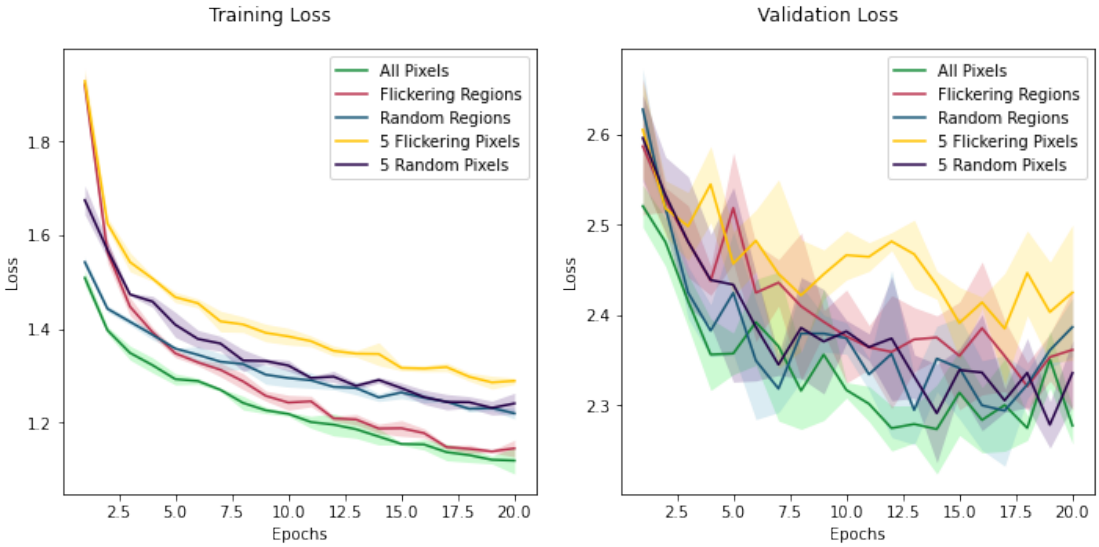

Figure 4.7: Training and validation loss for our five variants of fine tuning. (1) Fine tuning on all pixels (2) Fine tuning on flicking regions (3) Fine tuning on random pixels, where the number of pixels equals the total number of flickering pixels (4) Fine tuning on five flickering pixels (5) Fine tuning on five random pixels. The lines shown are the results of three training runs, all averaged together. The translucent areas surrounding each line show $\pm$ one standard deviation.

Since it is using the most data, the 'All Pixels' variant should give the best performance. Variants (2)-(5) all use significantly less data. Variants (2) and (3) both train on a similar number of pixels, the difference is that variant (2) trains on pixels that are found to be flickering between frames and variant (3) trains on random pixels. Since the amount of data used for these two variants is roughly the same, any difference in training loss is due to our query strategy. We can thus evaluate how effective training on flickering regions is. The same holds true for variants (4) and (5). Since they train on the same number of pixels, any difference in loss will be down to how we have chosen the pixels.

Figure 4.7 shows the training loss for these five variants when run for 20 epochs. As expected the best loss is achieved when all pixels are used. Initially, our query strategies (Flickering Regions and 5 Flickering Pixels) are outperformed by randomly selected pixels. However, after five epochs, training on flickering regions archives better loss than training on an equivalent number of randoms pixels. The same cannot be said for training on 5 flickering pixels. This method consistently underperforms training on 5 random pixels.

Why does training on flickering regions perform better than it's random counterpart, but training on 5 pixels does not? Figure 4.9 shows two images that were used during the fine tuning process. Highlighted in green are the areas detected as glass and highlighted in blue are flickering areas. These blue regions show the pixels that we have trained on for variant (2). It is generally true that flicking pixels exist on the edges of areas detected as glass. See on Figure 4.9 how blue regions typically border or surround green regions. Query strategies like margin sampling or entropy, if applied to this context, would also choose pixels on the edges of areas detected as glass. Furthermore, we know that these query strategies work better than randomly selected pixels [27]. It therefore seems plausible that the reason why flickering regions outperformed it's random counterpart, but 5 flickering pixels did not, is because flickering regions includes pixels on the edges of areas detected as glass, and thus contains pixels that would have been chosen by margin sampling or entropy query strategies.

All in all, fine tuning on flickering regions only uses 26.0% of all pixels while achieving close to the same performance. But this is only for the training loss. Figure 4.7 also shows validation loss. The validation loss for fine tuning on flickering areas consistently underperforms its random counterpart. Although this plot is noisy, it casts doubt on the ability of our flickering regions query strategy to perform well outside of the training data. Also surprising is the strong performance of training on 5 random pixels which achieves nearly the same validation loss as using all pixels.

