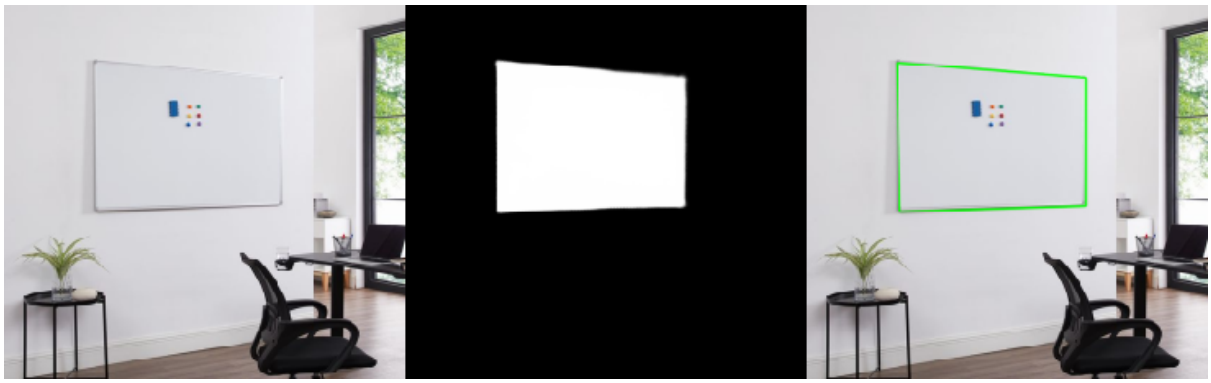
Using machine learning to improve the whiteboard experience

Bachelor Project, BIBAPRO1PE
IT University of Copenhagen, Spring 2022



Balder Sandstrøm — basa@itu.dk

May 16, 2022

Abstract

Virtual meetings and conferences are getting more common and more mainstream in the workplace. This shift in use of technology means that other work practices has to adapt to work with virtual meetings. One such practice is the use of writing on whiteboards. Just like some people prefer to read from a book instead of a monitor, a practice like writing on a whiteboard might never be replaced by writing on a tablet. This creates a set of problems of how can the whiteboard be integrated to work with the virtual world. There's many ideas and potential solutions for this like using text recognition, but most of these solutions require the whiteboard to be detected in the first place. To solve this issue, a whiteboard detection model is proposed which is composed of a convolutional neural net to classify whiteboards in real-time videos through semantic image segmentation and computer vision to process the outline of the classified whiteboards into a set of points which can be used for further analysis and processing.

Contents

1 Introduction

1.1 Background

These days virtual meetings are becoming more common and are often combined with physical meetings connected virtually through video conference software. This raises many new technological problems such as writing physically on a whiteboard and sharing it with the video conference. This project will focus on how to identify the outline of whiteboards to allow for further image analysis of the whiteboard. While being able to zoom in and show a whiteboard as a separate video in a video conference is quite useful in itself, the technology can also be useful for other use cases. Examples of other use cases which would require further processing and analysis could be to use text recognition to highlight and sharpen the written content on the whiteboard, making sharing it virtually more clear. It could also be used for hiding people walking in front of the whiteboard, making the content always visible to the video conference. Though, for this project no further analysis and processing will be done after detecting the whiteboard - that is left for future research.

To identify the outline of a whiteboard in a real-time video, a combination of machine learning and computer vision will be used to draw and identify the outline of the whiteboard. This will be done by finding and labeling image data to train a convolutional neural network learning, and using computer vision as post-processing to create the final outline of the whiteboard.

1.2 Problem description

How can machine learning and computer vision be used to detect whiteboards in real-time videos to improve the whiteboard experience in virtual meetings and conferences?

1.3 Requirements and constraints

A few requirements are set for the project:

- The model must be able to run at roughly 30 frames per second on a medium to high end graphics card.
- The model must be able to accurately detect the outline of a whiteboard on some test data which it hasn't seen before during training.

This project is done under specific constraints which limits the scope of the project:

- Everything done during the project is made on a medium to high end computer. This means that only limited testing and experimentation is done, as training the neural network takes between 8 to 16 hours every time.
- The dataset used to train the model is manually created from publicly available sources which limits the scope of what the model can learn during training.

2 Model

The whiteboard detection model is composed of two parts: The first part consists of a convolutional neural network, and the second part consists of post-processing with computer vision. The convolutional neural net's task is to do image segmentation on the image to classify which pixels in the image that are whiteboard and which are not. The post-processing's task is to convert this binary classification image to a set of lines representing the outline of the whiteboard.

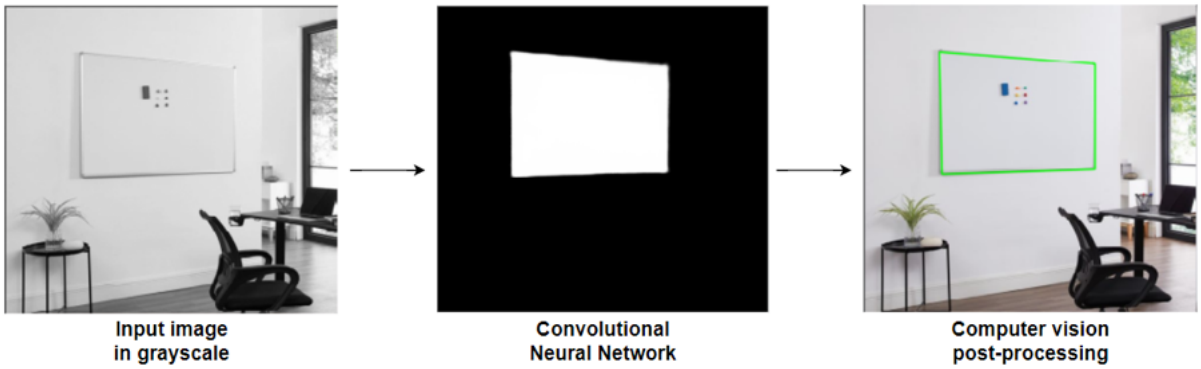


Figure 1: *Image of how the model converts a raw image to a binary classification image, and then to the outline of a whiteboard*

2.1 Convolutional Neural Network

Choosing a neural network to fit the task of doing image segmentation in real-time to detect whiteboards comes down to a few decisions.

It must be very fast in order to do classification of 30 images per second. It also has to be very precise, almost precise to the pixel, in order to create an accurate outline of the whiteboard. At last, it must be trainable with a small dataset and respond well to data augmentation. For these reasons, an Attention U-Net model is chosen which is very close to an original U-Net.

The original U-Net[4] proposes a fully convolutional architecture consisting of an contracting path and an expanding path. The contracting path captures context and features of the image by downscaling the image while finding increasingly more features with convolutional layers. The expanding path upscales the image again while finding the precise location of these features. This is done by concatenating the output each contracting layer with the each expanding layer.

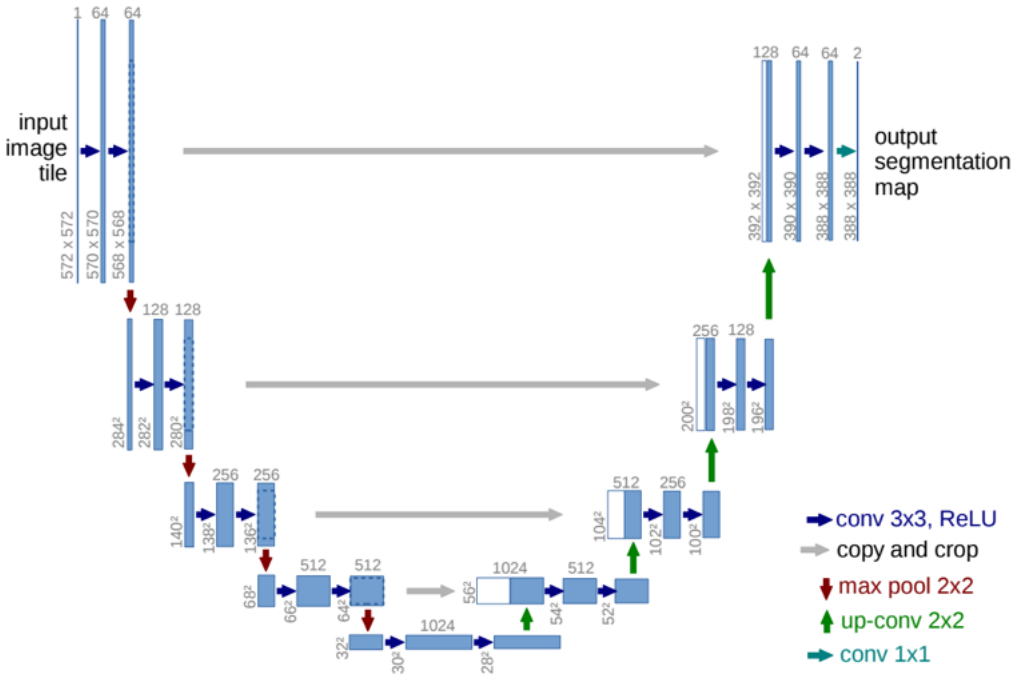


Figure 2: *Figure from U-Net paper[4] of the proposed model architecture*

The Attention U-Net[3] introduces the concept of an attention gate which is used for each expanding layer in the network. This attention gate helps the model learn to focus on target structures and suppress irrelevant regions of the input image. This is quite useful for the whiteboard model, as the whiteboard is always localized in a single area of interest in the image. Using this soft attention will thus help suppressing any other noise around the whiteboard resulting in a better and cleaner model. The additional attention gates are also not very computational heavy, and can therefore be used for this model without sacrificing performance.