A comparison of images before and after fine tuning is shown in Figure 4.8. Again, detected glass is highlighted in green, flickering areas in blue, and areas where both glass and flicker are detected in cyan. We have shown the output of three models. The first is the baseline GSDNet4Video model. The second and third columns use fine tuned models, where we have fine tuned using all pixels and using flickering regions respectively (variants (1) and (2) of our training procedure). We can see that both fine tuning variants shrink (but do not eliminate) the regions of flicker.

We can also verify this reduction in flicker numerically. During the fine tuning process we monitor how large the areas of flicker are for each image. Since we calculate the regions that are flickering,

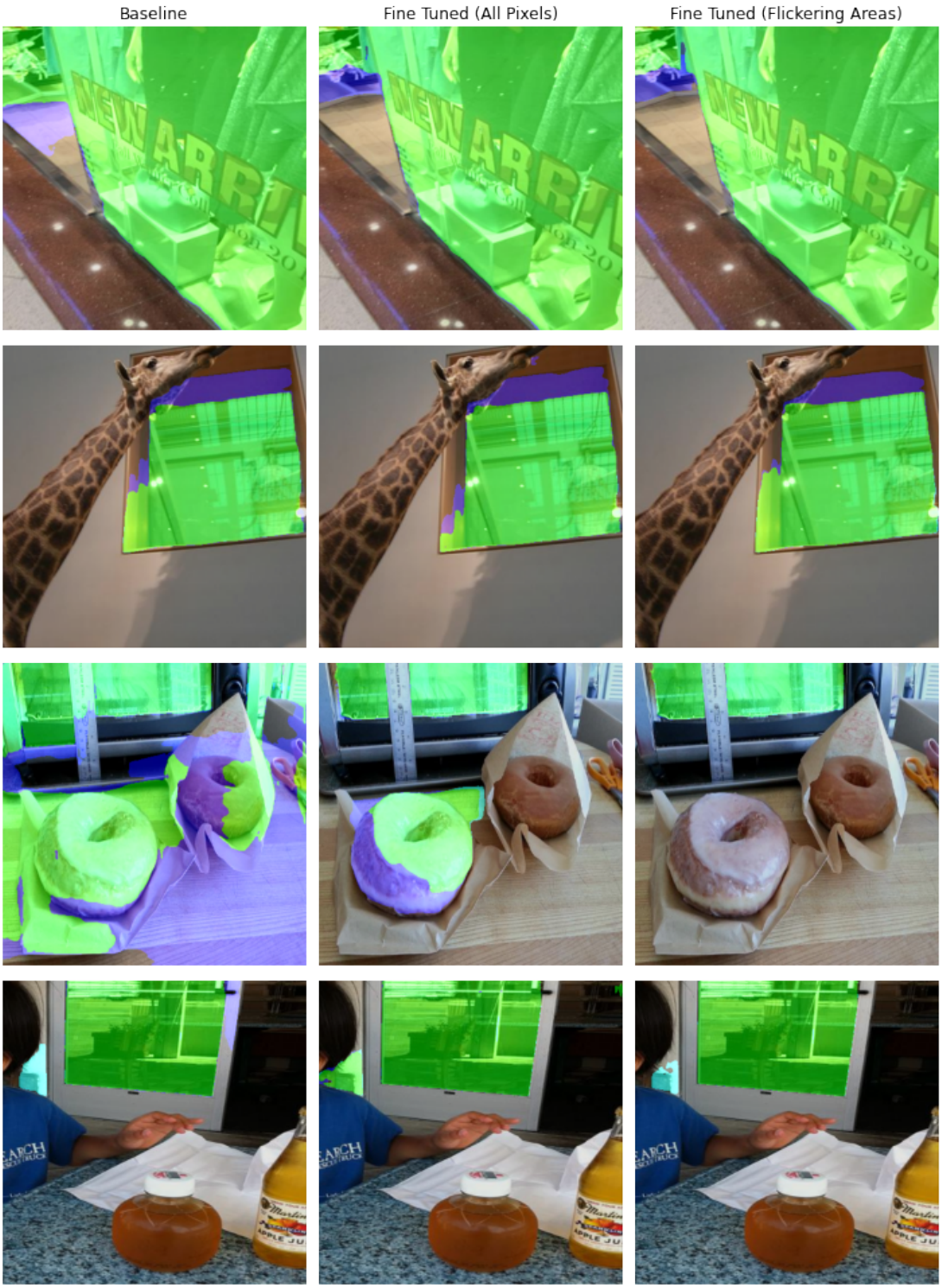

Figure 4.8: The three columns show the outputs of three models. The first is our baseline GSDNet4Video network. The second and thrid columns are fine tuned models, trained on all pixels and flickering regions respectively. Highlighted in green are the pixels that have been detected as glass. Highlighted in blue are pixels where flickering has been detected. Highlighted in cyan are pixels where both flicker and glass have been detected. In these examples, we see a reduction of the blue/cyan areas for our fine tuned models, meaning that flicker is reduced.

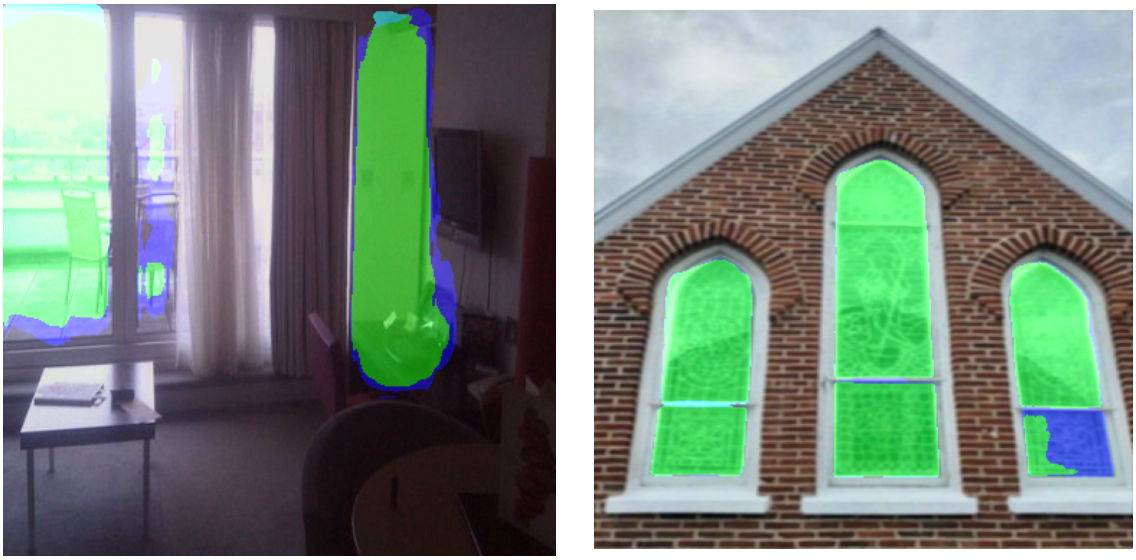

Figure 4.9: Images used during fine tuning. Highlighted in green are the pixels which have been classified as *glass*. Highlighted in blue are the pixels that are flickering, that is, pixels belonging to objects that have changed classification since the previous frame. Cyan areas are where both glass and flicker is detected.