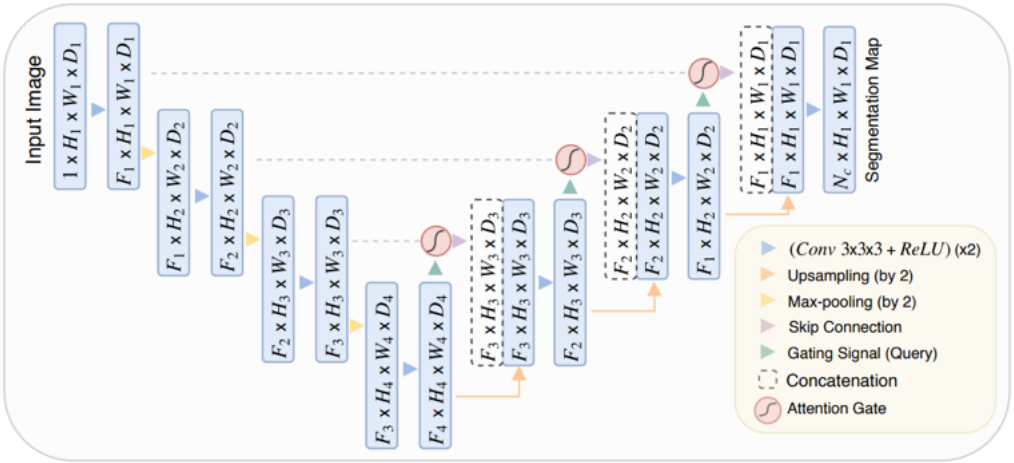


Figure 3: *Figure from Attention U-Net paper[3] of the proposed model architecture*

The specific topology of the Attention U-Net used for the project has an input layer

of size 256x256x1. It only uses a single color channel, because it's a lot easier to detect edges in grayscale format. Since the most important features the model has to look for in the image are the edges of the whiteboard, it makes using grayscale an obvious choice. It also removes 2/3 of the data compared to an RGB image, making the model and training the model faster. The output layer is also 256x256x1 since it's doing binary classification, and the output image should be the same size as the input image to find the exact location of the whiteboard.

The amount of features used in the neural net are 16, 32, 64, 128, 256, and 512 which are quite a bit smaller than the proposed sizes of 64, 128, 256, 512, and 1028 from the original U-Net[4]. But they also used an input size of 572x572 which almost has 5 times as many pixels. An additional sixth layer is also added to the model. This is done to make the first amount features found in the contracting path be 16, and the last be 512. 16 seems to be good amount of features due to whiteboards being quite simple in shape, and 512 features are still needed to have enough parameters in the model for it to be able to learn enough. Adding anymore than 512 features in the last layer, results in the model running too slow. This amount of features in the neural net results in the model having a total of 7.851.672 parameters.

For the expanding path, transposed convolutions are used instead of bilinear or nearest upsampling. Using transposed convolutions adds additional trainable parameters for the expanding path which is quite useful since the model needs to learn how to fill in missing areas of the whiteboard.

At last, the output of the model is sent through a sigmoid activation layer to get an output image of probabilities between 0 and 1 for each pixel, where 1 is whiteboard and 0 is non-whiteboard.

2.2 Post-processing

To convert the output from the convolution neural network to a set of lines representing the outline of the whiteboard, a series of post-processing algorithms are used. All algorithms used are provided by OpenCV and are not developed or implemented from scratch, and will therefore not be described in details for this project.

The first algorithm used is contour finding, or border following. This algorithms approximate the boundaries of shapes that has roughly the same color. This results in a list of contours, each consisting of a sequence of points. Since the convolutional neural net doesn't always predict a single shape, and multiple contours might be found, the contour with the largest area is picked and the rest is discarded.

Hereafter, the convex hull algorithm is used on the contour to remove any concave sections. The convolution neural net does not always predicts shapes without concavity in certain areas. This can happen when someone or something is in front of the whiteboard, creating an area of uncertainty. After using convex hull the outline of the whiteboard should now be visually quite accurate, but it still consists of more points than needed.

Therefore, the next step is trying to approximate as few lines as possible to fit around the outline. For this, the Ramer-Douglas-Peucker algorithm is used to decimate the points into only four points representing the corners of the whiteboard (sometimes 5 and 6 points depending on the image).

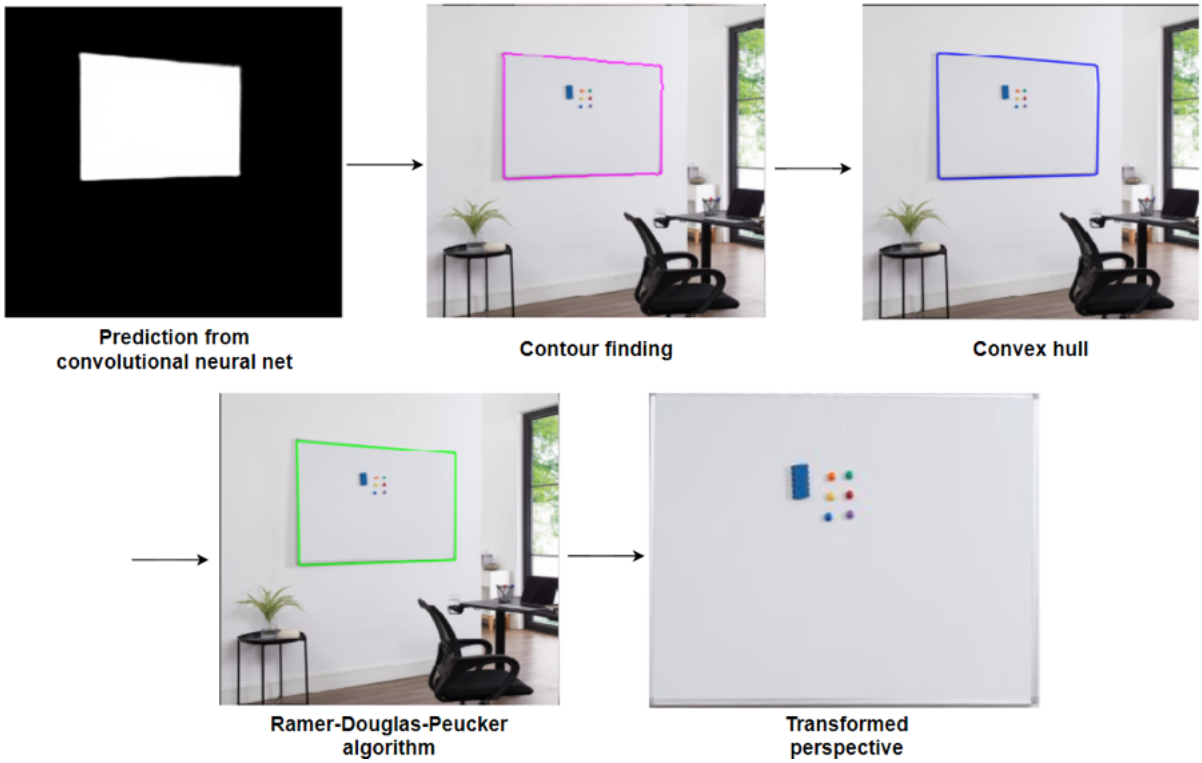


Figure 4: *Image of the algorithms used and its output during each post-processing step*

Now, the corners of the whiteboard are found, and can be used as an accurate area of interest for further processing. For this project, the only further processing done is transforming the perspective of the whiteboard to fit the screen. This does however only work when the outline consists of four points, and will result in undefined behavior if more corners are detected.

3 Data

3.1 Intro

The dataset is one of the most critical parts of training an image segmentation model. Preferably, the data should represent what the model will be used for in practise as close as possible. Since the whiteboard model will be used mainly for professional use in home offices, conference rooms, and offices, it would be ideal to gather as many videos and images in this context from the perspective of a webcam. Unfortunately, this was not really possible to gather during this project as this data is not widely available. This is because images of videos of this kind are kept private, confidential or just not shared with the public. Additionally, no public datasets of different whiteboards in varying environments exist already, meaning that all data had to be gathered manually from publicly available sources. The sources used are random images from Google of whiteboards combined with educational images taken from educational videos on Youtube. As a result of using a lot of educational videos which are often recorded in a school environment, the segmentation model is better at segmenting whiteboards in this environment compared to an office environment. This does not invalidate the research of this project, as the dataset could easily be extended in the future to improve its practical use.

3.2 Data selection

In this section it's described the reasoning of factors and features that has been considered for selecting the data for the dataset. The dataset is created of a combination of 38 still images and 29 videos, totaling a data pool of 67 different sources.

All the videos have someone actively drawing on the whiteboard, and sometimes have additional people in the foreground. The still images consist mostly of whiteboards without any people. This balance of picking images with and without people in the foreground, should help the model learn what an actual whiteboard looks like, and at the same time be able to ignore people covering parts of the whiteboard.

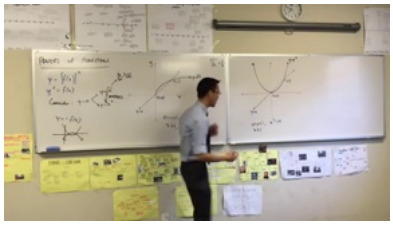
The edges of whiteboards differs greatly from whiteboard to whiteboard. Some have a very clear distinct frame surrounding the drawing surface of the whiteboard, and some whiteboards are edgeless, where the surface of the whiteboard is also the actual edge. Therefore a variety of whiteboards with different edge features are used in the dataset.

While not the most common outside of school environments, multiple whiteboards can be joined together to create a single larger whiteboard. This creates a fake joined edge in the middle of the two whiteboards which needs to be ignored by the model. The result of including these cases might confuse the model about where the actual edges are, but has been included none the less as it's a desired feature to be able to detect, and not having it in the training data will make the model too uncertain about which of the whiteboards to detect.

A combination of clean whiteboards without any content on it, and whiteboards filled with content like drawings, posters and sticky notes is chosen for the dataset. This teaches the model that whiteboards does not always have a clean white surface and can be covered with a lot different things, and it should therefore be able to ignore this.

Most of the dataset consists of images where the whole whiteboard fits in the image, and does not have parts which extends beyond the image frame. Though, some of the data does have whiteboards which cannot fit in the images. This is important to include as it cannot be expected that the camera capturing the whiteboard can be place such that the whole whiteboard is visible. This is a fine balance between teaching the model to always look for four edges and four corners which resembles the quadrilateral surrounding the whiteboard, and at the same time teaching the model to be receptive of missing corners and edges. This also means that the model will not always detect a quadrilateral, but sometimes also 5 and 6 sided polygons when corners are missing.

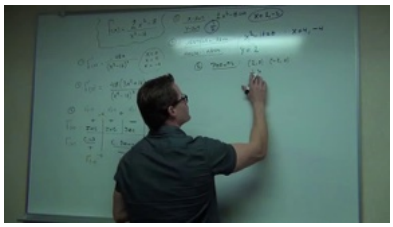
At last, a combination of different environments, surroundings, and lighting is chosen for the dataset. The model does not just learn about what a whiteboard is, it also has to learn about what a whiteboard is not. Therefore it's important to have data which is not just placed on a blank wall, but also includes other objects as doors, windows, monitors, etc. in the dataset. Lighting can also change the look of a whiteboard tremendously. Not only can the color of the surface change from white, to beige, blue, and yellow, it can also change the reflections in the whiteboard. Some whiteboards are very reflective, and having the sun or any other direct light shine on it can create all kinds of weird artifacts, which the model needs to learn to ignore.



(a) *Two large whiteboards joined together*

(b) *Whiteboard with a fake edge*

(c) *Whiteboard filled with content like sticky notes*

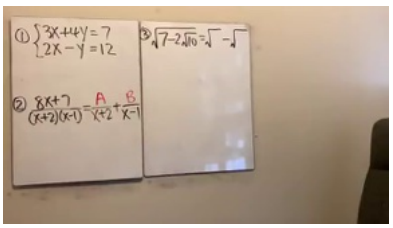


(d) *Large whiteboard that covers most of the frame and has a non-quadrilateral outline*

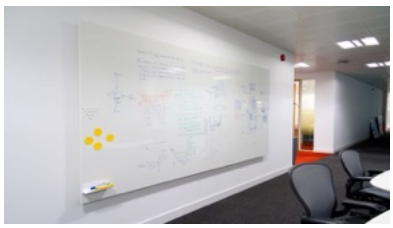
(e) *Whiteboard with a missing edge and a white monitor screen*

(f) *Small whiteboard with no distinct frame surrounding it*

(g) *Whiteboard with its bottom half cutout out of the frame*



(h) *Two small whiteboards joined together*



(i) *Large whiteboard viewed from non-facing perspective*

Figure 5: *Examples of whiteboard images used in the dataset*

3.3 Dataset creation

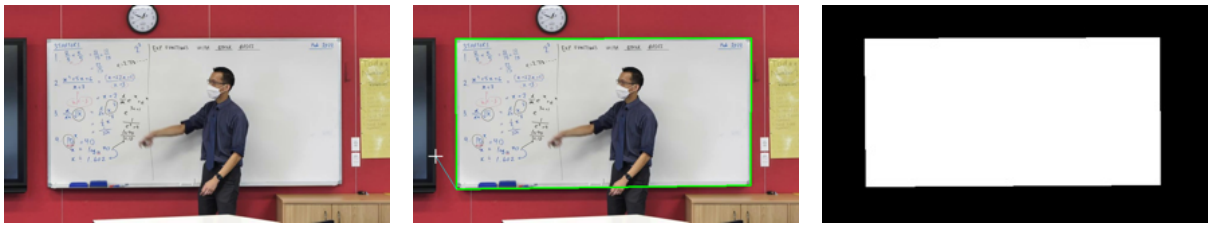
Creating a dataset for an image segmentation model can be a very tedious process, as not only do you need to find and download all the data individually, you also need to create and annotate segmentation masks for each data sample, and format it correctly to match the input layer of the model. Therefore four Python scripts are created to help automate the process.

The first step is finding the data. This process is hard to automate as the data need to be carefully selected to know what the model might be able to learn. Because of this, all still images are manually searched for and downloaded from Google, and all videos are manually found on Youtube. To download the videos from Youtube, a script is created which takes in a link to the video, and then downloads it in the resolution of 640x360, 30 frames per second, in the video format of mp4 with no audio.

Next, to convert the videos into individual images, a script is created to cut a single video into its individual frames. Each image and each set of video frames are then saved in its own folder with a unique name. Since many videos also consists of intros, outros, and various other screen content, these section have to be removed manually by going

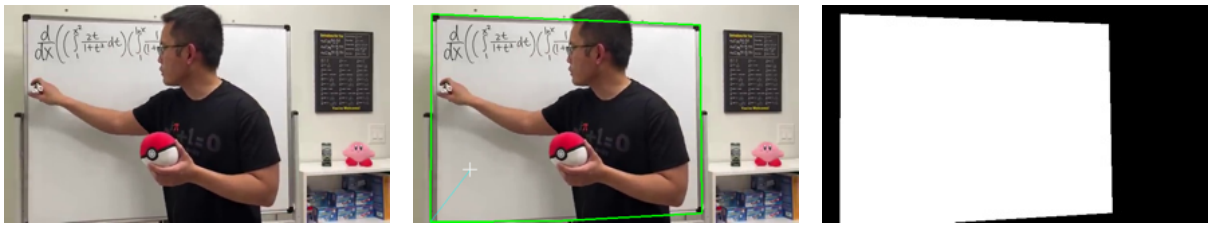
through all the frames and removing them. This is not an optimal solution as its quite time consuming and error prone to leaving frames where the whiteboard is not shown in the frame, resulting in wrong training data. Creating a script to do this automatically is very hard and not in the scope of this project.