determining the size of the flickering regions is simply a matter of counting the total number of flickering pixels. Figure 4.10 shows the total number of flickering pixels per epoch across all images in the training set, for variants (2) and (3) of fine tuning. Variant (2) trains on all flickering pixels, whereas variant (3) trains on the same number of pixels, but scattered randomly throughout the image. Both variants reduce flicker, however when we specifically train on flickering regions, the flicker is reduced by more. This shows that our query strategy of training on only the flicking regions, more effectively reduces flicker than training on an equivalent number of random pixels.

These results are of course derived from the synthetic videos detailed in Section 3.2.3. In the next section we will examine how these fine tuned models perform on real videos.

4.3.4 Analysis on Real Videos

For a side by side comparison of the baseline and a model fine tuned using all pixels see the video found at `https://youtu.be/ZS90ZS_6w4M`. We also include a comparison of the baseline and a model that has been fine tuned by training only on flickering regions. A video comparison can be found at `https://youtu.be/UCVzW0wkBcQ`. In Section 4.3.3 these are variants (1) and (2).

We observe no noticeable change in temporal stability between the baseline and fine tuned models. This does not necessarily mean that there is no difference, just that if improvements or degradations exist, they are not perceptible. For completeness we have included Table 4.4, which shows our qualitative analysis of performance in terms of accuracy and temporal stability. If all models perform roughly the same, an equals sign is used. Where one model outperforms the other two, a tick is used. Pixel accuracy for the 100 random pixels, distributed throughout the first 10 frames of each video, is also included. The usual caveats regarding these measures still apply.