Once the raw data is downloaded, extracted, and saved to individual folders, the truth mask is created with a script which takes in a single frame and then makes the user draw the outline of the whiteboard. This only needs to be done for a single frame of the video, since the whiteboard doesn't change place in any of the video data. The outline is drawn with the mouse cursor by clicking each corner of the polygon containing the whiteboard. To improve this experience, undo and reset functionalities are created to make miss clicks less punishing as the data is quite low resolution and it's sometimes hard to determine the exact outline of the whiteboard. When the outline is drawn and ready to be saved, a white mask is drawn on top of a black image withing the boundaries of the outline. It's then saved to a new folder with the same unique name as the image folder to easily match images with its corresponding mask.



(a) *Raw image* **(b)** *Image with drawn outline* **(c)** *Generated segmentation mask*

Figure 6: *Mask creation of a simple image with a rectangular whiteboard outline*



(a) *Raw image* **(b)** *Image with drawn outline* **(c)** *Generated segmentation mask*

Figure 7: *Mask creation of an image with a non-rectangular whiteboard outline*

After the masks have been created for all images, another scripts goes through each folder of images and randomly chooses 120 images resulting in a total of 8279 images. This number is not that important and is mostly chosen based on hardware limitations. Since a lot of still images are used, these images will then be duplicated 120 times, in contrast to the video frames which most likely will result in 120 different frames of the video. Each image is matched with its mask and resized to 256x256 pixels and then saved as a Numpy array. This array is then saved as a Numpy file which avoids having to write

each image to disk, and allows for quick loading when starting a training session instead of having to read each image one by one again. The Python script which creates this dataset also has two data loaders which can be imported for training. One data loader for the training set consisting of the first 90% of the data, and one for the validation set consisting of 10% of the data. The image data also needs to be augmented, but this is done during training as described in the next section.

3.4 Augmentation

To accommodate for the relatively small dataset image augmentation is used extensively. Instead of doing all the augmentations directly on the dataset before training, it is done in realtime while training. This avoids having to load in too many images at the start of a training session which is helpful when training with hardware limitations. It also make overfitting the data a lot harder, as the model will never see the same exact image twice. This realtime augmentation is only done on the training set, and not on the validation set. The validation set is only augmented once in the beginning to keep the validation set consistent throughout the training process. While the model could probably learn the same either way, it's important to have a steady metric of the validation set to efficiently evaluate how the model is progressing during training.

The augmentations used on the dataset are:

Horizontal Flip Flipping the image horizontally is an easy an obvious augmentation as it doesn't affect the image semantically in anyway but still in theory doubles the amount of images in the dataset. A vertical flip could also be used as most whiteboards still looks like whiteboards when flipped vertically. But since the surroundings of the whiteboard, such as people, chairs, and doors, are also important for the model, and these objects does change when flipped vertically, it is not used for this project.

Affine Using various affine transformation helps the model learn different sizes of whiteboards. The affines used are scale, translate, rotate, and shear. To not confuse the model too much, all affines are limited to mostly keeping the original image within frame, to not loose parts of the whiteboard. The edges of the whiteboard are an important factor of recognizing a whiteboard. Without any edges it can quickly look like just a wall or any other object with a white surface.

A downside of using affines is that new space becomes available around the original image. This requires a decision of what should occupy this new space. Often some kind of reflection or mirroring is used, but since this might result in very odd looking whiteboards, a solid black color is chosen in this project. This creates the side effect that the model gets used to seeing this black color around the image which does not resemble how it looks in reality. What the exact implications of this is, is hard to tell, but this could be subject to change if better alternatives exist.

Perspective Changing the perspective of the images has the same effects of the affines, but also teaches the model about detecting whiteboards from different viewing angles which is important as the whiteboards will not always face the camera.

Color Color augmentations such as changing brightness, saturation, contrast, and hue are used to simulate the different lightings a whiteboard can be in. Sometimes it's placed in a dark room, sometimes it's in a room with very blue light, and sometimes the sun is shining directly on the whiteboard. It also teaches the model about the different shades a whiteboard comes in, and different shades its surroundings can be in. At last, it teaches the model about how an image can look very different depending on the camera used, especially with webcams as they're often relatively low quality and lose details of the actual lighting in the room.

Blur and noise To accommodate for differences in video quality Gaussian blur and Gaussian noise is used. As explained previously, video recorders and webcams differs in quality, and will often not give a super sharp image depending on many factors. Therefore blur and noise is added to simulate these differences.

Cutout Since the model can't expect that the whole whiteboard is always visible, black and white cutouts are used to hide parts of the image. This is the case when a person is standing in front of the whiteboard, something is placed on the whiteboard, an object is obscuring a part of the whiteboard, or some of the whiteboard is out of frame. The cutouts varies in size, but are limited to a max size to not cover too much of the whiteboard. As explained previously, if too much of the whiteboard is missing, it's very hard to determine the exact outline of it, even for humans, and it will make the model too uncertain and result in blurry edges.

Superpixels Superpixels are used to add a bit more variety to the image. Superpixels are based on an algorithms which randomly clusters pixels based on their color similarity and proximity. This doesn't change the image very much, but helps a bit with overfitting.

Grayscale Converting the image to grayscale is the last augmentation used in the pipeline and is always applied. As described in the model architecture, the model is able to learn better when the image is in grayscale, but to still get the desired effects of the color augmentations, this is done after all other augmentations are applied.

In the figure below, 10 examples are shown of what these augmentations combined can create from a single image.

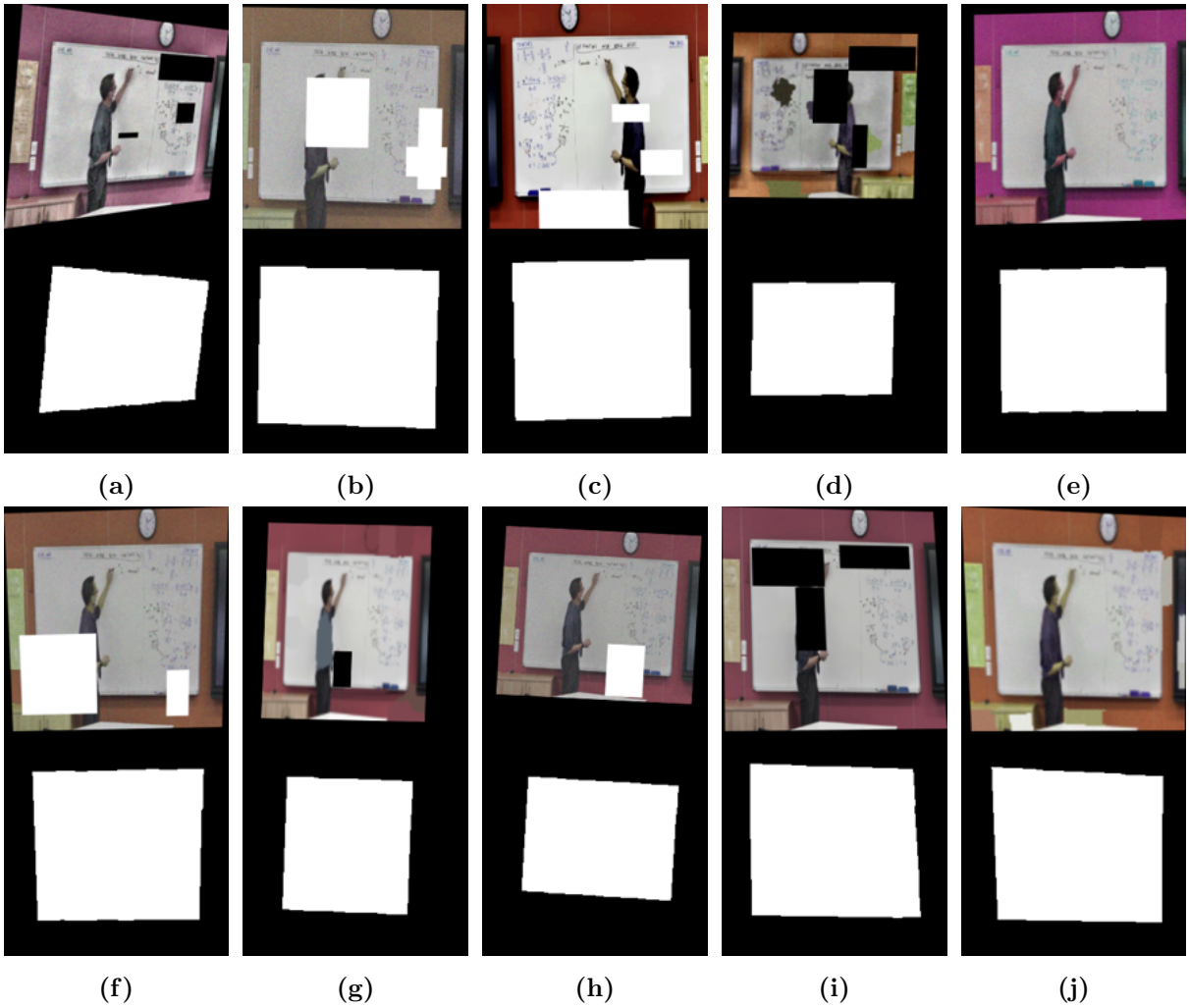


Figure 8: *Examples of augmentations done on a single image and its corresponding mask*

While these are the augmentations used during this project which are roughly fine-tuned to the specific dataset, it's a very flexible area, and adjusting some parameters or adding and removing some augmentations might result in a better model.

4 Tools

This sections describes the primary tools used throughout the project. For each tool the general purpose of the tool is described and briefly how it's specifically used during the project.

4.1 Python

Python is used as the programming language for all code written. Python is a very simple but powerful language as it's very easy to learn, read, and write, and at the same time allows for high performance code by binding to underlaying C and C++ code. Python is also widely acknowledged as one of the best and most used languages for machine learning. There's a lot of robust libraries created to make the development process of machine learning as fast and simple as possible. A large portion of research and documentation done around the subject of machine learning is also done with Python. This removes a lot of barriers when doing research as you often need to try out what other people have already tried, and not having to recreate an implementation from scratch in another language saves development time.

4.2 Numpy

Numpy is a Python library which provides a ton of high level functions to manipulate multi-dimensional data efficiently. The library makes use of a binding to the C language to make use of parallelism in the CPU with the use of SIMD instructions. Having low level access to the CPU makes it multiple magnitudes faster to do computations on large multi-dimensional arrays instead of writing raw loops in Python.

Computer vision and image segmentation requires a lot of processing of multi-dimensional arrays as images consists of pixels in the form of WIDTH x HEIGHT x CHANNELS, and sometimes processing of arrays of images of this form is needed, like when feeding a batch of images during training of the neural net. Therefore having a library like Numpy to help working with these data structures is really helpful and often necessary to achieve high performance.

4.3 OpenCV

OpenCV is a library which provides functions to do computer vision in real time. It makes it easy to read and manipulate frames of images and videos. Some examples of its more simpler functions is converting RGB to grayscale, resizing images, drawing on images, and using various filters such as Gaussian blur. It also provides a lot of advanced functions like edge detection, contour recognition, and perspective transformations. The Python implementation of OpenCV is tightly integrated with Numpy which allows for high performance, even with the heavy computations used in computer vision.

In the project, OpenCV is used in pre-processing to read frames from and image or video file or directly from the webcam and then resizing and converting the color format to be able to feed it to the neural net. In post-processing the output prediction is converted from a binary black and white image to a series of points resembling

the polygon outline of the whiteboard. This is done with the use of the functions: `findContours`, `convexHull`, `approxPolyDP`, `getPerspectiveTransform`, and `warpPerspective`.

It's also used extensively in the data gathering pipeline to split videos into individual frames, and generating the truth masks for training. At last, it's used to test models by showing different stages of the prediction pipeline, from the raw input, to the raw prediction, and then the various stages of post-processing.

4.4 PyTorch

PyTorch is Python library for machine learning. It's based on the Torch framework which is developed by Facebook. The framework provides a large set of tools to create machine learning models, and for training and testing models. Another popular choice of framework for this purpose is TensorFlow developed by Google. PyTorch is chosen rather than TensorFlow due to its easy documentation, more research friendly and pythonic code architecture, and easy control of GPU usage which is important when working with limiting GPU power.

4.5 Tensorboard

TensorBoard is a machine learning tool which provides various metrics and visualization such as accuracy and loss graphs when creating and training a machine learning model. The tool is created by TensorFlow but PyTorch has its own implementation which makes it easy to use even when using PyTorch.

The TensorBoard is used to visualize how the model improves by graphing validation accuracy and loss, and training accuracy and loss. Having a graph of the training progress makes it a lot easier to detect overfitting and underfitting of data, and makes it possible to better compare different models against each other, The tool was also used to show a sample of the predictions made by the model during each epoch of training. This helps to detecting obvious flaws in models early in training which can save a lot of time. It also made it possible to see differences in how models perform when approaching the training limit where the model no longer improves significantly.

4.6 Albumentations

Albumentations is a Python library for fast and flexible image augmentation for computer vision tasks. It offers various augmentations such as affine transformations: rotation, flip, scaling, etc. It has various Dropout transforms to create noise and cutouts of images, and more semantic augmentations like different blurs, color shifts and distortions. The library also developed to work with PyTorch making it an obvious choice for this project.

The library is used to augment the dataset to achieve a larger dataset. This helps solves the problem of overfitting with a small dataset and help the model generalize better.

5 Experiments

5.1 Training

For training the model, a GTX 1080 is used which is a medium to high-end graphics card with 8 GB of GDDR5X ram. Since the model is doing binary classification, Binary cross entropy is used as loss function. The Adam Optimizer, or Adaptive Moment Estimation algorithm, is used to guide the training process by adjusting learning rates and weights. A scheduler is used to decrease the learning rate by a multiplier when the training process reaches a plateau and hasn't improved in a certain number of epochs. This forces the optimizer to take smaller steps towards minimizing the loss further.

A simple accuracy metric is used to compare different models against eachother. This accuracy calculates the average difference in probability of each predicted pixel compared to the ground truth mask:

$$Accuracy = 1 - \frac{\sum_{i=1}^{n} |Y - \hat{Y}|}{n \cdot W \cdot H} \tag{1}$$

W and H are the width and height of the images (256x256), n is the number of images in a batch, Y denotes the ground-truth mask, and $\hat{Y}$ is the predicted mask.

To train the model, a Python script is created with various global variables that dictates the learning process environment. These variables include:

- The name of the model, e.g. `att_unet_bce`
- The input and output format of the images which are both 256x256x1
- The amount of epochs to train, or how many times the full dataset is looped through. This is set to 200, but manually stopped around 80 since the model stops improving significantly after this point.
- Batch size which is how many images are fed to the model at a time. This is set to 16. The higher the number, the faster training is, but it also requires more VRAM. A higher batch size also results in worse generalization by the model.
- The initial learning rate which is set to 0.001.
- Patience and multiplying factor of the learning rate scheduler. Patience is set to 3 epochs, and multiplying factor is set to 0.5.
- Training and validation set percentage which sets how much of the dataset should be used for training, and how much for validation. This is set to 10%.

For each epoch of training the full training set is looped through in steps of batch sizes, followed by a loop through the validation set. To get the best results, the training set is shuffled before each epoch to make the model more robust and avoid over- and underfitting. After each epoch, the average loss and accuracy is calculated for the training and validation set. If the model has improved by lowering the average validation loss, it's is saved as a new unique checkpoint to easily try different stages of the model

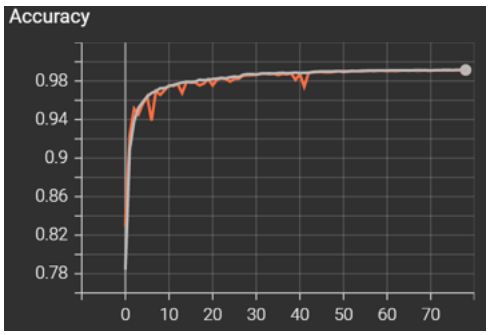
after training.

To closely follow the progress of training, various outputs are written to the console. This includes showing the current epoch and showing progress bars during training and validation which also outputs the current batch loss, batch accuracy, and how many images per second is currently processed. After training and validation is done for an epoch, a short summary is outputted showing average training loss and accuracy, and average validation loss and accuracy.