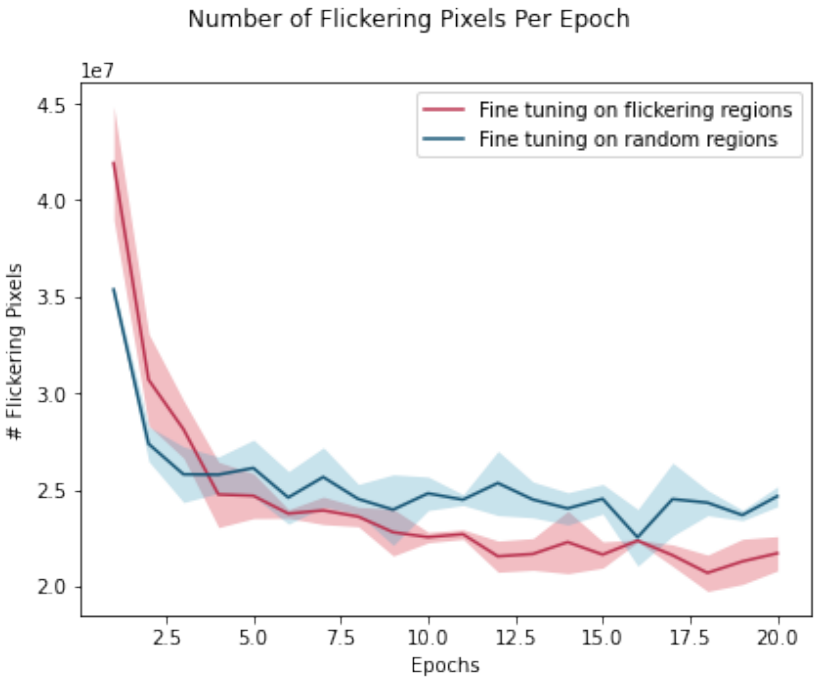

Figure 4.10: This graph quantifies the amount of flickering that exists within the training set across all 20 training epochs. Variants (2) and (3) of fine tuning are shown here. The lines show the averaged result of three training runs. The lighter coloured areas behind the lines show $\pm$ one standard deviation.

Compare Table 4.4 to Table 4.3. Table 4.4 contains many more equals signs. This is not surprising since the fine tuning process by definition, only makes small changes to the model. The difference in temporal stability is especially hard to gauge here. As a result, the temporal stability of all three models are evaluated to be roughly the same. The majority of videos also appear to be roughly the same in terms of accuracy. The exceptions are videos 5, 7, and 13, which are more accurate when run on the baseline model, and videos 4, 8, and 14, which are more accurate when run on the model fine tuned using all pixels. For the model that has been fine tuned using flickering regions, videos 4, 8, and 14 are more accurate than the baseline, but not the model fine tuned using all pixels. This conforms to our intuition that fine tuning using all data should yield better performance than fine tuning with only part of the data. The average pixel accuracy for the baseline, fine tuned (with all pixels), and fine tuned (with flickering regions) models are 0.875, 0.859, and 0.871 respectively. This indicates a slight degradation in accuracy for the fine tuned models. Taken together, the pixel accuracy and qualitative evaluation indicates that any difference in accuracy between these models is small.

Figure 4.11 shows a sequence of frames taken from video clip 14. The output of all three models is shown. This is an example of where both fine tuning approaches have improved accuracy. In this example fine tuning using flickering regions gets a more accurate result than fine tuning on all pixels, however this is not typically the case. Notice also that for all three models the shape of the detected glass (the green highlighted area) stays quite constant over the course of the five frames.

Video Number	Baseline			Fine Tuned (All Pixels)			Fine Tuned (Flickering Regions)		
	PA	**Acc.**	**TS**	**PA**	**Acc.**	**TS**	**PA**	**Acc.**	**TS**
1	0.98	=	=	0.98	=	=	0.98	=	=
2	0.72	=	=	0.81	=	=	0.98	=	=
3	0.98	=	=	0.98	=	=	0.97	=	=
4	0.85		=	0.83	✓	=	0.81		=
5	0.99	✓	=	1.0		=	1.0		=
6	0.81	=	=	0.85	=	=	0.83	=	=
7	0.91	✓	=	0.78		=	0.71		=
8	0.78		=	0.6	✓	=	0.42		=
9	0.8	=	=	0.72	=	=	0.8	=	=
10	0.87	=	=	0.98	=	=	0.95	=	=
11	0.9	=	=	0.95	=	=	0.94	=	=
12	0.99	=	=	0.98	=	=	0.99	=	=
13	0.95	✓	=	0.96		=	0.98		=
14	0.88		=	0.89	✓	=	0.89		=
15	0.98	=	=	0.81	=	=	0.84	=	=
16	0.99	=	=	1.0	=	=	1.0	=	=
17	0.88	=	=	0.89	=	=	0.89	=	=
18	0.91	=	=	0.87	=	=	0.84	=	=
19	0.39	=	=	0.35	=	=	0.79	=	=
20	0.93	=	=	0.95	=	=	0.95	=	=

Table 4.4: Accuracy (Acc.) and temporal stability (TS) are both evaluated qualitatively for 20 videos. Pixel Accuracy (PA) for 100 random pixels in the first 10 frames of each video is also shown for each model. Three models are compared here: the baseline GSDNet4Video model, a version of GSDNet4Video fine tuned on all pixels, and a version of GSDNet4Video fine tuned on all flickering pixels (our flickering regions query strategy).

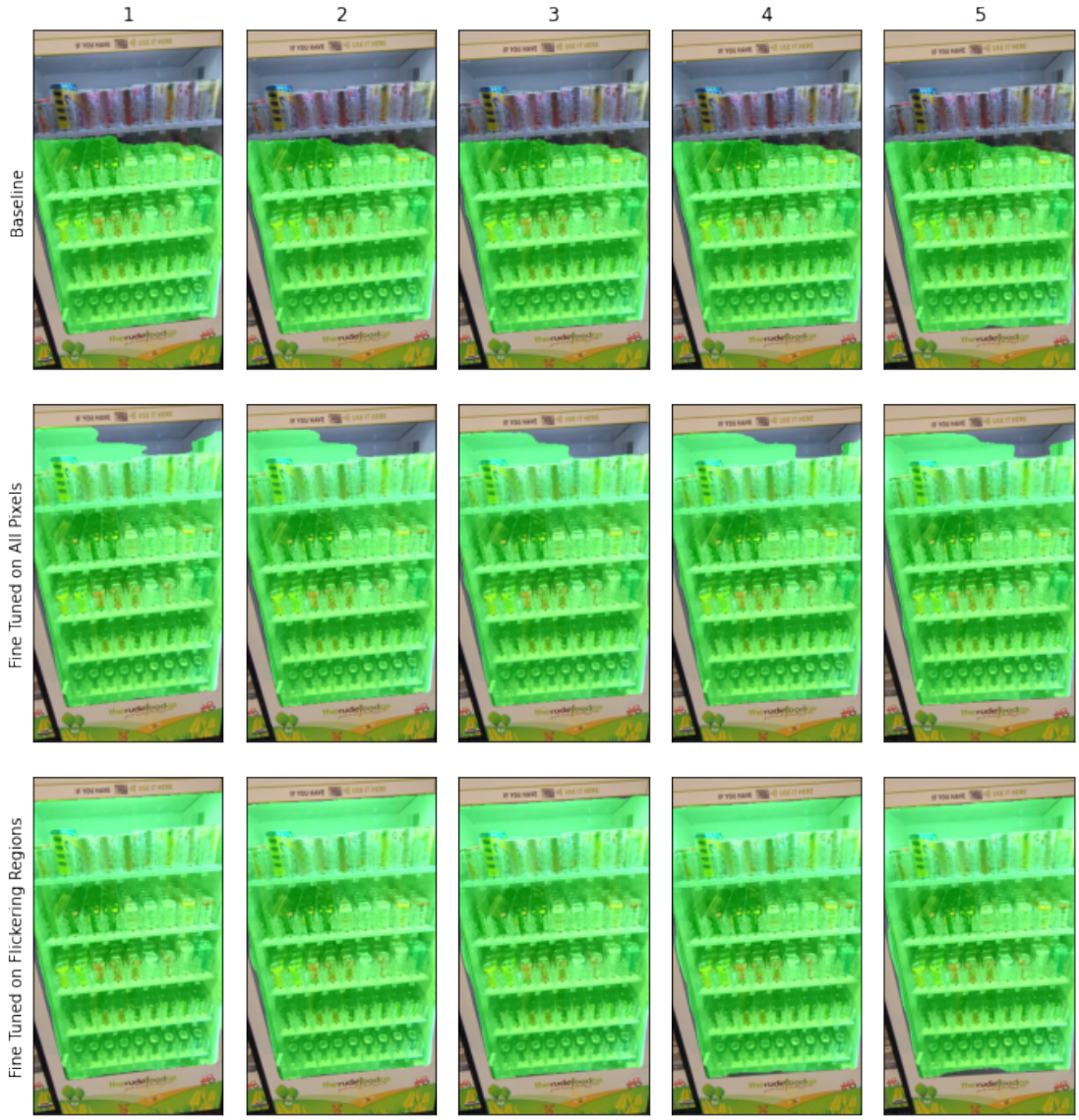

Figure 4.11: Five consecutive frames taken from video 14. Glass detection for the three models is shown in green. The top row shows the output of the GSDNet4Video baseline model. The middle row shows the output of GSDNet4Video after it has been fine tuned on all pixels. The bottom row shows the output of a variant that has been fine tuned on only flickering regions.

Chapter 5

Conclusion

In this chapter we provide a brief summary of our methods and results. We also suggest several future directions this work could be taken in and how the field of glass detection in video could be developed more generally.

5.1 Present Work

In this project we present GSDNet4Video, the first CNN specifically designed for glass detection in RGB videos. This is done by adding a fourth input channel containing a previous frame's mask to GSDNet [15]. If trained correctly, we can then build a dependency between the results of a previous frame and the current frame, thus increasing temporal stability. We demonstrate through our qualitative study that GSDNet4Video has superior temporal stability than GSDNet. Of the 20 videos evaluated, all 20 perform as well as, or better than, GSDNet for temporal stability. Three measures of accuracy were used: pixel accuracy for 100 random pixels, accuracy on single images, and accuracy evaluated qualitatively on real videos. All three measures have their weaknesses, but they all point to the same conclusion. The accuracy of GSDNet is better than GSDNet4Video. However, this decrease is accuracy is only small. Furthermore, the original ResNeXt-101 backbone has been replaced with ResNet-18, reducing inference time by over 50%.

Building upon the GSDNet4Video baseline we then attempt to fine tune the network using active learning. Specifically this is done using novel query strategies based on flickering between consecutive frames. Through augmenting images in the Glass Surface Dataset (GSD), we are able to create short, fully labelled, synthetic videos. These are then used for fine tuning the baseline. We compare the quality of the fine tuning when all pixels in the videos are used for training, flickering regions in the video are used for training, and only five flickering pixels in the video are used