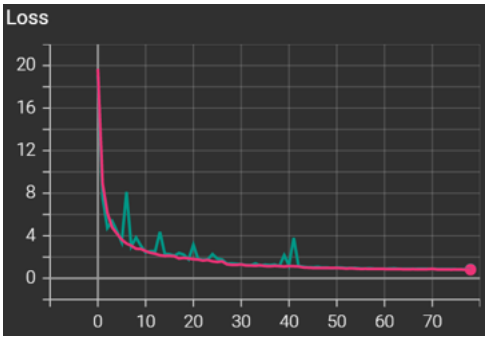
```
########## Epoch 6/200 ##########
TRAIN: 7456img [05:27, 22.76img/s, accuracy (batch)=0.967, loss (batch)=0.0587]
VALIDATE: 832img [00:08, 102.67img/s, accuracy (batch)=0.961, loss (batch)=0.0598]
SUMMARY | Loss: 0.0610, Accuracy: 0.9644, Val Loss: 0.0559, Val Accuracy: 0.9657
Model improved, saving to ./saves/att_unet_bce2_6.pt
########## Epoch 7/200 ##########
TRAIN: 7456img [05:06, 24.33img/s, accuracy (batch)=0.958, loss (batch)=0.0754]
VALIDATE: 832img [00:08, 103.31img/s, accuracy (batch)=0.962, loss (batch)=0.0596]
SUMMARY | Loss: 0.0558, Accuracy: 0.9671, Val Loss: 0.0704, Val Accuracy: 0.9596
########## Epoch 8/200 ##########
TRAIN:  48%|                    | 3552/7452 [02:31<03:20, 19.42img/s, accuracy (batch)=0.96, loss (batch)=0.0849]
```

Figure 9: *Image of command line output during training*

These metrics are not only outputted to the console, but also logged to TensorBoard to get a visual graph. It's important to match training and validation metrics against each other visually to detect overfitting, underfitting, and stagnation. Graphing the metrics in TensorBoard also makes comparing models to each other easy.



(a) *Accuracy graph where orange is validation accuracy and gray is training accuracy*



(b) *Loss graph where teal is validation loss and magenta is training loss*

Figure 10: *Examples of accuracy and loss metrics is graphic with TensorBoard*

A last thing which is done during validation, is adding a sample of 16 images from the validation set and model's output prediction to TensorBoard to get an insight in how the model is doing. This is especially helpful when making large changes to models, as sometimes error and flaws occurs. By showing a set of samples from the validation set, it quickly becomes obvious that a certain model does not perform very well, and can save a lot of training time.

(a) *Image samples from validation set*

(b) *Samples of model predictions*

Figure 11: *TensorBoard image grid of 16 random samples from the validation set for a specific training epoch*

5.2 Testing

Testing the quality of a model for this project needed more than just graphs and metrics. While accuracy and loss is a great indicator of how a model performs, it does not show the differences in behavior on test data it has not seen before. Therefore, a Python script is created to visually show in real-time how it behaves. This scripts takes in a couple command line arguments when running the script in formatted as:

```
usage: test.py [-h] [-model [MODEL]] [-image [IMAGE]]
        [-video [VIDEO]] [-webcam]

options:
  -h, --help            show this help message and exit
  -model [MODEL], -m [MODEL]
                        Model name: <NAME>
  -image [IMAGE], -img [IMAGE], -i [IMAGE]
                        Takes image file: <000/005>
  -video [VIDEO], -v [VIDEO]
                        Takes video file: <008>
  -webcam, -cam, -wb, -w
                        Uses webcam
```

It takes as argument a name of the specific model which is tested, and a option between using a video, an image, or the webcam directly. Most of the time the video argument is used to test a premade short video composed of multiple short sections of

different test videos which are not in the training and validation set. But sometimes it is helpful to test specific still images and also the webcam to see how much it over-predicts on a casual room.

The output of the test script is two video windows: one which shows the final transformed perspective of the whiteboard, and one which is composed of six different panels that shows the different steps that the model goes through, from the raw image to the outline of the whiteboard. All six frames are in its resized shape of 256x256 resolution to fit in one single window.

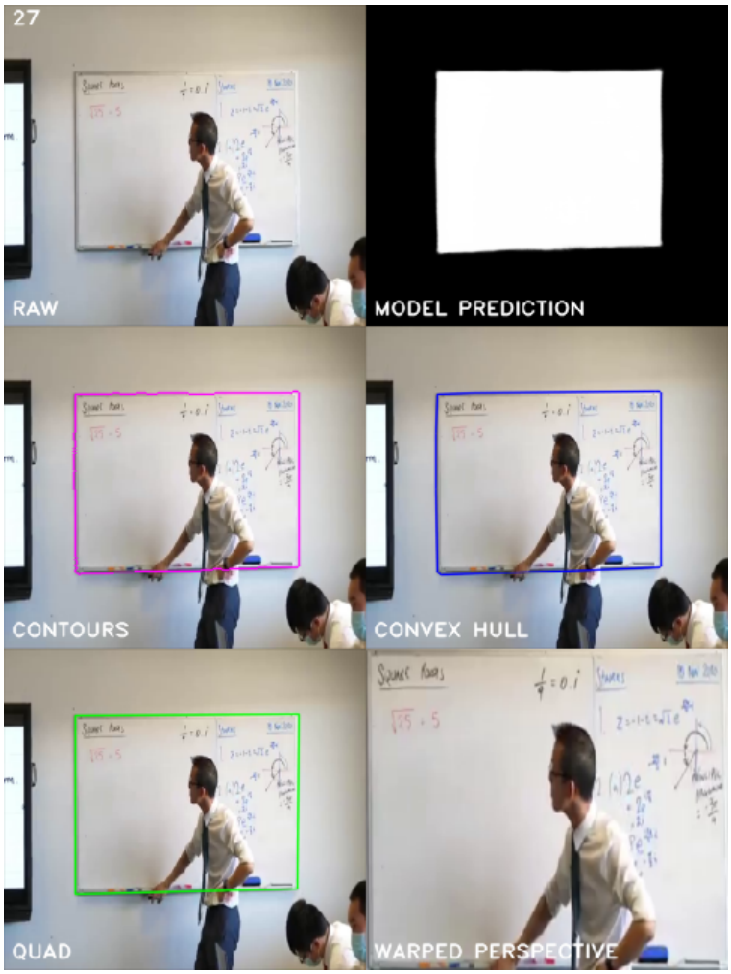


Figure 12: *Image of the six-panel window which shows the different steps the whiteboard model goes through*

The six panels are referenced in order from 1-6 going from the top-left corner to the bottom-right corner.

1. This panel shows the raw image after being resized to 256x256 pixels.
2. This panel shows the raw prediction made by the convolutional neural net as a binary image.
3. This panel shows a purple outline of the largest contour found from the prediction mask.

4. This panel shows a blue outline of the prediction after being run through the convex hull algorithm.

5. This panel shows a green outline representing the polygon lines of the outline after going through the Ramer-Douglas-Peucker algorithm.

6. This panel shows the perspective transformation by transforming the points of the polygon to fit the whole panel.

The window also shows the frames per second in the top-left corner to make sure that the model is able to run at roughly 30 frames per second.

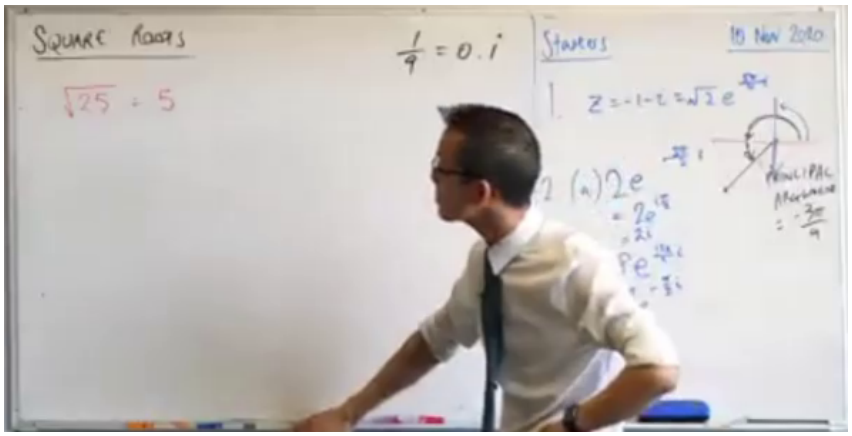


Figure 13: *Image of the warped perspective window which shows the final cutout of the whiteboard*

The window which shows the final warped perspective is created to show some kind of final output that the model could be used for. Due to the model predictions not being entirely stable, this window does move around and change in size a bit depending on the difficulty of the test video.

Additional features are developed for the test script which includes capturing snapshots and recordings of the six-panel window by pressing `c` to capture a snapshot, and `r` to start and stop a recording.