for training. As expected, training on all pixels produces the best results. Training on only five flickering pixels produces the worst results, underperforming fine tuning done on five randomly selected pixels. This indicates that our scheme for choosing these five pixels is not an effective query strategy. Fine tuning on all flickering pixels produces some promising results, but more investigation is required before its efficacy as a query strategy can be determined. In terms of training loss, training on all flickering regions outperforms training on an equivalent number of random pixels. Using only 26.0% of the data, it reaches nearly the same loss as

training on all pixels. However, this result is not replicated for the validation loss, where training on all flickering pixels underperforms training on an equivalent number of random pixels. This suggest that while this query strategy performs well on training data, it fails to generalize. Furthermore, our validation plots are especially noisy, so the inconsistency between training and validation loss may be a matter of finding more optimal hyperparameters. We have shown that learning rate can make a significant impact on the how effective training can be. During the fine tuning procedure, we can see that flicker is indeed reduced for synthetic videos. However, whether flicker is reduced for real videos is an open question. Qualitative analysis shows that any differences between the baseline and fine tuned models are minor.

While the creation of the baseline GSDNet4Video successfully achieved its goal of improving temporal stability, questions still remain around how effective a query strategy based on flicker can be. In the next section we propose the next steps that should be taken for these questions to be answered.

5.2 Future Work

As we have demonstrated, video models can be built without a fully labelled video dataset. That being said, a dataset dedicated specifically to glass detection in videos would be of immense use to the field. This could contain masks for every frame or just masks for a few frames. If these models are to be deployed in navigation systems, speed is also another area that needs to be improved upon. Replacing the backbone has significantly reduced inference time, but the network still doesn't run in real time. We recommend the techniques described in [26] and [32] which are used to speed up segmentation in videos. The performance of our baseline models could also be improved simply by taking more regular checkpoints. Due to time and memory constraints checkpoints are currently saved every 50 epochs. However, creating them more regularly would likely yield us a model with a better overall minimum.