Pausing the video by pressing `p` is also available which stops the video from going to the next frames. The model keeps processing the current frame to allow for further input and testing

It's also possible to pause the post-processing of the model by pressing `q`. This makes the predicted outline of the whiteboard and the warped perspective window fixed and stable which might be a desired feature to have when using the model in practice. When the outline is paused, it's indicated by drawing the outline in red instead of green.

At last a feature to cover parts of the raw image is implemented. By clicking with the mouse on the raw image a black square appears which can be dragged around. Clicking with the mouse again removes it. By using the scroll wheel on the mouse, the square increases and decreases in size. This is helpful to test how the model behaves when large areas of the whiteboard are hidden such as corners and edges.

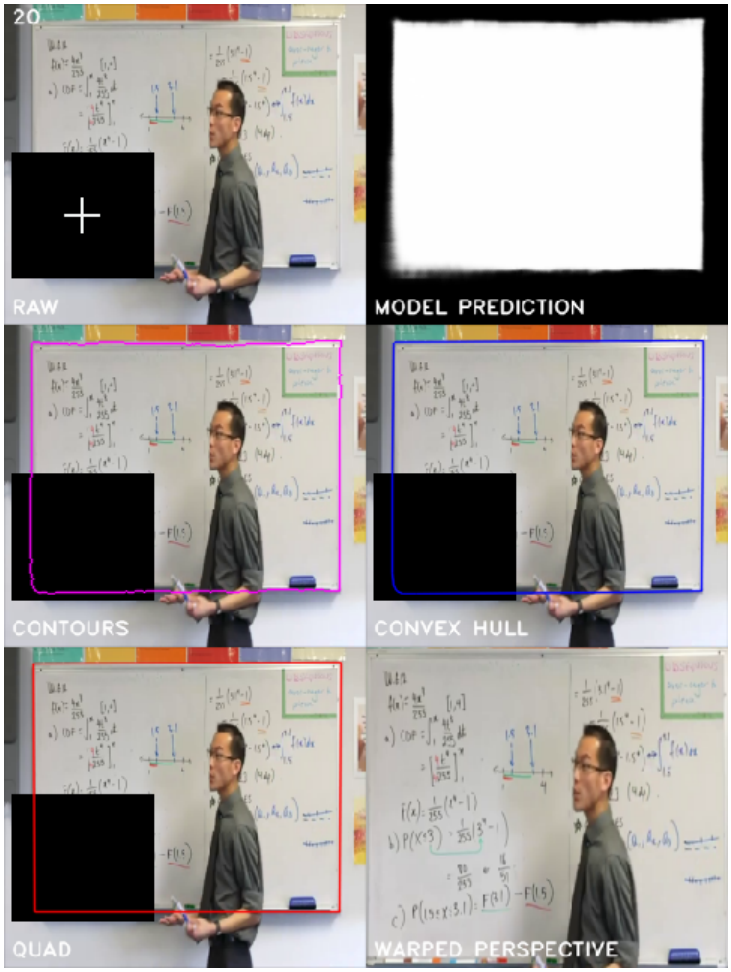
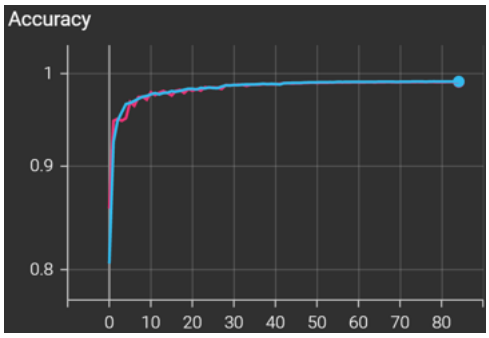


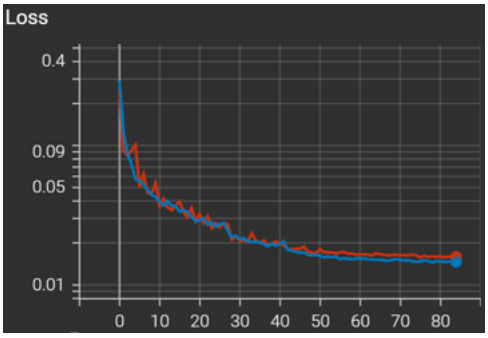
Figure 14: *Image of the six-panel window where post-processing is paused, and a black square is hovered over a corner of the raw image*

5.3 Results

Training the model for 85 epochs, with the best model checkpoint at epoch 78, resulted in an average validation loss of 0.01605 compared to the average training loss 0.01452, and an average validation accuracy of 0.9905 compared to the average training accuracy of 0.9911. Looking at Figure 15 it shows that training and validation metrics follows each other quite closely from the start which could indicate that little to no overfitting has occurred. They do seem to be very close to each other from the start, which is not something seen often. This might be due to the validation set looking a lot like training set.



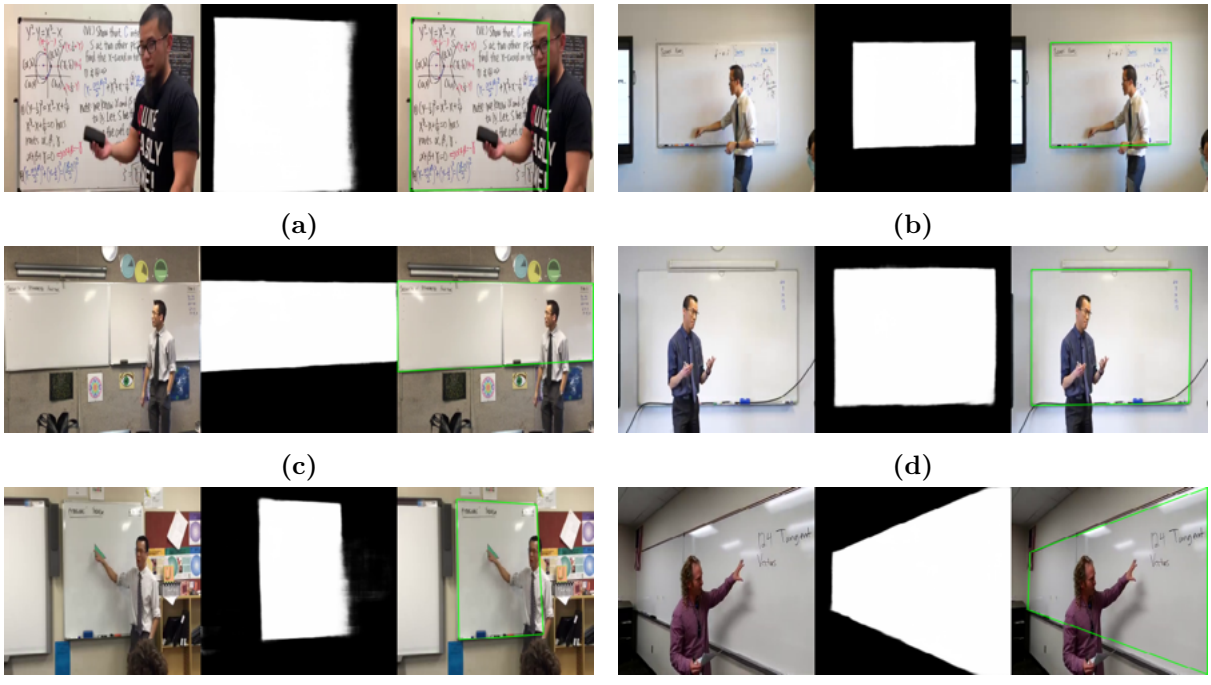
(a) *Logarithmic accuracy graph where magenta is validation accuracy and blue is training accuracy*



(b) *Logarithmic loss graph where orange is validation loss and blue is training loss*

Figure 15: *Accuracy and loss graphs after training for 85 epochs*

In Figure 16 samples from the testing data is shown which the model has not seen before during training. The results are quite good, though for (e) the smartboard to the left creates some uncertainty, and for (f) the outline fitting is not quite right. This is one of those cases where two corners of the whiteboards are out of frame, which creates a six sided polygon instead of a quadrilateral, but this was ignored in this case and resulted in a quadrilateral outline by the post-processing step anyway.



(a) **(b)** **(c)** **(d)** **(e)** **(f)**

Figure 16: *Model predictions on test data showing the raw image, the neural net prediction, and the post-processing result*

It's also interesting to test the model on how it behaves when large chunks of the whiteboard is missing. In Figure 17 two examples are shown: one where most of the edge is missing, and one where most of the corner is missing. The prediction is a lot better

when it's just a part of the edge missing, but gets unsure when the corner is missing. This is expected and hard to avoid, as even for humans this is a hard task to predict exactly.