As pointed out in Section 2.2, GSDNet provides unique opportunities for video due to it's consideration of reflections. It's training relies on accurate reflection removal. Therefore any improvement in reflection removal may in turn improve the performance of GSDNet. One way to improve reflection removal is to use multiple images [31]. Better performance could potentially be achieved by leveraging the fact that videos contain multiple images and thus achieve better reflection removal.

A key practical consideration for anyone looking to build on our code base is training time. Training our baseline for 400 epochs takes over four days, which is a major bottleneck when developing these models. When profiling our code, we found that long training times were mostly due to the data augmentation. Removing this bottleneck should be a priority for anyone trying to build on top of our code base.

Training on flickering regions and training on five flickering pixels exhibit very different behaviours. The former outperforms it random counterpart but the latter does not (on training loss, not on validation loss). Our method for training on five pixels chooses five pixels central to the flickering regions. It may be the case that central pixels of these flickering regions are less informative than edge pixels. As discussed in Section 4.3.3, flickering regions tend to border on regions classified as glass. This means that many of the pixels chosen by our flickering regions query strategy, are also pixels that would have been chosen by an entropy or margin sampling query strategy. In order to determine the usefulness of the

flickering regions query strategy, we suggest fine tuning on only pixels with high entropy and comparing results. Another experiment could train on only pixels at the edges of the flickering regions. The key question that needs to be answered here is this; was the training on flickering regions only successful because it happened to include pixels with high entropy?

Of course, training on flickering regions was not entirely successful. While fine tuning on flickering regions outperformed its random counterpart on training loss, it did not for validation loss. To explore this discrepancy further, metrics other than loss should be applied to the validation set. If the validation sets average intersection of union and amount of flicker were tracked over the course of the fine tuning, we would get a more complete picture. The validation loss suggests that fine tuning on flickering regions does not generalize well, however, these results are noisy. Rerunning the fine tuning process many more times would reduce the noise in our validation plot. Noise for our validation loss may also be due to suboptimal hyperparameters. A lower or decaying learning rate should be explored to ensure that the best performance can be achieved. Hopefully once this is done, we will be able to conclusively state whether the flickering regions query strategy does or does not result in good general performance.

Finally, we suggest a hybrid approach. Training on both flickering regions and random pixels could potentially maintain good performance on the training loss, while simultaneously improving the performance on the validation loss.

Bibliography

[1] S. Abu-El-Haija, N. Kothari, J. Lee, P. Natsev, G. Toderici, B. Varadarajan, and S. Vijayanarasimhan. Youtube-8m: A large-scale video classification benchmark. *arXiv preprint arXiv:1609.08675*, 2016.

[2] V. Bazarevsky and A. Tkachenka. Mobile real-time video segmentation, 2018.

[3] W. H. Beluch, T. Genewein, A. Nürnberger, and J. M. Köhler. The power of ensembles for active learning in image classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 9368–9377, 2018.

[4] M. Berman, A. R. Triki, and M. B. Blaschko. The lovász-softmax loss: A tractable surrogate for the optimization of the intersection-over-union measure in neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[5] G. Brostow. Lecture notes on image segmentation. `http://www0.cs.ucl.ac.uk/staff/G.Brostow/classes/IP2008/L1_Segmentation_02.pdf`, October 2008.

[6] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE transactions on pattern analysis and machine intelligence*, 40(4):834–848, 2017.

[7] H. K. Cheng, Y.-W. Tai, and C.-K. Tang. Modular interactive video object segmentation: Interaction-to-mask, propagation and difference-aware fusion. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5559–5568, 2021.

[8] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[9] A. Fathi, M. F. Balcan, X. Ren, and J. M. Rehg. Combining self training and active learning for video segmentation. Georgia Institute of Technology, 2011.

[10] J. Fu, J. Liu, H. Tian, Y. Li, Y. Bao, Z. Fang, and H. Lu. Dual attention network for scene segmentation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 3146–3154, 2019.

[11] U. Gaur, M. Kourakis, E. Newman-Smith, W. Smith, and B. Manjunath. Membrane segmentation via active learning with deep networks. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 1943–1947. IEEE, 2016.

[12] Y.-C. Guo, D. Kang, L. Bao, Y. He, and S.-H. Zhang. Nerfren: Neural radiance fields with reflections. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 18409–18418, 2022.

[13] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[14] Z. Huang, K. Wang, K. Yang, R. Cheng, and J. Bai. Glass detection and recognition based on the fusion of ultrasonic sensor and rgb-d sensor for the visually impaired. In *Target and Background Signatures IV*, volume 10794, pages 118–125. SPIE, 2018.