(a) *Edge blocked by large solid black square* **(b)** *Corner blocked by large solid black square*

Figure 17: *Model predictions on test data with blocking square hiding parts of the whiteboard*

6 Improvements

In this section it's discussed what could further be done to improve the model. This include improvements to the model within the scope of the project, and also how one could extend it to solve additional problems and create a more generalized model.

6.1 Data

As briefly described in the data section, the dataset is quite limited to what is publicly available and accessible online. The result of this is a model which is quite good at detecting whiteboards in school environments, and whiteboards that stand alone and aren't surrounded with many other objects. When the whiteboard is placed in a room filled with other objects it gets more unsure about its predictions. Its predictions are also quite greedy which means it will falsely identify non-whiteboards as whiteboards such as doors, white walls, and drapes - almost anything which is rectangular and white. To improve this, the dataset could be extended to include thousands of unique data sources instead of only 67 which are used in this project. Ideally this data should include offices, homes, and conference rooms captured from the perspective of different types of webcams and other video capturing devices to get the best result for its intended purpose.

6.2 Augmentations

The augmentations used for the dataset is only roughly fine-tuned to give good enough results with the tools available. As described in the augmentations section, it's a very flexible area choosing the right amount of augmentations.

Ideas to specific changes that could be made includes changing the black background color which occurs when doing the affine transformations. An easy way to solve this is to never scale the image down but only up. This would solve the issue by always keeping the original image in frame, but also create new issues, as always scaling the image up often removes parts of the whiteboard.

Another improvement would be to use random colors for the cutouts, instead of only black and white. Black and white is chosen based on the limitations of the Albumentations library which does not support random colors yet. But random colors would be more suitable, as objects blocking the whiteboard in reality would almost never be a solid black or white color. Or even better, random objects could be used to block instead of solid colors to better represent reality

Some other augmentations which could be tried, are other blurs and noises instead of Gaussian, such as ISO noise to simulate how cameras differs in quality.

It's hard to manually find the most optimized augmentations when working with hardware limitations, as you need to train the model fully every time a small change is made. By having more GPU power available you could create and automatic script which tries all possibilities within some range of probabilities, parameters and augmentations to find the best set of augmentations.

6.3 Hyperparameters

Hyperparameters references to the parameters which is used to control the learning process. This includes network topology features like input layer size, convolution sizes, and the amount of hidden layers. It also includes training parameters like batch size, learning rates, schedulers, and optimizers, which only has an effect on the model during training but can have significant implications to how a model performs, its speed, and its ability to generalize.

The hyperparameters used for this project are most likely not the best hyperparameters for this model. They're chosen based on only a few tests and experiments due to time constraints and hardware limitations. To make a more educated guess as to what the best hyperparameters are, an automatic process could be created to test multiple model trained on a range of different hyperparameters. Tools exist to solve this which integrates well with PyTorch such as Bayesian optimization, which makes it an obvious choice to try to improve the model.

6.4 Shape awareness

One problem with using a UNet model to predict whiteboards where objects can be blocking parts of the whiteboards, is that the model is made to classify individual pixels and does not have a broader understanding of the context the pixels are in. This means that when someone is standing in front of the whiteboard, the model would ideally not classify the pixels of the person as whiteboard, but still learns to do from the data its trained on. This creates uncertainty since it's not that obvious whether it should classify a person as whiteboard when not standing in front of it. Combining this with additional noise, the model will often not predict a sharp quadrilateral or polygon, and instead result in a fussy and noisy prediction.

A way to solve this is adding some kind of shape awareness to the model which is better at always predicting sharp quadrilateral shapes. Though this is out of scope of this project, and would need additional research.

With the assumption that the outline of the whiteboard can always be described as a set of points in a polygon and always a single whiteboard per image, one could also imagine that instead of having a image segmentation model outputting a segmentation mask which classifies each pixel as whiteboard or non-whiteboard, the model would directly output a sequence of points representing the outline. This would eliminate the post-processing step of finding the outline with computer vision. This solution is purely hypothetical, and it's not known whether it would be a better alternative, but in theory this would force the model to always predict a shape of a whiteboard with sharp edges.

6.5 Blackboards and glassboards

To extend the model's practical use, the dataset could also include images of blackboards - and even glassboards. Whiteboards does seem to be the most used drawing surface now a days, but glassboards are becoming more and more attractive, and blackboards are still used in much of the world. Therefore, it seems like a obvious extension to the model, to make it work for all drawing surfaces. But this might not come without its own problems, as the distinction between what is a drawing surface and what is not becomes very vague.

Many questions arises like: Is it a TV or a blackboard; a blank canvas or a whiteboard; a window or a glassboard, etc. These problematic implications of adding blackboards and whiteboards are purely hypothetical, and might just require more parameters in the model and more data - after all, as humans, we're quite good at making the required distinctions, so a sophisticated neural net should be able to do so too.

6.6 Lens distortions

The images used for the dataset are all undistorted, meaning that a linear line in real life is also captured as linear with the camera. This limits the model to a specific type of camera outputs. Many cameras now a day are using wide angle lenses, fish-eye lenses, and even 180+ degrees lenses to capture as much of a space as possible. This is quite beneficial in conference rooms where multiple people sit very close to the camera. Instead of having people sit further away from the camera, you can use some kind of wide angle lens. These types of cameras outputs a raw image which is distorted resulting in curved lines.

Creating the truth masks for this kind of data would require a new solution, as it's not possible to draw the outline of the whiteboard as a simple polygon. This could be solved by using Bézier curves instead of linear lines to draw the outline, or using a brush to fill in the area of the whiteboard. While these solutions would work, it would make data generation even more time consuming

6.7 Instance segmentation

A problem mentioned in the data section is when two or more whiteboards are joined together with shared edges or just multiple whiteboards are in the same image. In the model developed for this project, this is recognized as a single whiteboard which can create problems since the edges are no longer a hard semantic boundary of the whiteboard which the model can look for. One way to solve this is using an instance segmentation model instead of semantic segmentation model, which can not only detect what is whiteboard board in an image, but also detect multiple instances of whiteboards. Using instance segmentation is in general a lot more computational heavy than just semantic segmentation. This means that achieving 30 frames per second on medium range hardware is gonna be very hard. Instance segmentation models such as Mask-RCNN[2] is only able to achieve up to 5 frames per second, though some newer models have been developed like YOLACT[1] which can achieve up to 33.5 frames per second on a very high end graphics card.

7 Conclusion

In this project, a model consisting of an Attention U-Net model and computer vision is proposed to detect whiteboards in real-time videos. Using an Attention U-Net makes it possible to train an image segmentation model on a relatively small dataset with the help of data augmentation. Computer vision is a great tool for transforming the classification image made by the Attention U-Net into a few set of points representing the outline of the whiteboard. Having the minimum amount of points in the outline is quite useful when having to further process the area of interest, which is the whiteboard. Though, hypothetically there's no reason why a single network architecture couldn't combine these two tasks, and output the outline directly from the neural net.

While the model is not perfect, it does perform quite well on data that has the same features as the dataset which consists of large amount of whiteboards in school environments. For it to work on all kinds of whiteboards in different environments like offices and conference rooms, more data needs to be added to the dataset.

The tools and Python scripts created for the project have been really helpful at reducing time consuming tasks. Though, if the model had to be improved further additional tools would be needed and the existing tools would need a rework to be easier to work with. They're not created to be easy to understand for other people and do multiple things at the same time. They're created solely with the purpose of being functional a solve a certain task.

There's many improvements that could be made to the model like more data, better data augmentations, and more optimized hyperparameters, but due to the scope of the project and hardware limitations some trade-offs had to be made between perfection and good enough. The model is determined to be good enough, since the requirements of predicting 30 frames per seconds and being able to accurately detect whiteboards on test data which the model hasn't been trained with are met.

References

[1] Daniel Bolya et al. *YOLACT: Real-time Instance Segmentation*. 2019. DOI: 10.48550/ARXIV.1904.02689. URL: https://arxiv.org/abs/1904.02689.

[2] Kaiming He et al. *Mask R-CNN*. 2017. DOI: 10.48550/ARXIV.1703.06870. URL: https://arxiv.org/abs/1703.06870.

[3] Ozan Oktay et al. *Attention U-Net: Learning Where to Look for the Pancreas*. 2018. DOI: 10.48550/ARXIV.1804.03999. URL: https://arxiv.org/abs/1804.03999.

[4] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. DOI: 10.48550/ARXIV.1505.04597. URL: https://arxiv.org/abs/1505.04597.