[15] J. Lin, Z. He, and R. W. Lau. Rich context aggregation with reflection prior for glass surface detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13415–13424, 2021.

[16] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[17] H. Mei, X. Yang, Y. Wang, Y. Liu, S. He, Q. Zhang, X. Wei, and R. W. Lau. Don't hit me! glass detection in real-world scenes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[18] G.-T. Michailidis and R. Pajarola. Bayesian graph-cut optimization for wall surfaces reconstruction in indoor environments. *The Visual Computer*, 33(10):1347–1355, 2017.

[19] S. Minaee, Y. Boykov, F. Porikli, A. Plaza, N. Kehtarnavaz, and D. Terzopoulos. Image segmentation using deep learning: A survey. *IEEE Transactions on Pattern Analysis Machine Intelligence*, 44(07):3523–3542, jul 2022.

[20] A. Pasad, A. Gordon, T.-Y. Lin, and A. Angelova. Improving semantic segmentation through spatio-temporal consistency learned from videos. *arXiv preprint arXiv:2004.05324*, 2020.

[21] M. Paul, M. Danelljan, L. Van Gool, and R. Timofte. Local memory attention for fast video semantic segmentation. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1102–1109. IEEE, 2021.

[22] F. Perazzi, A. Khoreva, R. Benenson, B. Schiele, and A. Sorkine-Hornung. Learning video object segmentation from static images. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3491–3500, 2017.

[23] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, B. B. Gupta, X. Chen, and X. Wang. A survey of deep active learning. *ACM computing surveys (CSUR)*, 54(9):1–40, 2021.

[24] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[25] B. Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[26] E. Shelhamer, K. Rakelly, J. Hoffman, and T. Darrell. Clockwork convnets for video semantic segmentation. In *European Conference on Computer Vision*, pages 852–868. Springer, 2016.

[27] G. Shin, W. Xie, and S. Albanie. All you need are a few pixels: semantic segmentation with pixelpick. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1687–1697, 2021.

[28] Z. Teed and J. Deng. Raft: Recurrent all-pairs field transforms for optical flow. In *European conference on computer vision*, pages 402–419. Springer, 2020.

[29] Y. Tian, G. Cheng, J. Gelernter, S. Yu, C. Song, and B. Yang. Joint temporal context exploitation and active learning for video segmentation. *Pattern Recognition*, 100:107158, 2020.

[30] A. Vezhnevets, J. M. Buhmann, and V. Ferrari. Active learning for semantic segmentation with expected change. In *2012 IEEE conference on computer vision and pattern recognition*, pages 3162–3169. IEEE, 2012.

[31] R. Wan, B. Shi, L.-Y. Duan, A.-H. Tan, and A. C. Kot. Benchmarking single-image reflection removal algorithms. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3922–3930, 2017.

[32] H. Wang, X. Jiang, H. Ren, Y. Hu, and S. Bai. Swiftnet: Real-time video object segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1296–1305, June 2021.

[33] W. Wang, T. Zhou, F. Porikli, D. Crandall, and L. Van Gool. A survey on deep learning technique for video segmentation. *arXiv preprint arXiv:2107.01153*, 2021.

[34] X. Wang and J. Wang. Detecting glass in simultaneous localisation and mapping. *Robotics and Autonomous Systems*, 88:97–103, 2017.

[35] H. Wei, X.-e. Li, Y. Shi, B. You, and Y. Xu. Multi-sensor fusion glass detection for robot navigation and mapping. In *2018 WRC Symposium on Advanced Robotics and Automation (WRC SARA)*, pages 184–188. IEEE, 2018.

[36] S. Woo, J. Park, J.-Y. Lee, and I. S. Kweon. Cbam: Convolutional block attention module. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19, 2018.

[37] E. Xie, W. Wang, W. Wang, P. Sun, H. Xu, D. Liang, and P. Luo. Segmenting transparent object in the wild with transformer. *arXiv preprint arXiv:2101.08461*, 2021.

[38] X. Yang, H. Mei, K. Xu, X. Wei, B. Yin, and R. W. Lau. Where is my mirror? In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 8809–8818, 2019.

[39] D. Yoo and I. S. Kweon. Learning loss for active learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 93–102, 2019.

[40] L. Yu, H. Mei, W. Dong, Z. Wei, L. Zhu, Y. Wang, and X. Yang. Progressive glass segmentation. *IEEE Transactions on Image Processing*, 31:2920–2933, 2022.

[41] X. Zhang, R. Ng, and Q. Chen. Single image reflection separation with perceptual losses. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4786–4794, 2018.